Recognizing Patterns in Streams with Imprecise Timestamps

Haopeng Zhang, Yanlei Diao, Neil Immerman
Department of Computer Science
University of Massachusetts, Amherst

ABSTRACT

Large-scale event systems are becoming increasingly popular in a variety of domains. Event pattern evaluation plays a key role in monitoring applications in these domains. Existing work on pattern evaluation, however, assumes that the occurrence time of each event is known precisely and the events from various sources can be merged into a single stream with a total or partial order. We observe that in real-world applications event occurrence times are often unknown or imprecise. Therefore, we propose a temporal model that assigns a time interval to each event to represent all of its possible occurrence times and revisit pattern evaluation under this model. In particular, we propose the formal semantics of such pattern evaluation, two evaluation frameworks, and algorithms and optimizations in these frameworks. Our evaluation results using both real traces and synthetic systems show that the event-based framework always outperforms the point-based framework and with optimizations, it achieves high efficiency for a wide range of workloads tested.

1. INTRODUCTION

Large-scale event systems are becoming increasingly popular in domains such as system and cluster monitoring, network monitoring, supply chain management, business process management, and healthcare. These systems create high volumes of events, and monitoring applications require events to be filtered and correlated for complex pattern detection, aggregated on different temporal and geographic scales, and transformed to new events that represent high-level meaningful, actionable information.

Complex event processing (CEP) [1, 2, 4, 7, 8, 15, 17, 18, 22, 23] is a stream processing paradigm that addresses the above information needs of monitoring applications. CEP extends relational stream processing with a sequence-based model (in contrast to the traditional set-based model), and hence considers a wide range of pattern queries that address temporal correlations of events. Prior research [1] has shown that such pattern queries are more expressive than selection-join-aggregation queries and regular languages.

Existing work, however, fundamentally relies on two assumptions. First, the occurrence time of each event is known precisely. Second, events from various sources can be merged into a single stream such that a binary relation (denoted by $\prec$) based on the event occurrence time gives rise to a total order [1, 8, 14, 18, 23] or a partial order [2, 3, 4, 7, 15, 22] on the event stream. These assumptions are used in systems that consider either point-based or interval-based event occurrence times; the only difference between them is in the specifics of the definition of of the binary relation ($\prec$), but not in the underlying assumptions.

We observe that in many real-world applications, the above assumptions fail to hold for a variety of reasons:

Event occurrence times are often unknown or imprecise. For instance, in RFID-based tracking and monitoring, raw RFID data provides primitive information such as ($time$, tag_id, $reader\_id$) and is known to be lossy and even misleading. Meaningful events such as object movements and containment changes are often derived using probabilistic inference [16, 20]. The actual occurrence time of object movement or containment change is unknown and can only be estimated to be in a range with high probability.

Event occurrence times are subject to granularity mismatch. In cluster monitoring, for instance, a commonly used monitoring system, Ganglia [10], measures the max and average load on each node once every 15 seconds, whereas the system log records the jobs submitted to each node using the UNIX time at the granularity of a microsecond. To identify the jobs that max out a compute node, one has to deal with the uncertainty that the peak load can occur anywhere in a 15-second period, making it hard to judge whether it occurred before or after the submission of a particular job. That is, the temporal relationship between a load measurement event and a job submission event is non-deterministic, neither in total order or in partial order (which we show formally in Section 2).

Events collected from a distributed system are subject to the clock synchronization problem. Consider causal request tracing in large concurrent, distributed applications [5, 11], which involves numerous servers and system modules. As concurrent requests are serviced by various servers and modules, an event logging infrastructure generates event streams to capture all system activity, including thread resource consumption, packet transmission, and transfer of control between modules. The challenge is to demultiplex the event streams, accounting resource consumption by individual requests. In particular, the clock synchronization problem makes it hard to order events relevant to each request properly [11].

In this paper, we address pattern query evaluation in streams with imprecise occurrence times of events—-such events preclude the models based on a total order or partial order of events. A starting point of our work is to employ a *temporal uncertainty model* that assigns a time interval to each event for representing all of its possible occurrence times and to revisit pattern query evaluation under this new temporal model. Our technical contributions include:

Formal Semantics. We propose the formal semantics of pattern query evaluation under the temporal uncertainty model, which includes two parts: pattern matching in possible worlds of deterministic timestamps, and match collapsing into a succinct result format. This formal semantics offers a foundation for reasoning about the correctness of implementations and yet provides useful results.

Evaluation Frameworks and Optimizations. We propose two evaluation frameworks that generate query matches according to the formal semantics, but without enumerating a large number of possible worlds. The first evaluation framework, called point-

VLDB '10, September 13-17, 2010, Singapore

based, requires minimum change of an existing pattern query engine, hence easy to use. The second evaluation framework, called event based, directly operates on events carrying uncertainty intervals. We present evaluation methods in these frameworks, prove their correctness, and further devise optimizations to improve efficiency.

Evaluation. Our evaluation using both real traces in MapReduce cluster monitoring and synthetic streams yields interesting results: (*i*) Despite the simplicity of the point-based framework, its performance is dominated by the event-based framework. (*ii*) Queries that use a traditionally simpler strategy to select only the first match of each pattern component, instead of all possible matches, actually incurs higher cost under temporal uncertainty. (*iii*) Optimizations of the event-based framework are highly affective and offer thousands to 10's of thousands of events per second for all queries tested. (*iv*) Our event-based methods achieve high efficiency in the case study of cluster monitoring despite the large uncertainty intervals used.

2. MODEL AND SEMANTICS

In this section, we provide background on pattern query evaluation, present our temporal uncertainty model, and formally define the semantics of pattern query evaluation under our model.

2.1 Background on Pattern Queries

We begin by providing background on pattern queries [1, 4, 7, 18, 23] to offer a technical context for the discussion in the rest of the paper. A pattern query addresses a sequence of events that occur in order (not necessarily in contiguous positions) in the input stream and are correlated based on the values of their attributes. Below, we highlight the key features of pattern queries using the SASE language adopted widely in recent work [1, 13, 15, 23]. The overall structure of a pattern query is as follows:

```
PATTERN <pattern structure>
[WHERE  <pattern matching condition>]
[WITHIN <time window>]
[RETURN <output specification>]
```

Query 1 below shows an example in cluster monitoring: for each compute node in a MapReduce cluster, the query detects a map or reduce task that causes the CPU to max out. The PATTERN clause describes the structure of a **sequence pattern**, which in this example contains four events of the specified types occurring in order. The WHERE clause specifies further constraints on these events. The common constraints are **predicates** that compare an attribute of an event to a constant or compare the attributes of different events, as shown in Query 1. In addition, the WHERE clause can further specify the **event selection strategy**, e.g., using "skip till any match" in this query (which we discuss more shortly). The WITHIN clause restricts the pattern to a 15 second period. Finally, the RETURN clause selects the events to be included in the pattern match. By default, all events used to match the pattern are returned.

```
Query 1:
PATTERN SEQ(TaskStart a,CPU b,TaskFinish c,CPU d)
WHERE   a.taskId = c.taskId AND
        b.nodeId = a.nodeId AND
        d.nodeId = a.nodeId AND
        b.value > 95% AND
        d.value <= 70% AND
        skip_till_any_match(a, b, c, d)
WITHIN  15 seconds
RETURN  a, b, c
```

The event selection strategy addresses how to select the events relevant to a pattern query, not necessarily in contiguous positions in the input stream. For the purpose of this work, we consider two common strategies (while referring the reader to [1] for details of all possible strategies): (*i*) *Skip till next match* [1, 7]. In the pattern matching process, the next relevant event does not need to be contiguous to the previous one; all irrelevant events will be skipped until the next relevant event is read. This strategy makes the query insensitive to the presence of irrelevant events. (*ii*) *Skip till any match* [1, 15, 23]. This strategy relaxes the previous one by allowing non-deterministic actions on a relevant event: in one instance, it selects the event to extend the current partial match of the pattern; in another instance, it ignores this event and awaits a future event, potentially resulting in a different match. For Query 1, consider an event stream of five streams, denoted by "a, b_1, b_2, c, d". Skip till any match will return two matches that result from "a, b_1, c, d" and "a, b_2, c, d", respectively, whereas skip till next match will return only the former because the arrival of b_1 will move the matching process permanently forward towards c.

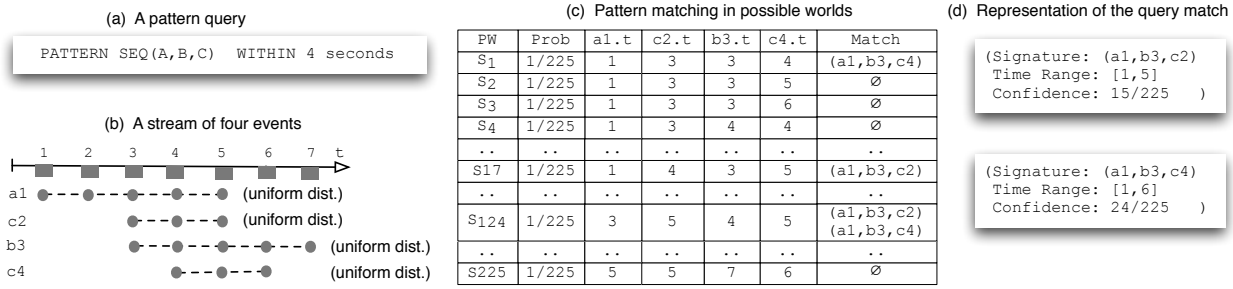
2.2 Temporal Uncertainty Model

We now switch to consider events with uncertain occurrence times and propose an event model that accommodates temporal uncertainty. As in most temporal data model research [6], we assume a discrete, totally ordered time domain T; without loss of generality, we number the instants in T sequentially as 1, 2, ... Each event represents an atomic occurrence of interest at an instant. However, the exact occurrence time of an event may not be available due to the reasons mentioned in the previous section. To address this issue, our model allows the event provider to specify an uncertainty interval, UI: $[lower, upper] \subseteq T$, to bound the occurrence time of an event, with an optional probability mass function $p : UI \rightarrow [0,1]$ to characterize the likelihood of occurrence in the uncertainty interval (by default, a uniform distribution is used).

In summary, an event in our model has the following format: $(EventType, EventId, UI : [lower, upper], (p : UI \rightarrow [0,1])?, Attributes)$, where $EventType$ specifies the attributes allowed in the events of this type and $EventId$ is the unique event identifier. For example, a_1=(A, 1, [10, 13], (v_1, v_2, v_3)) represents an event of type A and id 1, an uncertainty interval from time point 10 to time point 13, and three attributes required in this event type. If the occurrence time of an event is certain, we simply set the upper and lower bounds of the interval to the same point.

Ordering Properties. Given the temporal uncertainty model, it is evident that we cannot find a binary relation (denoted by $\prec$) based on the event occurrence time that ensures a total or partial order on an arbitrary event stream. Consider a (strict) partial order, defined to be a binary relation on a sequence $\mathcal{S}$ that is (1) irreflexive, $\forall e \in \mathcal{S}$, $\neg(e \prec e)$; (2) asymmetric, if $e_1 \prec e_2$ then $\neg(e_2 \prec e_1)$; and (3) transitive, if $e_1 \prec e_2$ and $e_2 \prec e_3$ then $e_1 \prec e_3$. Under the temporal uncertainty model, it is easy to construct an event stream with two events that violate the asymmetry requirement; that is, one possibility of their occurrence times entails $e_1 \prec e_2$, and another possibility of their occurrence times entails $e_2 \prec e_1$. Similarly, we can show that there exists no total order on events under this model.

Arrival order is a different issue. In data stream systems, out-of-order arrival is signaled if the arrival of events is not in increasing order of their occurrence times [19]. In our problem, there is no clear notion of "increasing order of the occurrence time", so we loosely define out-of-order arrival to be the case that e_1 is seen before e_2 in the stream but the earliest possible occurrence time of e_1 is after the latest possible time of e_2, i.e., $e_1.lower > e_2.upper$. To facilitate query evaluation, we assume that using buffering or advanced techniques for out-of-order streams [13, 19], we can feed events into the pattern query engine such that if e_1 is seen before e_2, with respect to the occurrence time, e_1 either completely precedes e_2 or overlaps with e_2 in an arbitrary way, i.e., $e_1.lower \leq e_2.upper$.



(c) Pattern matching in possible worlds

PW	Prob	a1.t	c2.t	b3.t	c4.t	Match
S_1	1/225	1	3	3	4	(a1,b3,c4)
S_2	1/225	1	3	3	5	∅
S_3	1/225	1	3	3	6	∅
S_4	1/225	1	3	4	4	∅
..	..	..	..	..	..	..
S17	1/225	1	4	3	5	(a1,b3,c2)
..	..	..	..	..	..	..
S_{124}	1/225	3	5	4	5	(a1,b3,c2) (a1,b3,c4)
..	..	..	..	..	..	..
S225	1/225	5	5	7	6	∅

(d) Representation of the query match

```
(Signature: (a1,b3,c2)
 Time Range: [1,5]
 Confidence: 15/225   )
```

```
(Signature: (a1,b3,c4)
 Time Range: [1,6]
 Confidence: 24/225   )
```

Figure 1: Semantics of pattern query evaluation under our temporal uncertainty model.

2.3 Formal Semantics under the Model

We next introduce the formal semantics of pattern query evaluation under our temporal uncertainty model. Our formal semantics includes two parts: pattern matching in a set of possible worlds, and collapsing the matches of possible worlds into query matches.

Pattern Matching in Possible Worlds. In our model, every event has several possibilities of its occurrence time, i.e., at consecutive time points $\{(t_j, p(t_j))|j = 1,\ldots,|UI|\}$. Given a sequence of events $\mathcal{S} = \{e_1,\ldots,e_i,\ldots e_n\}$ that fit in a time window, a unique combination of the possible occurrence time of each event, $(t_{ij}, p(t_{ij}))$, gives rise to a sequence S_k in which events have deterministic occurrence times and can be sorted by their occurrence times (conceptually). Borrowing the familiar concept from the literature of probabilistic databases, we refer to S_k as a *possible world* for pattern query evaluation, and compute its probability as $\mathbb{P}[S_k] = \prod_{i=1}^{n} p(t_{ij})$. We then perform pattern matching in every possible world S_k, as in any existing event system.

Example: Fig. 1(a) shows a sequence pattern with a 4-second time window (assuming that a time unit is a second). Fig. 1(b) shows a stream of four events, denoted by a_1, c_2, b_3, and c_4, and their uncertainty intervals on the time line, all using the (default) uniform distribution of the likelihood of occurrence. Since a_1, c_2, b_3, and c_4 have 5, 3, 3, and 5 possible occurrence times, respectively, there are 225 unique combinations of their occurrence times, hence 225 possible worlds. Fig. 1(c) shows these possible worlds, the probabilities of these worlds, and the pattern matching result in each possible world, strictly based on the query semantics for an event stream with deterministic occurrence times. As can be seen, a possible world can return zero, one, or multiple matches.

In general the number of events, N, that fit in a time window can be large. If on average each event has an uncertainty interval of size U, then there is an exponential number of possible worlds, $O(U^N)$.

Query Match Construction. The large number of possible worlds can cause a large number of match sets to be returned from these worlds. Returning all of them to the user (even if the computation is feasible) is undesirable. In our work , we instead present these match sets in a succinct way. More specifically, we collect the match set Q_k from each possible world S_k and proceed as follows:

- Union the matches from all match sets Q_k, $k = 1, 2, \ldots$
- Group all of the matches by *match signature*, which is defined to be the unique sequence of event ids in a match.
- For each group with a unique match signature, compute the (tightest) *time range* that covers all of the matches, and compute the *confidence* of the match as the sum of the probabilities of the possible worlds that return a match of this signature.

Finally, the triples, {(signature, time range, confidence)}, are returned as the query result at a particular time.

Example: In Fig. 1, the matches produced from the 225 possible words have two distinct signatures: The first one is (a_1, b_3, c_2). The tightest time range that covers the matches of this signature is [1,5]; e.g., the match from the possible world S_{17} is on the points (1,3,4) and that from S_{124} is on (3,4,5) . Further, 15 out of 225 possible worlds return matches of this signature, yielding a confidence of 15/225. The second signature is (a_1, b_3, c_4) with its time range and confidence computed in a similar way. The complete query result at t=7 is shown in Fig. 1(d).

3. POINT-BASED EVALUATION FRAMEWORK

Given our temporal uncertainty model and formal semantics of pattern queries under this model, we next seek an efficient approach to evaluating these queries. Evidently, the possible worlds semantics does not offer an efficient evaluation strategy since the number of possible worlds is exponential in the number of events in a time window. We next introduce efficient evaluation frameworks that guarantee correct query results according to the formal semantics, but without enumerating the possible worlds.

In this section, we introduce our first evaluation framework, called a *point-based* framework. Our design is motivated by the fact that existing pattern query engines take events that occur at specific instants, referred to as *point events*. If we can convert events with uncertainty intervals to point events, we can then leverage existing engines to do the heavy lifting in pattern evaluation. Our design principal is to require minimum change of a pattern engine so that the proposed framework can work easily with any existing engine.

There are three main issues to address in the design of a point-based evaluation framework: First, existing pattern query engines require that events be arranged in total or partial order based on their occurrence times. As stated in §2.2, under our temporal uncertainty model there is in general no total or partial order on events. As we convert such events to point events, what ordering property can we offer? Second, running an existing pattern query engine directly on the converted point event stream does not produce results consistent with our formal semantics. Our goal is to produce all the matches that would be produced from the possible worlds, referred to as the *point matches*. What is the minimum necessary change of an existing engine to do so? Third, without enumerating all possible worlds, how do we compute the time range and confidence for each unique signature of point matches? By addressing the above questions, we design an evaluation framework with three main steps, as shown in Fig. 2(a). The notation used in the following discussion is summarized in Table 1 in the appendix.

Step 1: Stream Expansion. We first illustrate a point event stream using the example in Fig. 1(b). To generate the point event stream, we (conceptually) iterate over all the time points, from 1, 2, ... At every point t, we collect each event e from the input whose uncertainty interval spans t, and inject to the new stream a point event that replaces e's uncertainty interval with a fixed timestamp t. In this example, the point event stream will contain a_1^1, a_1^2, a_1^3, c_2^3, b_3^3, a_1^4, c_2^4, b_3^4, c_4^4, ... (where the superscript denotes the occurrence time). As such, the new stream is sorted by the occurrence time.

Our implementation is more complex than the conceptual procedure above due to the various event arrival orders. Recall from §2.2

that the only constraint on the arrival order in our work is that if e_1 arrives before e_2, then with respect to the occurrence time, e_1 either completely precedes e_2 or overlaps with e_2 (in an arbitrary way), i.e., $e_1.lower \leq e_2.upper$. To cope with this, our implementation uses buffering (of limited size) to emit point events in order of the occurrence time. Let $e_1, \ldots, e_{n-1}$ denote the events in the past and e_n the newly arrived event. We create point events for all the instants in e_n's uncertainty interval and add them to a buffer (possibly containing other point events). Further, let *now* be a time range $[lower = max_{i=1}^{n}(e_i.lower), upper = max_{i=1}^{n}(e_i.upper)]$. Also assume that the maximum uncertainty interval size for the event stream is U_{max} (which can often be requested from event providers). Then given the arrival order constraint, we know that any unseen event must start after $now.lower - U_{max}$, labeled as t_{emit}. Now we can safely output the buffered point events up to t_{emit}.

Step 2: Pattern Matching. We next evaluate pattern queries over the point event stream by leveraging an existing pattern query engine such as [1, 15]. The challenge is to configure the engine properly and add minimum changes so that we can produce all the point matches that would exist in the possible worlds.

First, we show that the pattern query engine must be configured with the most flexible event selection strategy, "skip till any match", no matter what event selection strategy is actually used in the query.

Example. Fig. 2(b) shows all the time points of the four events in Fig. 1(b). We can also visualize the dots as point events arranged in increasing order of time. Consider all the point matches that start with a_1^2. While the formal semantics requires enumerating all possible words that involve a_1^2 (45 of them) to find those matches, a more efficient algorithm can directly search through the point events *in query order* and capture *all possible ways* of matching points from distinct input events. In this example, the point event a_1^2 matches the first component of the pattern (A,B,C). Then at time t=3, we would naturally select b_3^3 to extend the partial match to (a_1^2, b_3^3); at the same time, we would also skip b_3^3 to preserve the previous partial match (a_1^2). At t=4, we can select c_2^4 to produce a match (a_1^2, b_3^3, c_2^4), or select c_4^4 to produce a different match (a_1^2, b_3^3, c_4^4). Again, we can skip these events to preserve the partial match (a_1^2, b_3^3) so that it can be later matched with the c events at t=5. In addition, at t=4 we can select b_3^4 to match with a_1^2, yielding a new partial match (a_1^2, b_3^4), which again will be extended with the c events at t=5.

In summary, given a point event that generates an initial partial match of a pattern, we can use "skip till any match" to dynamically construct a direct acyclic graph (DAG) rooted at this event and spanning the point event stream, such that each path in this DAG corresponds to a unique point match starting from the root (some pruning may be needed to ensure correctness, as discussed below).

Next, we discuss the necessary changes of the function, $next()$, that a pattern query engine uses to match events with successive pattern components. For instance, the matching of a pattern (A, B, C) will invoke $next(\emptyset)$, $next((a_1))$, $next((a_1, b_3))$, ... until a match is generated. Our goal is to revise $next()$ so that we produce the *exactly* same set of point matches as in possible worlds. A simple change is to disable the use of the points of the same event to match different components of the pattern. For example, given a pattern (A, B, A), we cannot use the same points of an 'a' event to match the first and third components. We simply add an id check to $next()$ so that it ignores any point event sharing the id with any of those already included in the partial match.

A more significant issue is to support different event selection strategies used in queries. If a query uses *skip till any match*, we already have the correct set of matches (which we prove in Appendix B.2). However, if the query uses *skip till next match*, it means that the matching process should always select the next relevant event

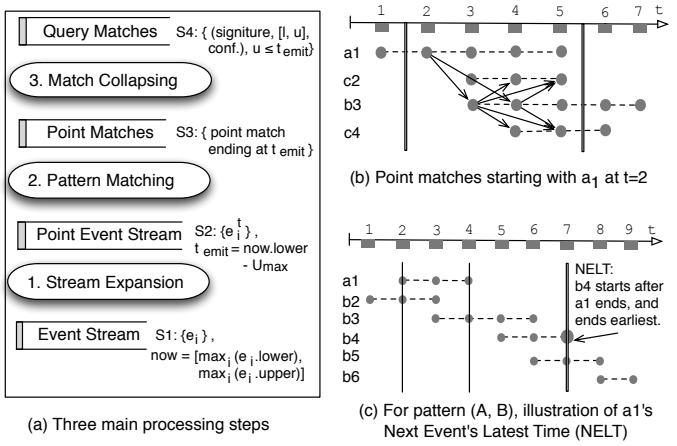


Figure 2: The point-based evaluation framework.

to match the current pattern component in consideration. More specifically, if multiple events in the stream can match the current pattern component, the *first* event in occurrence time should be selected to extend the partial match. While this selection strategy is easier to support than skip till any match in a deterministic world, it is actually more difficult under temporal uncertainty. Our algorithm above that blindly runs skip till any match on the point event stream may produce more matches than existent in the possible worlds.

Example. Fig. 2(c) shows a simple pattern (A,B) and an event stream with a_1 and five b events (in arrival order). Can any b event be the next to a_1 in some possible world? The answer is positive as long as we can find a possible world in which a point of a_1 precedes a point of the b event with no other b in between. It is easy to see that any b that overlaps with a_1, e.g., b_2 and b_3, can be the next event in some possible world. Further, b_4 and b_5 that start after a_1 ends still have a chance to be the next event in a possible world. For b_4, one such possible world contains b_2^2, b_3^3, a_1^4, b_4^5, ...; that is, b_2 and b_3 occurred before a_1, making room for b_4 to be the next event for a_1. For b_5, a possible world contains b_2^2, b_3^3, a_1^4, b_5^6, b_4^7 ... However, it is impossible for b_6^8 or any point of b_6 to be the next b in any possible world as they are always preceded by b_4^7.

The above example illustrates our notation of the *Next Event's Latest Time* (NELT), a timestamp associated with any event that has just been selected in a partial match. Consider a pattern ($E_1, \ldots, E_\ell$) and a partial match ($e_{m_1}, \ldots, e_{m_j}$), with e_{m_j} being the last selected event. Among all events that can match the next pattern component E_{j+1} and start after e_{m_j} ends, the event that ends the earliest sets the NELT of e_{m_j} using the upper bound of its interval. In the above example, with a_1 selected in the partial match, its NELT is set to $b_5.upper$ when b_5 is seen. That is, any point event of b that occurs after this timestamp cannot be next to a_1 in any possible world. We simply ignore such point events to ensure correct results and to save time. In our implementation, we revise the function $next()$ such that it stops the matching when the timestamp NELT is signaled. We prove the correctness of this method in Appendix B.2. The key to our proof is the *dichotomy* property of NELT: if event e matches pattern component E_{j+1}, any point of e that occurs before or at e_{m_j}'s NELT can be in a point match, and none of the points of e that occurs after e_{m_j}'s NELT can.

Step 3: Match Collapsing. The match collapsing module collects point matches as they are produced by the pattern matching module as the emit time, t_{emit}, advances (see Fig. 2). We first assign match, $m : (e_{m_1}^{t_1}, \ldots, e_{m_\ell}^{t_\ell})$, to a group corresponding to its signature, i.e., its sequence of unique ids. All point matches of this signature must have been reported by time $e_{m_\ell}.\text{upper} + W$, where W is the query window size. Thus when $t_{emit} > e_{m_\ell}.\text{upper} + W$, we have

all the point matches of this signature, and then collapse them into a succinct format defined in §2.3. Finding the tightest time range for all the point matches is straightforward. The remaining task is to compute the confidence.

Let S_α be the set of point matches of signature $\alpha = (e_{m_1}, \ldots, e_{m_\ell})$. For a *skip till any match* query, the confidence $C_{any}(\alpha)$ equals:

$$C_{any}(\alpha) = \sum_{m \in S_\alpha} \mathbb{P}\left[(e^{t_1}_{m_1}, \ldots, e^{t_\ell}_{m_\ell})\right]) = \sum_{m \in S_\alpha} \prod_{j=1}^{\ell} \mathbb{P}\left[e^{t_j}_{m_j}\right] \quad (1)$$

This calculation is correct because the probability of the point match $e^{t_1}_{m_1}, \ldots, e^{t_\ell}_{m_\ell}$ is the product of the probabilities of its individual events, and different point matches represent disjoint sets of possible worlds, hence independent of each other.

Calculating the confidence, $C_{next}(\alpha)$, of a *skip till next match* query is more subtle because some matches require that there are no intervening events of certain types. For example, for a^2_1, c^5_2, b^3_3 to be a match of the query in Fig. 1, we require that event c_4 does not occur at time 4. Thus we must multiply by the probability that no intervening event spoils a given match.

Let $\alpha = \{e_{m_1}, \ldots, e_{m_\ell}\}$ be a match signature for a skip till next match query and let $m = (e^{t_1}_{m_1}, \ldots, e^{t_\ell}_{m_\ell})$ be a potential point match. Then m is indeed a point match iff (1) $t_1 < \ldots < t_\ell$, and (2) for each $e^{t_j}_{m_j}$ ($2 \le j \le \ell$), no point event matching E_j occurs between $e^{t_{j-1}}_{m_{j-1}}$ and $e^{t_j}_{m_j}$. Let $\Theta_j(m)$ be the set of all such excluded point events. Thus condition (2) may be written $\Theta_j(m) = \emptyset$ for $2 \le j \le \ell$, or $\Theta(m) = \emptyset$ for short. Then the confidence of the skip-till-next match, $C_{next}(\alpha)$, equals:

$$C_{next}(\alpha) = \sum_{m \in S_\alpha} \mathbb{P}\left[(e^{t_1}_{m_1}, \ldots, e^{t_\ell}_{m_\ell})\right] = \sum_{m \in S_\alpha} \prod_{j=1}^{\ell} \mathbb{P}\left[e^{t_j}_{m_j}\right] \cdot \mathbb{P}\left[\Theta(m) = \emptyset\right] \quad (2)$$

We then consider two cases. In the first case, an event can match at most one pattern component, due to the exclusiveness of the event types and predicates using in the pattern components. Thus, any event can occur in at most one of the $\Theta_j(m)$ sets. So the probabilities of these sets being empty are independent of each other. Hence, we can rewrite Eq. 2 as:

$$C^1_{next}(\alpha) = \sum_{m \in S_\alpha} \frac{\prod_{j=2}^{\ell} \mathbb{P}\left[e^{t_{j-1}}_{m_{j-1}}\right] \cdot \mathbb{P}\left[e^{t_j}_{m_j}\right] \cdot \mathbb{P}\left[\Theta_j(m) = \emptyset\right]}{\prod_{j=2}^{\ell-1} \mathbb{P}\left[e^{t_j}_{m_j}\right]} \quad (3)$$

The equation above leads to a memoization-based algorithm to compute $C^1_{next}(\alpha)$. For all point matches in S_α, it computes the quantity $\mathbb{P}\left[e^{t_{j-1}}_{m_{j-1}}\right] \mathbb{P}\left[e^{t_j}_{m_j}\right] \mathbb{P}\left[\Theta_j(m) = \emptyset\right]$ once and records it for reuse for other point matches sharing this quantity. To efficiently compute $\mathbb{P}\left[\Theta_j(m) = \emptyset\right]$, we construct an index on the fly to remember those events that can potentially match a pattern component. $\mathbb{P}\left[\Theta_j(m) = \emptyset\right]$ is the product of the probability of each of these events occurring outside the range between $e^{t_{j-1}}_{m_{j-1}}$ and $e^{t_j}_{m_j}$. This algorithm is detailed in Algorithm 1 in Appendix B.1.

The second case is more complex in that an event can match more than one pattern component. The idea is that we can further enumerate the points of those events, $\{S_q\}$, that have matched multiple components. So conditioned on the specific points of events in $\{S_q\}$, we can factorize $\Theta_j(m) = \emptyset$ based on independence. So,

$$C^2_{next}(\alpha) = \sum_{e_{q_i} \in \{S_q\}} \prod_i \mathbb{P}\left[e^{t_i}_{q_i}\right] \cdot \sum_{m \in S_\alpha} \prod_{j=1}^{\ell} \mathbb{P}\left[e^{t_j}_{m_j}\right] \cdot \prod_{j=2}^{\ell} \mathbb{P}\left[\Theta(m) = \emptyset | \{e^{t_i}_{q_i}\}\right] \quad (4)$$

The corresponding algorithm also uses the event index to compute the quantity, $\mathbb{P}\left[\Theta(m) = \emptyset | \{e^{t_i}_{q_i}\}\right]$, as well as memoization.

4. AN EVENT-BASED FRAMEWORK

In this section, we present a second evaluation framework which is event based rather than point based. This way, we can eliminate the high cost of enumerating exponentially many possible point-based matches. It is not obvious how to efficiently find exactly all the possible matches in this way. Below, we describe two evaluation methods and several optimizations that together achieve this goal. It is worth noting that a few key algorithms developed in the point-based framework can be shared in the event-based framework.

4.1 The Query Order Evaluation Method

To simplify the discussion, we start with two temporary assumptions: Fix a pattern $p = (E_1, \ldots, E_\ell)$. An event can match only one of the ℓ components, and furthermore the events in a time window are presented to the matching module in query order. That is, the events matching the component E_1 are presented before those matching the component E_2, and so on. These assumptions will be later eliminated using a flexible evaluation algorithm.

Even with these simplifying assumptions, it is still quite subtle to find the event-based matches. To do so, we will conceptually walk through the events three times: first forward, revising the lower endpoints of each event interval as we form a potential match, second backwards, revising the upper endpoints of each event interval in the match, and third backwards again, checking that all the matched events can simultaneously fit within the query window, W.

Finding the Match Signature. We begin by introducing a boolean function ext such that $\text{ext}(m, e)$ is true iff event e may extend the partial match m of pattern p. To compute $\text{ext}(m, e)$, we inductively define the concept *valid lower bound* (vlb). We write $e \models E_j$ to mean that event e matches the pattern component E_j. In the base case, if $e \models E_1$, then $e.\text{vlb} = e.\text{lower}$. Inductively, assume that $m = (e_{m_1}, \ldots, e_{m_j})$ and $e_{m_j}.\text{vlb}$ is defined. If $e \models E_{j+1}$, define $e.\text{vlb} = \max(e_{m_j}.\text{vlb} + 1, e.\text{lower})$. Thus, $e.\text{vlb}$ is the first time that e might occur in match m of pattern p.

Using vlb we can immediately define ext:

$$\text{ext}(m, e) \quad \equiv \quad (|m| < \ell \;\&\; e \models E_{|m|+1} \;\&\; e.\text{vlb} \le e.\text{upper})$$

This completes the first pass in which we have computed the potential match $m = (e_{m_1}, \ldots, e_{m_\ell})$ and its valid lower bounds.

Example. Fig. 3(a) revisits our running example. We (temporarily) reorder events c_2 and b_3 so that they are presented in query order (A, B, C). We compute the valid lower bounds of events and evaluate the ext function as the same time. For example, $a_1.\text{vlb} = 1$, $\text{ext}(\emptyset, a_1) = \mathit{True}$; $b_3.\text{vlb} = 3$, $\text{ext}((a_1), b_3) = \mathit{True}$; and $c_2.\text{vlb} = 4$, $\text{ext}((a_1, b_3), c_2) = \mathit{True}$, yielding a match ($a_1$,$b_3$,$c_2$).

Second Pass. Now we walk back down the potential match, m, revising the upper bounds of each interval. We inductively define *revised upper bound* (rub) analogously to vlb: In the base case, $e_{m_\ell}.\text{rub} = e_{m_\ell}.\text{upper}$. Inductively, assuming $e_{m_{j+1}}.\text{rub}$ is defined, we let $e_{m_j}.\text{rub} = \min(e_{m_{j+1}}.\text{rub} - 1, e_{m_j}.\text{upper})$. As we compute the revised upper bounds, we check that each interval is nonempty.

Example. Fig. 3(b) shows the computation of the revised upper bounds after the match (a_1,b_3,c_2) is recognized. That is, $c_2.\text{rub} = 5$, $b_3.\text{rub} = 4$, and $a_1.\text{rub} = 3$.

Third Time Lucky. Finally we can consider the query window size, W. Since, the last possible time for e_{m_1} is $e_{m_1}.\text{rub}$, the last possible time for e_{m_ℓ} is at most $T_m = e_{m_1}.\text{rub} + W - 1$. If $e_{m_\ell}.\text{rub} \le T_m$, then the revised upper bounds are in fact the valid upper bounds, and we have validated the match m. Otherwise, we must walk back down the third time computing the *valid upper bounds* (vub) as follows: $e_{m_\ell}.\text{vub} = T_m$. Inductively, assuming $e_{m_{j+1}}.\text{vub}$ is defined, we let $e_{m_j}.\text{vub} = \min(e_{m_{j+1}}.\text{vub} - 1, e_{m_j}.\text{rub})$. At any timme during the third pass, if for some event e_{m_j}, we have

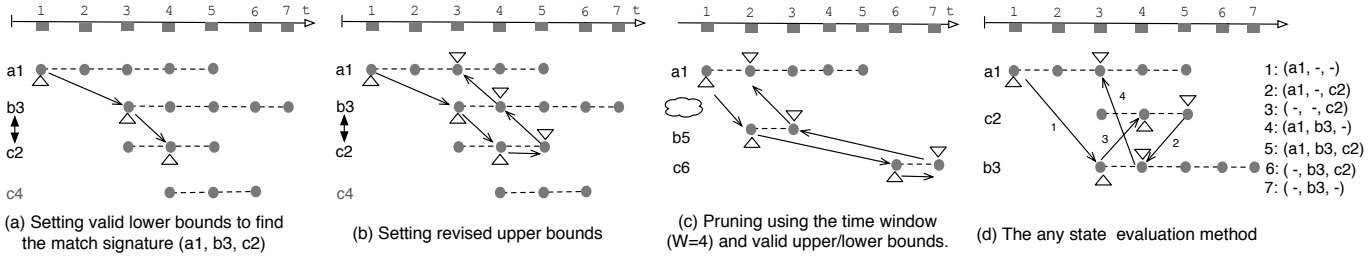


Figure 3: Illustration of the event-based evaluation (assuming that events are presented in query order).

e_{m_j}.vub $<$ e_{m_j}.vlb, then the match fails.

Example. Fig. 3(c) shows an example using three events a_1, b_5, and c_6. In the first pass, we compute the valid lower bounds as: a_1.vlb $= 1$, b_5.vlb $= 2$, and c_6.vlb $= 6$. After the second pass, we have: c_6.rub $= 7$, b_5.rub $= 3$, and a_1.rub $= 2$. Then we have $T_m = a_1$.rub $+ W - 1 = 5$. Since c_6.rub $= 7 > T_m = 5$, we start the third pass, in which we set c_6.vub $= T_m$ and can immediately see that c_6.vub $= 5 < c_6$.vlb $= 6$. So the match is pruned.

Given the extended ext() function and events presented in query order, we are now ready to evaluate patterns directly on the event stream, using a method which we call a *query order evaluation method*: If a query uses the *skip till any match* strategy, we run the pattern engine using ext() and skip till any match. If a query uses *skip till next match* instead, we further augment the pattern engine using the NELT (Next Event's Latest Time) based mechanism as described in §3. For efficiency, our implementation actually uses a one-pass, incremental algorithm as the ext function runs forwards on the event stream, which is detailed in §C.1. We prove the correctness of our query order evaluation method in §C.5.

Computing the Confidence. We last compute the confidence of a match. Recall that in the point-based evaluation framework, if skip till any match is used in a query, we can simply sum up the probabilities of the point matches sharing the signature. In the event-based framework, however, we are only given the events in the match. Hence, we need to enumerate all valid point matches and sum up their probabilities. This procedure is detailed in Algorithm 2 in Appendix C.4. We note here that this procedure employs memoization to avoid repeated computation and can be performed incrementally as events are selected. If skip till next match is used in a query, we can reuse the confidence algorithm in the point-based framework with a small change to first enumerate the points in the valid lower and upper bounds of the events and create point matches.

4.2 The "Any State" Evaluation Method

We next relax the assumption that events are presented in query order. Instead, we consider events in their arrival order. Fig. 3(d) shows the events a_1, c_2, and b_3 in their arrival order. If we run the above algorithm, ext$(\emptyset, a_1)$ will select a_1, ext$((a_1), c_2)$ will skip c_2, and ext$((a_1), b_3)$ will select b_3. However, we have permanently missed the chance to extend (a_1, b_3) with c_2. To address the issue, we extend the pattern evaluation method so that it can begin from any pattern component and attempt to select any event that can potentially match others pattern components until the match completes or fails—we call this new method an "any state" evaluation method. To make our terminology clear, we reserve the term "partial match" to mean a match of a prefix of the pattern, and refer to the partial processing result using this flexible evaluation method as a "run".

The main idea is the following: A new run is started if the current event can match any of the pattern components, say E_i. When the next event comes, if it can match any other pattern component E_j and further satisfy the ordering constraints with the events already selected by the run, then the current run is cloned: in one instance, the new event is selected into the run; in the other instance, it is ignored to preserve the previous run so that it can be matched differently in the future. The details and optimizations of the method are given in §C.2 and the correctness proof in §C.5.

Example. Fig. 3(d) shows the any state evaluation method for the three events a_1, c_2, and b_3. It lists the runs created as these events arrive: a_1 causes the creation of the run denoted by $(a_1, -, -)$. Then c_2 causes two new runs, $(a_1, -, c_2)$ and $(-, -, c_2)$, to be created. The arrival of b_3 clones all three existing runs, then extends them with b_3, and add a new run $(-, b_3, -)$.

4.3 Optimizations

We next present two optimizations to improve the performance of the event-based evaluation framework.

Sorting for Query Order Evaluation. We observe that there is a significant difference in complexity between the query order evaluation method, which assumes events to be presented in query order, and the any state evaluation method, which evaluates events in arrival order. If we can sort the input stream to present events in query order for pattern evaluation, we might achieve an overall reduced cost. Sorting based on query order is not always possible, especially when an event can match multiple components of a pattern. However, for many common queries, an event can match at most one pattern component, due to the exclusiveness of the event types and predicates used. In this case, we sort events such that if two events match two different components, E_i and E_j $(i < j)$, and overlap in time, the one matching E_i will be output before the other matching E_j, despite their arrival order. To do so, we use buffering and exploit ordering information given by the arrival order or heartbeats [19, 13, 21]. See the appendix (§C.3) for details.

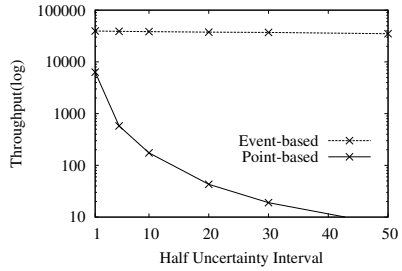
Selectivity Order Evaluation. The any state evaluation method can be applied to events ordered by any criterion, besides the arrival order. Borrowing the idea from recent work [15], our second optimization creates a buffer for each component E_j and triggers pattern evaluation when all buffers become non-empty. At this time, we output events from the buffers in order of the selectivity of E_j; that is, we output events first for the highly selective components and then for less selective components. This way, we can reduce the number of runs created in the any state evaluation method.

5. PERFORMANCE EVALUATION

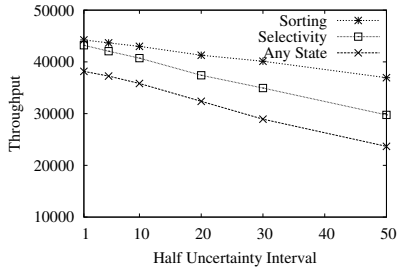
We have implemented both evaluation frameworks using the SASE pattern query engine [1]. In this section, we evaluate these frameworks with the relevant optimizations using both synthetic event streams with controlled properties and real traces collected from MapReduce-based cluster monitoring.

5.1 Evaluation using Synthetic Streams

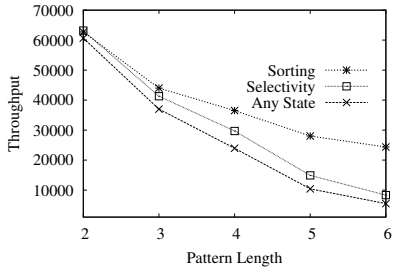
We implemented an event generator that creates a stream of events of a single attribute. The events form a series of increasing values from 1 to 1000 and once reaching 1000, wrap around to start a new series. Events arrive in increasing order of time t but each event has an uncertainty interval $[t - \delta, t + \delta]$; we call δ the half uncertainty



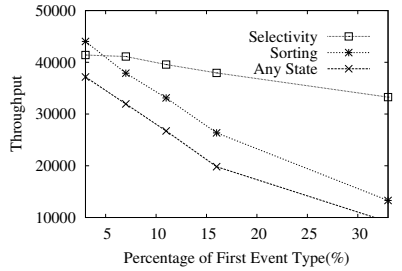
(a) Varying uncertainty interval (skip-till-any)



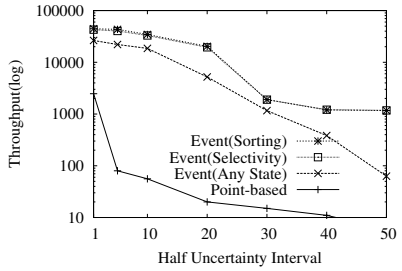
(b) Event-based optimizations with varied uncertainty intervals (skip-till-any)



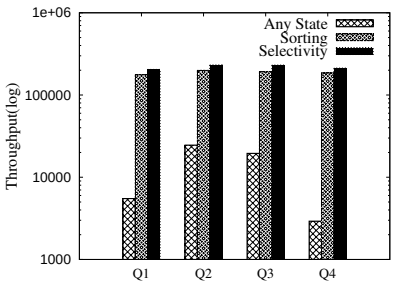
(c) Event-based optimizations with varied pattern lengths (skip-till-any)



(d) Event-based optimizations with varied event selectivities (skip-till-any)



(e) Varying uncertainty interval (skip-till-next)



(f) Case study of cluster monitoring

Figure 4: Performance results using synthetic event streams and real traces in cluster monitoring.

interval size. Each stream contains 0.1 to 1 million events to ensure stable performance. Queries follow the following pattern:

SEQ($E_1, \ldots, E_\ell$) WHERE $E_1\%v_1 = 0, \ldots, E_\ell\%v_\ell = 0$ WITHIN W

Our workloads are controlled by a set of parameters: the half uncertainty interval size δ (default 5), the time window size W (default 100 units), the pattern length ℓ (default 3), the event selection strategy (skill to any match or skip till any match), and the selectivity of each pattern component controlled by the value v_j ($1 \leq j \leq \ell$).

Point vs. Event based Evaluation (skip till any match). We begin by comparing the point-based and event-based evaluation methods (without optimizations) for skip till any match queries. We first increase the half uncertainty interval size δ from 1 to 50. Fig. 4(a) shows that the point-based method degrades its performance fast because as δ increases, the number of point events also increases. More points lead to more runs, in the worst case $O(\delta^\ell)$, hence a excessive cost. The event-based method is not sensitive to δ as it does not enumerate points for pattern evaluation and hence has a constant number of runs. Although to compute confidence it does enumerate points in the valid intervals, this cost is relatively small. Similar results for W and ℓ are shown in the appendix.

Optimizations of the Event based Method (skip till any match). We next evaluate the two optimizations, sorting for query order evaluation and selectivity order evaluation, for enhancing the basic event-based evaluation method, called the any state method.

Fig. 4(b) shows the results with varied δ. The performance of the any state method degrades linearly with δ. This is because as δ increases, there will be more matches to produce since events overlap more. Moreover, each run needs to wait longer before it can be pruned. Sorting for query order evaluation performs the best, because pattern evaluation proceeds from E_1 to E_ℓ, avoiding the overhead of starting a run from any state. This can reduce the number of runs significantly. The selectivity-based method lies between the above two. It buffers events separately for every pattern component. Before all buffers receive events, it can remove some out-of-date events and hence reduce the number of runs started.

Fig. 4(c) compares these methods as the pattern length ℓ is increased. The any state method loses performance quickly. Since a run can start by matching any pattern component in this method, a longer pattern means a higher chance for an event to match a component and start a run. Sorting still works the best, alleviating the performance penalty of the any state method. The selectivity method degrades similarly to the any state method as it suffers from a similar problem of starting more runs from the additional components.

We then examine the effect of event frequencies. We keep the query selectivity roughly the same, increase the percentage of events matching the first pattern component E_1 by adjusting its predicate, and decrease that for the last pattern component E_ℓ accordingly. As a result, more events can match E_1 and fewer can match E_ℓ. Fig. 4(d) shows the results. In this case, sorting creates more runs because it starts from E_1, and is only slightly better than the any state method. The selectivity method works the best, because it can remove out-of-date events from the buffer of E_1 before it sees events in other buffers, especially that for E_l, hence avoiding many runs.

Point vs. Event based Evaluation (skip till next match). We next consider queries using skip till next match. Without temporal uncertainty, this strategy is more efficient than skip till any match in the number of query matches produced because it only aims to find the "first" match of each pattern component. Under temporal uncertainty, however, it is in fact harder to find such first matches. We next show the performance of the point based method and the event based methods including the optimizations. Fig. 4(e) shows the results as δ is varied. Compared to Fig. 4(a), the point-based method experiences an earlier drop in performance due to the combined costs of numerous point events and the more complex confidence computation. The event-based methods also reduce performance to 1000-2000 events/sec. As δ goes up, more matches are produced and for each match, the confidence computation enumerates the points in the events' valid intervals. The cost of confidence computation becomes dominant when $\delta \geq 30$.

5.2 Evaluation in Cluster Monitoring

To evaluate our techniques in real-world settings, we performed a

case study of Hadoop clustering monitoring: In a 11-node research cluster, we ran a Hadoop job for inverted index construction on 457GB of web pages. This job used around 6800 map tasks and 40 reduce tasks on 10 compute nodes and ran for 150 minutes. The Hadoop system logs events for the start and end times (in us) of all map and reduce tasks as well as common operations such as the pulling and merging of data in the reducers. For this job, the Hadoop log contains 7 million events. In addition, this cluster uses the Ganglia monitoring tool [10] to measure the max and average load on each compute node, once every 15 seconds. By consulting a research group on cluster computing, we constructed four useful pattern queries, similar to Query 1 in §2, to study the effects of different Hadoop operations on the load on each compute node.

Most notably, these monitoring queries require the use of uncertainty intervals. The first reason is the granularity mismatch between Hadoop events (in microsecond) and Ganglia events (once 15 seconds). The second reason is that the the start and end timestamps in the Hadoop log were based on the clock on the job tracker node, not the actual compute nodes that ran these tasks and produced the Ganglia measurements. Thus, there is a further clock synchronization issue. So, we rounded all Hadoop timestamps using 0.1 second as a time unit, set $\delta_H = 0.5$ second for Hadoop events, and $\delta_G = 7.5$ seconds for Ganglia events. We ran our event based methods on the merged trace of the Hadoop log and the Ganglia event stream.

Fig. 4(f) shows the throughput results. Again, the sorting and selectivity methods significantly improve over the any state method. In this study, the selectivity method outperforms sorting due to the disparity of event frequencies: The Hadoop events are voluminous whereas the Ganglia events are less frequent, among which the high load events are even less common. The selectivity method can exploit this fact to reduce the number of runs in pattern evaluation. Nevertheless, both optimizations achieve high throughout, much higher than the average arrival rate of 778 events per second.

6. RELATED WORK

We have discussed most relevant work on complex event processing in earlier sections. Below, we survey a few broader areas.

Interval-based event processing. Several event processing systems [3, 2, 4, 7, 22] model events using a time interval, representing the event validity in the interval. Our problem is different because the events we consider are instantaneous but the occurrence times are unknown or imprecise. Our intervals are used to bound the occurrence time and characterize likelihood of occurrence in the interval, leading to different query semantics and evaluation techniques.

Out of order event streams have been intensively studied in stream processing [3, 4, 13, 19]. Out-of-order arrival is detected when the arrival of events does not present an increasing order of application time, or event occurrence time in event processing [19]. Under the temporal uncertainty model, there is no clear notion of "increasing order of event occurrence time"; hence, the notion of out of order events is blurred. While we leverage existing work such as [13, 19] to assume an arrival order as described §2.2, a thorough study of the connection between events with imprecise timestamps and out of events is deferred to our future work.

Probabilistic event processing. Although our work uses the possible world semantics to formally define query results, it differs from probabilistic event processing [16] that addresses the uncertainty of the *values* in events but not the *timestamps*.

Temporal databases are surveyed in [6]. The most relevant work is supporting valid-time indeterminacy in temporal databases [9], which uses a similar temporal model as ours. However, our formal semantics differs in that it involves enumerating the possible worlds involving both the events that match a pattern and those that do not, which is not required in [9] but needed in our work to correctly support complex event selection strategies on streams, such as skip till next match. Furthermore, our work supports pattern queries over live streams, as opposed to stored data, hence the need to deal with arrival orders and employ incremental computation.

7. CONCLUSIONS

In this paper, we addressed pattern evaluation in event streams with imprecise timestamps. We presented a temporal uncertainty model for such events, formal semantics of pattern evaluation under this model, novel evaluation frameworks, and methods and optimizations in these frameworks. Our evaluation results show that the best of our methods achieves thousands to 10's of thousands of events per second both in a real-world application of cluster monitoring and in a wide range of query workloads on synthetic streams. In the future, we plan to extend our work to support advanced pattern features such as negation and Kleene closure, consider more efficient techniques when given a confidence threshold or requested to return only a ranked list of matches based on confidence, and further study out of order streams under the temporal uncertainty model.

8. REFERENCES

[1] J. Agrawal, Y. Diao, et al. Efficient pattern matching over event streams. In *SIGMOD*, 147–160, 2008.

[2] M. Akdere, U. Çetintemel, et al. Plan-based complex event detection across distributed sources. *PVLDB*, 1(1):66–77, 2008.

[3] M. H. Ali, C. Gerea, et al. Microsoft cep server and online behavioral targeting. *PVLDB*, 2(2):1558–1561, 2009.

[4] R. S. Barga, J. Goldstein, et al. Consistent streaming through time: A vision for event stream processing. In *CIDR*, 363–374, 2007.

[5] P. Barham, A. Donnelly, et al. Using Magpie for request extraction and workload modelling. In *OSDI*, 259–272, 2004.

[6] M. H. Bhlen and C. S. Jensen. Temporal data model and query language concepts. *Encyclopedia of Information Systems*, 2003.

[7] A. J. Demers, J. Gehrke, et al. Cayuga: A general purpose event monitoring system. In *CIDR*, 412–422, 2007.

[8] L. Ding, S. Chen,et al. Runtime semantic query optimization for event stream processing. In *ICDE*, 676–685, 2008.

[9] C. E. Dyreson and R. T. Snodgrass. Supporting valid-time indeterminacy. *ACM Trans. Database Syst.*, 23(1):1–57, 1998.

[10] Ganglia monitoring tool. http://ganglia.sourceforge.net/.

[11] E. Koskinen and J. Jannotti. Borderpatrol: isolating events for black-box tracing. In *EuroSys*, 191–203, 2008.

[12] A. Lachmann and M. Riedewald. Finding relevant patterns in bursty sequences. *PVLDB*, 1(1):78–89, 2008.

[13] M. Liu, M. Li, et al. Sequence pattern query processing over out-of-order event streams. In *ICDE*, 784–795, 2009.

[14] E. Lo, B. Kao, et al. OLAP on sequence data. In *SIGMOD Conference*, 649–660, 2008.

[15] Y. Mei and S. Madden. ZStream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD Conference*, 193–206, 2009.

[16] C. Ré, J. Letchner, et al. Event queries on correlated probabilistic streams. In *SIGMOD*, 715–728, 2008.

[17] S. Rizvi, S. R. Jeffery, et al. Events on the edge. In *SIGMOD*, 885–887, 2005.

[18] R. Sadri, C. Zaniolo, et al. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.*, 29(2):282–318, 2004.

[19] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, 263–274, 2004.

[20] T. Tran, C. Sutton, et al. Probabilistic inference over RFID streams in mobile environments. In *ICDE*, 2009.

[21] P. A. Tucker, D. Maier, et al. Using punctuation schemes to characterize strategies for querying over data streams. *IEEE Trans. Knowl. Data Eng.*, 19(9):1227–1240, 2007.

[22] W. M. White, M. Riedewald, et al. What is "next" in event processing? In *PODS*, pages 263–272, 2007.

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

APPENDIX

A. NOTATIONAL CONVENTION

Table 1 summarizes the notation used in this paper.

Pattern(ℓ, W):	$(E_1, \ldots, E_\ell)$ of length ℓ and time window W
Event sequence S:	$e_1, \ldots, e_n$
Possible world j:	pw_j
Point match m:	$(e_{m_1}^{t_1}, \ldots, e_{m_\ell}^{t_\ell})$, $t_1 < ... < t_\ell$
Partial match m:	$(e_{m_1}^{t_1}, \ldots, e_{m_j}^{t_j})$
Query match Q_m:	given $\{ m : (e_{m_1}^{t_1}, \ldots, e_{m_\ell}^{t_\ell}) \}$, output: signature: $(e_{m_1}.\text{id}, \ldots, e_{m_\ell}.\text{id})$ range: $[min_m(e_{m_1}^{t_1}.lower), max_m(e_{m_\ell}^{t_\ell}.upper)]$ confidence: $\sum_{pw_j \rightarrow (e_{m_1}, \ldots, e_{m_\ell})} \mathbb{P}\left[pw_j\right]$

Table 1: Notation used in this paper

B. POINT-BASED EVALUATION

In this appendix, we provide the pseudocode for computing the confidence for skip till next match queries, prove the correctness of our point-based evaluation method for skip till any match and skip till next match queries, show additional implementation details, and offer extended discussions of this method.

B.1 Pseudocode

Algorithm 1 shows the computation of the match confidence for a *skip till next match* query in the point-based framework.

Algorithm 1 Compute the confidence for point-based framework.

Input: match m**:**$(e_{m_1}, e_{m_2}, ...e_{m_\ell})$**,** $\bar{S}_{m_2}, ..., \bar{S}_{m_\ell}$ ($\bar{S}_{m_i}$ **denotes the set of events that can potentially extend a partial match ending at** $e_{m_{i-1}}$**)**

if The query strategy is skip till any match **then**
  Set q as the confidence of m using Equation (1)
else if The query strategy is skip till next match **then**
  Pre-computation:
  for Point match $m_p \in m$ **do**
    for i=1 to i=ℓ **do**
      $e_{m_i}^{t_i}$ = point event of m_p at state i
      $e_{m_{i+1}}^{t_{i+1}}$ = point event of m_p at state $i+1$
      if $\mathbb{P}\left[e_{m_i}^{t_i} \prec e_{m_{i+1}}^{t_{i+1}}\right]$ is not computed yet **then**
        $\mathbb{P}\left[e_{m_i}^{t_i} \prec e_{m_{i+1}}^{t_{i+1}}\right] = \mathbb{P}\left[e_{m_i}^{t_i}\right] \times \mathbb{P}\left[e_{m_{i+1}}^{t_{i+1}}\right]$
        $\mathbb{P}\left[\text{None of } \bar{S}_{m_{i+1}} \text{ occurs between}[t_i, t_{i+1}]\right] = \prod_{e_{m_j} \in \bar{S}_{m_{i+1}}} \mathbb{P}\left[e_{m_j} \text{ not between } [t_i, t_{i+1}]\right]$
      end if
    end for
  end for
  $Confidence_m = 0$
  for Point match $m_p \in m$ **do**
    for $i = 1$ to $i = \ell$ **do**
      $e_{m_i}^{t_i}$ = point event of m_p at state i
    end for
    $Confidence_m$ += $\dfrac{\prod_{i=1}^{\ell-1} \mathbb{P}\left[e_{m_i}^{t_i} \prec e_{m_{i+1}}^{t_{i+1}}\right] \times \mathbb{P}\left[\text{None of } \bar{S}_{m_{i+1}} \text{ between } [t_i, t_{i+1}]\right]}{\prod_{i=2}^{\ell-1} \mathbb{P}\left[e_{m_i}^{t_i}\right]}$
  end for
end if

The above algorithm can be extended to support the more complex case in which one event can match multiple pattern components. In particular, when we index the events for S_{m_i}, we also keep track of the events that can match more than one pattern component. Once we find these events, then we could further enumerate these events when we compute the confidence as shown in Eq. (4).

B.2 Correctness Proof

Skip till any match. We first prove the correctness of our point-based evaluation framework when the skip till any match strategy is used in a query.

PROOF. If a query uses the skip till any match strategy, pattern matching in the point-based evaluation framework naturally runs skip till any match on the point event stream.

We first show that any point match returned by the skip till any match strategy exists in some possible world. This is because the point match already satisfies the ordering constraint as well as query-specified constraints such as predicates and the time window.

We next prove that any match that exists in some possible world would be returned by the skip till any match strategy on the point event stream. We prove this by contradiction. Assume that there is a match m with signature $(e_{m_1}^{i_1}, e_{m_2}^{i_2}, ...e_{m_\ell}^{i_\ell})$ in one possible world, but it is not returned by skip till any match on the point event stream. Since m is a match, the constituent point events are in order, i.e., $i_1 < i_2 < ... < i_\ell$, and satisfy query-specified constraints such as predicates and the time window. In the expanded stream, points are ordered by timestamps. So, we have $(e_{m_1}^{i_1} \prec e_{m_2}^{i_2} \prec ... \prec e_{m_\ell}^{i_\ell})$ in the point event stream. By definition, skip till any match will have one such run that first selects $e_{m_1}^{i_1}$, ignores other point events until $e_{m_2}^{i_2}$ arrives, selects $e_{m_2}^{i_2}$, ignores other point events until $e_{m_3}^{i_3}$ arrives, and so on, resulting in a match. This contradicts the assumption above. Hence our second statement is proved. □

Skip till next match. We next prove the correctness of the point-based evaluation framework when the skip till next match strategy is used in a query. Recall that for such queries, our evaluation framework uses the skip till any match strategy on the point event stream and the modified $next()$ function with the Next Event's Latest Time (NELT) as the termination criterion.

PROOF. Consider a pattern $(E_1, \ldots, E_\ell)$ and a partial match $(e_{m_1}^{t_1}, \ldots, e_{m_j}^{t_j})$ $(j \geq 1)$, with e_{m_j} being the last selected event. We prove the following statements are true:

(1) Any point event, denoted by e_i^t, that starts **after** e_{m_j}'s NELT cannot be used to extend the partial match $(e_{m_1}^{t_1}, \ldots, e_{m_j}^{t_j})$ in any possible world when that possible world runs under the skip till next match strategy. This is clear from the NELT definition: Among all events that can match the next pattern component E_{j+1} and start after e_{m_j} ends, the event that ends the earliest, denoted by e_k, sets the NELT of e_{m_j} using $e_k.upper$. Since e_i^t occurs after the e_{m_j}'s NELT, it will surely be preceded by the point event e_k^{NELT} in any possible world, and hence cannot be the next to e_{m_j}.

(2) Every point event, e_i^t, that can potentially match the pattern component E_{j+1} and starts **before or at** e_{m_j}'s NELT, can actually be used to extend the partial match $(e_{m_1}^{t_1}, \ldots, e_{m_j}^{t_j})$ in a possible world that runs under the skip till next match strategy. We construct one such possible world as follows: (i) the event e_{m_j} occurs at its last time point; (ii) all events that can potentially match E_{j+1} and overlap with e_{m_j}, excluding e_i^t, take a point before or at the same point as e_{m_j}, hence not meeting the ordering constraint; and (iii) all events that can potentially match E_{j+1} and occur after e_{m_j} ends but before e_{m_j}'s NELT, excluding e_i^t, take a point at or after NELT. This way, all other events that can potentially match E_{j+1} have made room for e_i^t to be the first match of the pattern component E_{j+1} (or one of the first that occur at the same time NELT).

Given the above statements, it is easy to see that our implementation uses the skip till next match strategy to achieve (2) while using the NELT to guarantee (1). □

B.3 NELT Implementation

In the implementation of NELT, we incrementally compute the NELT of an event e. Every time before e_i selects its next event e_{i+1}, it compares its current NELT with e_{i+1}.lower. If the e_{i+1}.lower > current NELT, then e_i would not select e_{i+1}. Otherwise, e_i would select it. Then it would compare its current NELT with e_{i+1}.upper. If current NELT is smaller, then the NELT would not change. If the current NELT is larger, then we would update e_i's NELT to e_{i+1}.upper. At the same time, we need to prune runs that have passed the previous NELT check, while would fail in the check using the new NELT. In order to efficiently prune these runs, we index them using e_i as the key. So when an event is selected by a run, it is not safe to say that it is possible to be the next event. For similar reasons, when we find a match using skip till next match, we do not output it immediately. While we buffer it until we are sure that there would be no NELT updates that can prune this match.

B.4 Extended Discussions

The point-based evaluation framework offers three key benefits: First, it has tremendous performance benefits over an evaluation method based on the formal semantics—the latter is infeasible in most workloads. More precisely, the point-based evaluation method dynamically finds the possible worlds in which the order of point events matches the query-specified order, and simply ignores other possible worlds. Second, it requires the minimum change of an existing pattern query engine, hence easy to use.

Third, when an application receives the query match (which is a collapsed format) but wants the detailed point matches, it can run the point-based evaluation method over the events in the query match, if the query uses the skip till any match strategy. This re-evaluation incurs little cost because it involves only a few events, as opposed to many more in the window on the input stream. However, if the query uses the skip till next match strategy, it is incorrect to rerun the point-based evaluation over only the events in the query match. Hence, the application cannot recover the point matches in this case.

This method, however, has its own set of drawbacks: The efficiency can still be limited because the point event stream is much larger than the original event stream, incurring significant costs in both stream expansion and pattern matching. It further causes the delay of returning matches due to the overhead of buffering point events and emitting them in increasing order of occurrence time.

C. EVENT-BASED EVALUATION

In this appendix, we provide additional details and optimizations of our evaluation methods, including the query order evaluation method, the any state evaluation method, and the sorting-based optimization. We also show the pseudocode for computing the confidence for skip till any macth queries. We finally present the correctness proofs of our evaluation methods.

C.1 An Incremental Method for Query Order Evaluation

While the query evaluation evaluation method in §4.1 is described using three passes over the events in a match, our implementation actually takes one pass over events as the ext function runs forwards on the input stream. Our one-pass algorithm employs incremental computation of valid lower and upper bounds of events and incremental evaluation of the window constraint. Using this algorithm, the valid lower and upper bounds may not be as tight as the true ones defined in the three-pass algorithm initially, but will converge to the true ones when the matching process completes and generates a match.

Consider a partial match $m = \emptyset$ or $(e_{m_1}, \ldots, e_{m_j})$, and the current event e in the input. The one pass algorithm takes three steps:

1. Compute e's valid lower bound given m. Initialize e's valid upper bound using its own upper bound.
2. If m is nonempty, update the valid upper bound of the events in m in reverse pattern order, i.e., from e_{m_j} down to e_{m_1}.
3. $\text{ext}(m, e) = \mathit{True}$ if (1) e.vlb $\leq$ $e.upper$ and (2) e.vlb $<$ e_{m_1}.vub $+ W$.

The second condition in the last step uses the window constraint to filter the current event, by comparing its valid lower bound with the valid upper bound of the first event in the partial match.

Example. Fig. 3(c) shows an example using three events a_1, b_5, and c_6. Upon arrival of c_6, we have a partial match (a_1, b_5). Step 1 above sets c_6.vlb $= 6$ and c_6.vub $= 7$. Step 2 sets b_5.vub $= 3$ and a_1.vub $= 2$. Then in Step 3, the window constraint $W = 4$ is expressed as c_6.vlb $< a_1$.vub $+ 4$. That is, the latest point in a_1's valid interval and the earliest point in c_6's valid interval must fit in the window; otherwise, c_6 cannot be included in a match starting with a_1. In this example, e_6 is pruned.

C.2 The Any State Evaluation Method

Details of the any state evaluation method. In our implementation, the any state evaluation method is an incremental algorithm that runs directly on the event stream. Given an event e, a run γ, and the set of events m selected in γ , our method proceeds as follows:

1. Type and Value Constraints: Check if e can match any new pattern component E_j based on the event type and predicates. If a predicate of E_j compares to other unmatched pattern components, defer it until it is instantiated later. If e matches E_j and can instantiate predicates between E_j and other matched components, evaluate those predicates to filter e.
2. Temporal Constraints: Let $E_i, \ldots, E_j, \ldots, E_k$ denote the contiguous matched pattern components involving E_j, $i \leq j \leq k$. Compute e's valid lower bound using $e_{m_{j-1}}$'s valid lower bound if existent, or e's lower bound otherwise. Compute e's valid upper bound using $e_{m_{j+1}}$'s valid upper bound if existent, or e's upper bound otherwise. Update the valid lower bound of the subsequent events $e_{m_{j+1}}, \ldots, e_{m_k}$ if present. Update the valid upper bound of the preceding events $e_{m_i}, \ldots, e_{m_{j-1}}$ if present. If these updates cause any of the events to have an empty valid interval, i.e., vlb $>$ vub, skip e. If e is retained, check the time window between the events matching the current two ends of the pattern to further filter e.
3. If e is retained, clone γ to γ' and select e to match E_j in γ'.

Example. Fig. 3(d) shows the any state evaluation method for the three events a_1, c_2, and b_3. It lists the runs created as these events arrive. Now consider the run (a_1, b_3, c_2). Fig. 3(d) also shows the computation of the valid intervals of these events. Before b_3 came, the valid intervals of a_1 and c_2 were simply set to their uncertainty intervals because they are not adjacent in the match. When b_3 arrives, four updates occur in order: (1) b_3.vlb = $max(a_1.\text{vlb} + 1, b_3.\text{lower}) = 3$; (2) b_3.vub = $min(c_2.\text{vub} - 1, b_3.\text{upper}) = 4$; (3) c_2.vlb = $max(b_3.\text{vlb} + 1, c_2.\text{lower}) = 4$; (4) a_1.vub = $min(b_3.\text{vub} - 1, a_1.\text{upper}) = 3$; These updates give the same result as in Fig. 3(b) assuming the events in query order.

Pruning runs. We observe that the any state evaluation method can create many runs. For efficiency, we prune nonviable runs using the window. Consider a run γ and the set of events m selected. At any point, we consider the smallest valid upper bound of the events in m. The run can be alive at most until $min_j(e_{m_j}.\text{vub}) + W$, called the time to live γ_{tll}. As more events are selected by γ, γ_{tll} will only decrease but not increase. Recall from §3 that our system has a notion $now = [max_{i=1}^{n}(e_i.lower), max_{i=1}^{n}(e_i.upper)]$ defined on all the events we have seen, and the maximum uncertainty interval

size U_{max}. Further, the arrival order constraint in our system implies that any unseen event must start after $now.lower - U_{max}$. So, a run γ can be safely pruned if $\gamma_{till} < now.lower - U_{max}$.

Another pruning opportunity arises when a run γ has part of the prefix unmatched; i.e., there is a pattern component E_j such as E_j is matched but E_{j-1} is not. We can prune γ based on the arrival order constraint between e_{m_j} and a future event matching E_{j-1}. Since any unseen event must start after $now.lower - U_{max}$, when e_{m_j}.upper $< now.lower - U_{max}$, we know that no future event can match E_{j-1}, and hence can safely prune γ.

C.3 Sorting for Query Order Evaluation

Recall that we proposed an optimization for sorting for query order evaluation. More precisely, we sort events such that if two events match two different pattern components, E_i and E_j ($i < j$), and overlap in time, the one matching E_i will be output before the other matching E_j, despite their arrival order. To do so, we create a buffer for each pattern component E_j except the first ($j > 1$). We buffer each event e matching E_j until a safe time to output it. Depending on the information available, the safe output time for e can be: (1) If we only have the arrival order constraint, then it is safe to output e if all unseen events are known to occur after $e.upper$, that is, e.upper $< now.lower - U_{max}$ (the earliest time of an unseen event given the arrival order constraint). (2) Many stream systems use heartbeats [19] or punctuations [13, 21] to indicate that all future events (or those of a particular type) will have a timestamp greater than τ. If we know that every event that can match a pattern component preceding E_j will have a start time after $e.upper$, then it is safe to output e.

C.4 Pseudocode

Algorithm 2 shows the computation of the match confidence for a *skip till any match* query in the event-based framework.

Algorithm 2 Compute the confidence for skip till any match in the event-based framework.

Input: Run r with a partial match $(e_{m_1}, ..., e_{m_j})$, $l \geq 1$

```
1: if l = 1 then
2:    for each point point_1 ∈ e_{m_1}'s valid interval do
3:       Record (point_1) as a partial point match ending at point_1
4:    end for
5:    Confidence ← 1
6: else
7:    Confidence ← 0
8:    for each point point_j ∈ e_{m_j}s valid interval do
9:       for each point point_{j-1} ∈ e_{m_{j-1}}s valid interval do
10:         if point_{j-1} occurs before point_j then
11:            for each point match path_{j-1} ending at point_{j-1} do
12:               path_j = (path_{j-1}, point_j)
13:               ℙ[path_j] = ℙ[path_{j-1}] × ℙ[point_j]
14:               Record path_j as a partial match ending at point_j
15:               Confidence += ℙ[path_j]
16:            end for
17:         end if
18:      end for
19:   end for
20: end if
```

C.5 Correctness Proofs

In order to show the correctness of our event-based framework, we will show that it can get the same results as the point-based framework.

Finding the Match Signature. We first show that the event-based framework can find the same match signature as the point-based framework.

PROOF. First we show that for any match signature found by the point-based framework, the event-based framework can also find it. When the pattern length is two, a point-match $(e_1^{t_1}, e_2^{t_2})$ tells us that $t_1 < t_2$. In the event-based framework, the boolean function ext$(e_1, e2)$ would be true because e_2.vlb $\leq t_2 \leq e_2$.upper. So the event-based framework can also find this match signature. It is trivial to extend the proof to any pattern length ℓ by induction.

Then we show that for any match signature found by the event-based framework, the point-based framework would find one or more point matches with the same signature. We can pick a point from each interval to compose the match signature. We can prove this by showing that we can simply pick the point at the valid lower bound of each event. Because e_i.vlb $< e_{i+1}$.vlb, so we can use these points to compose a point match with the same signature. Hence the correctness of finding the match signature is proved. □

Time Window Constraint. Since we have proved that both frameworks can capture the same match signatures without considering the time window constraint, we need to show that the event-based framework can support the time window correctly.

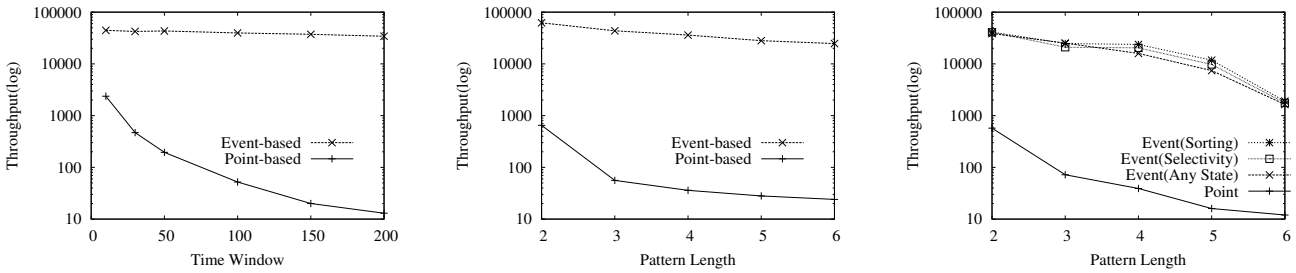
PROOF. First, we need to show that any match m satisfying the time window W, which is returned by the event-based evaluation framework, would have at least one point match using the same signature. We can prove this by showing that we can pick the valid lower bound of each event e from the match, because e.vlb $< e_{m_1} + W$. So this point match satisfies the time window constraint.

Then we need to prove that there is no point match using the same signature as any match pruned by the time window in the event-based framework. We can show this by contradiction. We assume that a match m is pruned by the time window in the event-based framework, but there is a point match m_p using the same signature which satisfies the time window constraint. Since we have this point match, according to the previous proof, we could say that every point is during the valid range of its corresponding event in m. Since the event match m violates the time window in the event-based framework, so at least e_ℓ.vlb $\geq e_{m_1}$.vub $+ W$. So we cannot pick a point from e_ℓ to compose the point match, which contradicts with the assumption. Hence the correctness of the time window constraint is proved. □

Skip till next match. The previous proofs show that the event-based framework can get the same results using the skip till any match selection strategy. When we use the skip till next match selection strategy, we can also apply the NELT check, such that we can filter events that cannot make a match for this selection strategy. In this way, the results of the event-based framework would stay the same as the point-based framework.

Flexible Evaluation from Any State: We need to prove that this method can capture the same set of matches as the query order evaluation method does (but it directly operates on the events in arrival order). We do not consider the case where the events arrive in query order, because the any state method would have the same process as the query order evaluation method.

PROOF. We need to show that by using the any state method, we still can capture the order of two events correctly in pattern matching (i.e., in query). Our ext function decides the order of two events according to the query order and their uncertainty intervals, so the arrival order or the order in which a run selects events in the any state method would not affect the results. So we can say that the any state method can capture the correct order between any two events. So when the any state method decides whether or not it selects an event into a run, it would not make any mistake. For skip till next match, we can get the same NELT as in the query order evaluation method.



(a) Varying time window (skip-till-any) (b) Varying pattern length (skip-till-any) (c) Varying sequence length (skip-till-next)

Figure 5: Additional performance results.

NELT is decided by the events that can potentially match the next pattern component. Obviously, these events would be the same in the query order evaluation method or in the any state method. So the any state method can run skip till next match strategy correctly. □

D. COMPLEXITY ANALYSIS

Our analysis below uses the following symbols: (*i*) U: the number of instants in an event's uncertainty interval. (*ii*) W: the size of the time window used in the query. (*iii*) ℓ: the number of pattern components of query. (*iv*) R_i: the arrival rate for events of type i. (*v*) N: the number of events in a time window.

Our analysis aims at a reasonable bound of the worst case performance. The exact performance characteristics of the point-based framework are presented in the evaluation section. In this regard, we make several assumptions to simplify the analysis: We consider different event types for different components in the query. We assume a uniform uncertainty interval size for all events. Furthermore, we assume that events of different types have the same arrival rate ($R_1 = R_2 = R_3 = \ldots = R$), and expect to see roughly the same number of events of each distinct type in a sufficiently large window.

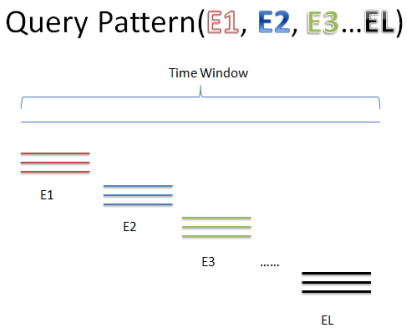


Figure 6: Event sequences that cause worst-case performance.

Complexity for skip till any match. We identify the worst case performance as the pattern evaluation involves the largest number of partial matches, also called runs, for a given sequence of events that could fit in a time window. The worst case requires the arrangement of the relevant events in a certain order. In particular, pattern evaluation incurs the largest number of runs if events are arranged as shown in Fig. 6. In this arrangement, the events of the first type (i.e., matching the first pattern component) appear first, before all events of other types. The events of the second type immediately follow the events of the first type. Then the events of the third type immediately follow, and so on. The events of different event types will not overlap at all.

For the point-based framework, $\#Runs = (RWU)^\ell$.

For the event-based framework, $\#Runs = (RW)^\ell$.

For the possible-world semantics, $\#Runs = N^\ell$, which is pattern-agnostic (note that in general $RW \ll N$). If we consider only the feasible runs in those possible worlds, i.e., satisfying the ordering constraint, then $\#Runs = U^{RW}$. But strictly based on the formal semantics, one has to pay the N^ℓ cost to recognize the U^{RW} runs.

Complexity for skip till next match. The worst case of skip till next match is the same with skip till any match. In this case, the NELT would not invalidate any event because every event satisfying the predicate starts before the NELT.

E. ADDITIONAL PERFORMANCE RESULTS

In this appendix, we provide more details of experimental setup and additional evaluation results.

All of our experiments were obtained on a server with an Intel Xeon 3GHz CPU and 8GB memory and running Java HotSpot 64-bit server VM 1.6 with the maximum heap size set to 3GB.

Point vs. Event based Evaluation (skip till any match). To study of the effect of the window size W, we increase it from 10 to 200 units. Fig. 5(a) shows the results about the point-based and event-based frameworks, demonstrating the performance benefits of the latter. We then consider the effect of the pattern length ℓ. As we vary ℓ from 2 to 6, we also adjust predicate selectivities so that longer patterns still obtain matches. Fig. 5(b) shows that while both methods are sensitive to ℓ, the point-based method suffers much severe performance penalty.

Point vs. Event based Evaluation (skip till next match). We consider queries using skip till next match and report the performance of the point based method and the event based methods including the optimizations. Fig. 5(c) shows the results as ℓ increases: Event-based methods work better than the point-based method. However, with increased ℓ, the cost of confidence computation also increases fast, hence the performance loss.