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, 8, 9, 15, 17, 21, 22] 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.

*This work has been supported in part by NSF grants IIS-0746939, CCF-0541018, and CCF-0830174, and a research gift from Cisco.

Proceedings of the VLDB Endowment, Vol. 3, No. 1

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, 9, 14, 17, 22] or a strict partial order [2, 3, 4, 8, 15, 21] 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 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, 19]. 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 [11], measures the max and average load on each node once every 15 seconds, whereas the system logs the jobs submitted to each node using the UNIX time (whose unit is a microsecond). To identify the jobs that max out a compute node, one has to deal with the uncertainty that the peak load reported by Ganglia can occur anywhere in a 15-second period, making it hard to judge whether it occurred before or after the submission of a specific job. That is, the temporal relationship between a load measurement event and a job submission event is not determined and cannot be modeled as a partial ordering (which we shall 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, 12], which involves numerous servers and system modules. As concurrent requests are served by various servers and modules, an event logging infrastructure generates event streams to capture all system activities, including thread resource consumption, packet transmission, and transfer of control between modules. The challenge is to demultiplex the event streams and account resource consumption by individual requests. The clock synchronization problem, however, makes it hard to merge the events relevant to a request from different machines into a single stream with a total or partial order [12].

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 components: matching a pattern in a set of possible worlds with deterministic timestamps, and collapsing matches into a succinct result format. This formal semantics offers a foundation for reasoning about the correctness of implementations.

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-based, requires minimum change of an existing pattern query engine and hence is easy to use. The second 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 incur higher cost under temporal uncertainty. (*iii*) Optimizations of the event-based framework are highly effective and offer thousands to tens 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. RELATED WORK

Interval-based event processing. Several event processing systems [3, 2, 4, 8, 21] model events using a time interval, representing the duration of the events. However, these systems deal with events with precise timestamps and often impose a strict partial order on the events. In contrast, our work deals with events that occur at a time instant but with uncertain timestamps. When a strict partial order is applied to events with uncertain timestamps, it will not allow us to enumerate all possible orderings of events and cause the loss of results that would exist in some of the possible worlds.

Out of order event streams. Existing work on out-of-order streams [3, 4, 13, 18] deals with events with precise timestamps, so the order between late events and in-order events is clear. Our work deals with imprecise timestamps and requires enumerating all possible orderings among events, which is a complex problem even without out-of-order events. In our context, out-of-order events can be handled using buffering and punctuation as in existing work.

Temporal databases are surveyed in [6]. The most relevant work is supporting valid-time indeterminacy [10], whose indeterminate semantics shares the basic idea as our semantics. However, the work in [10] only supports a single "select-from-where" block, while our work supports more complex event patterns that need to be expressed using nested queries in SQL (i.e., skip-till-next-match queries defined in the next section). Even for the simple patterns supported in [10], the proposed technique uses multiway joins, which is less efficient than either of the two evaluation frameworks we propose in this paper. Finally, our work supports pattern queries over live streams, as opposed to stored data, and hence also deals with arrival orders and incremental computation. The reader is referred to our technical report [23] for a more detailed discussion.

Probabilistic Databases. Our work also differs from probabilistic databases and stream systems, such as [7, 16], which address the uncertainty of the *values* in events but not the *timestamps*. If we were given n specific events in a window, it would be possible to cast our problem as a probabilistic database problem: treat the uncertain timestamp as an uncertain attribute, evaluate the pattern using non-equijoins on the timestamp, and then compute the join result distributions. However, when events carry imprecise timestamps and arrive in no particular order, defining the events in a time window is hard because event timestamps are uncertain, and defining a count window based on the arbitrary arrival order is not meaningful for pattern matching. Moreover, how to share computation across windows is another issue that probabilistic databases do not address.

3. 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.

3.1 Background on Pattern Queries

We begin by providing background on pattern queries [1, 4, 8, 17, 22] 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 temporal order and are further correlated based on their attribute values. Below, we highlight the key features of pattern queries using the SASE language which has been commonly used in recent work [1, 13, 15, 22]. 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 temporal order. The WHERE clause further specifies 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.

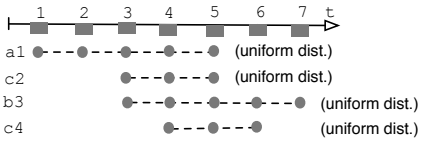
```
Query 1:
PATTERN SEQ(TaskStart a,CPU b,TaskFinish c,CPU d)
WHERE   a.taskId = c.taskId AND
        b.nodeId = a.nodeId AND
        d.nodeId = a.nodeId AND
        b.value > 95% AND
        d.value <= 70% AND
        skip_till_any_match(a, b, c, d)
WITHIN  15 seconds
RETURN  a, b, c
```

The event selection strategy addresses how to select the events relevant to a pattern query. For now, assume that events arrive in order of the occurrence time in the input stream (this assumption will be relaxed shortly). The events relevant to the pattern, however, are not necessarily in contiguous positions in the input stream. In this work, we consider two common event selection strategies (while referring the reader to [1] for all possible strategies). (*i*) **Skip till next match** [1, 8] specifies that in the pattern matching process, irrelevant events are skipped until an event matching the next pattern component is encountered. If multiple events in the stream can match the next

(a) A pattern query

```
PATTERN SEQ(A,B,C)  WITHIN 4 seconds
```

(b) A stream of four events



(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: 5/225 )
```

```
(Signature:(a1,b3,c4)
 Time Range: [1,6]
 Confidence: 4/225  )
```

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

pattern component, only the *first* of them is considered. (*ii*) **Skip till any match** [1, 15, 22] relaxes the previous one by considering *all* events in the stream that can match a pattern component. 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. In comparison, skip till next match will return only the former result because b_1 is the first event matching the b component of the pattern (and b_2 is simply ignored).

3.2 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 Section 1. To address this issue, our model allows the event provider to specify an uncertainty interval, $\mathbb{U}$: [lower, upper] $\subseteq T$, to bound the occurrence time of an event, with an optional probability mass function $f : \mathbb{U} \rightarrow [0,1]$ to characterize the likelihood of occurrence in the uncertainty interval (by default, a uniform distribution is used). The appropriate distribution of event occurrence time can be derived for each uncertainty source as in temporal databases [10]: for instance, the uniform distribution is often used to cope with granularity mismatch and clock specific distributions are used to model imprecise measurements.

In summary, an event in our model has the following format: ($event\_type$, $event\_id$, $\mathbb{U}$: [lower, upper], ($f : \mathbb{U} \rightarrow [0,1]$)?, $attributes$), where $event\_type$ specifies the attributes allowed in the events of this type and $event\_id$ is the unique event identifier. For example, a_1=(A, 1, [5, 9], (v_1, v_2, v_3)) represents an event of type A, id 1, an uncertainty interval from time 5 to time 9, and three required attributes. If the occurrence time of an event is certain, we 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 strict partial order on an arbitrary event stream. Consider a strict partial order, defined to be a binary relation on a sequence $\mathbb{S}$ that is (1) irreflexive, $\forall e \in \mathbb{S}, \neg(e \prec e)$; (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 the occurrence time [18]. In our problem, there is no clear notion of "increasing order of the occurrence time" due to imprecise timestamps. So we loosely define out-of-order arrival to be that e_1 is seen before e_2 in the stream but the earliest possible 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, 18], we can feed events into the query engine such that if e_1 is seen before e_2, then with respect to the occurrence time, e_1 either completely precedes e_2 or overlaps with e_2 in some way, i.e., e_1.lower $\leq e_2$.upper.

3.3 Formal Semantics under the Model

We next introduce the formal semantics of pattern query evaluation under our temporal uncertainty model, which has two parts:

Pattern Matching in Possible Worlds. In our model, an event has several possibilities of its occurrence time, i.e., at consecutive time points $\{(t_j, f(t_j))|j = 1, \ldots, U\}$, where $U = |\mathbb{U}|$. Given a sequence of events $\mathbb{S} = \{e_1, \ldots, e_i, \ldots e_n\}$, a unique combination of the possible occurrence time of each event, $(t_{ij}, f(t_{ij}))$, gives rise to a sequence S_k in which events have deterministic occurrence times and can be sorted by their occurrence times. Borrowing the familiar concept from the literature of probabilistic databases, we refer to S_k as a *possible world* for pattern evaluation, and compute its probability as $\mathbb{P}[S_k] = \prod_{i=1}^{n} f(t_{ij})$. 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 some of 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 potentially fit in a time window can be large. If the events have an average uncertainty interval size U, then the number of possible worlds is $O(U^n)$.

Match Collapsing. 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 matches* at a particular time.

Example: In Fig. 1, the matches from the 225 possible worlds 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 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 $\frac{15}{225}$. The second signature is (a_1, b_3, c_4) with its time range and confidence computed similarly. The final query matches at t=7 are shown in Fig. 1(d).

4. A POINT-BASED 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 that may fit 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 principle is to require minimum change of a pattern engine so that the proposed framework can work easily with any existing engine. Below, we discuss three main issues in the design of this framework (while giving the complete pseudocode in Appendix A.1).

Stream Expansion. The first issue is that existing pattern query engines [1, 8, 15] require that events be arranged in total or partial order based on their occurrence times. As stated in §3.2, 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?

To address the above question, we design a stream expansion algorithm that guarantees that the point events are produced in increasing order of time. Consider the example stream in Fig. 1(b). To generate a 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 ordered by the occurrence time of point events.

Our implementation is more complex than the conceptual procedure above due to the various event arrival orders. Recall from §3.2 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, i.e., e_1.lower $\leq$ e_2.upper. Our implementation uses buffering (of limited size) to cope with various arrival orders while emitting point events in order of occurrence time. Let e_1, ..., e_{n-1}, e_n be the events in arrival order. When receiving e_n, we create point events for all the instants in e_n's uncertainty interval and add them to the buffer (possibly containing other point events). Further, let **now** be a time range [lower $= max_{i=1}^{n}(e_i.\text{lower})$, upper $= max_{i=1}^{n}(e_i.\text{upper})$]. Also assume that the maximum uncertainty interval size for the event stream is U_{max} (which can be requested from event providers). Then we know that any unseen event must start after now.lower $- U_{max}$; otherwise, the unseen event will violate the the arrival order constraint with the earlier event e_i that sets now.lower $= e_i$.lower . So we can safely output the buffered point events up to now.lower $- U_{max}$, labeled as the **emit time** t_{emit}.

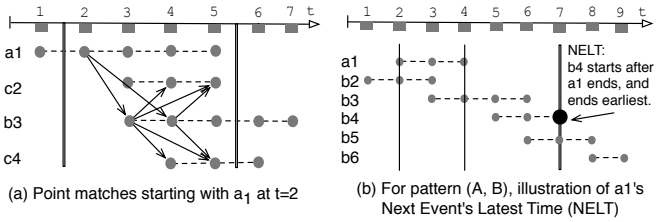


Figure 2: The point-based evaluation framework.

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 that directly running an existing engine 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*. How do we configure an existing engine and what is the minimum change needed to produce such matches?

Configuration. We first show that the pattern query engine must be configured with the most flexible event selection strategy, *skip till any match*, to produce a complete set of matches (no matter what strategy is actually used in the query).

Fig. 2(a) 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. The formal semantics requires enumerating all possible worlds that involve a_1^2 (45 of them) to find those matches.

We show that the skip till any match strategy offers a more efficient algorithm that directly searches through the point events *in query order* and captures *all possible ways* of matching points from distinct input events. In this example, the point event a_1^2 produces a partial match, (a_1^2), of the pattern (A,B,C). Then at time t=3, we will select b_3^3 to extend the partial match to (a_1^2, b_3^3); at the same time, we will 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 total, skip till any match generates 3 partial matches and 6 complete matches to produce the same results as 45 possible worlds would produce.

In summary, given a point event that creates an initial partial match of a pattern, the *skip till any match* strategy dynamically constructs a directed acyclic graph (DAG) rooted at this event and spanning the point event stream, such that each path in this DAG corresponds to a unique partial or complete point match. If a query uses the skip till any match strategy, we already have the correct matches, which we prove in Appendix A.2.

Extension for "skip till next match" queries. A *skip till next match* query means that the pattern matching process selects only the *first* relevant event for each pattern component, hence producing fewer results than a skip till any match query. While this strategy is easier to support than skip till any match in a deterministic world, under temporal uncertainty it becomes more difficult due to the uncertainty

regarding the "first" relevant event.

Fig. 2(b) 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 first b after a_1? The answer is yes if 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. Evidently, any b that overlaps with a_1, e.g., b_2 and b_3, can be the next event right after a_1 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, ... 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 since 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. NELT is of particular importance because of its *dichotomy* property: 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, but none of the points of e that occurs after e_{m_j}'s NELT can. 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. Then 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 extend the function, $next()$, that a pattern query engine uses to match events with pattern components. Given a pattern, $next(m, e)$ is true iff event e can extend the partial match m of the pattern. To support skip till next match queries, we revise $next(m, e)$ such that the matching stops when the time marked by the NELT of the last event in m is reached. The detailed algorithm for NELT is given in Algorithm 2 in Appendix A.1.

In summary, skill till next match queries are supported by running an existing pattern engine using the skip till any match strategy and extending the function $next()$ with the use of NELT. We prove the correctness of our method in Appendix A.2. Finally, note that due to temporal uncertainty, skill till next match queries cannot be run directly on the point event stream using the same strategy. For example, starting from a_1^3 in Fig. 2(b), the skip till next match strategy will produce only one point match, (a_1^3, b_3^4), while many other matches starting with a_1^3 exist in the possible worlds.

Match Collapsing. The final issue is to collapse point matches into query matches as defined in Section 3.3. In particular, without enumerating all possible worlds, how do we compute the time range and confidence for each unique signature of point matches?

Consider the set of point matches, $\{ m : (e_{m_1}^{t_1}, \ldots, e_{m_\ell}^{t_\ell}) \}$, that share the same signature α, denoted by S_α. The tightest time range for all the point matches is $[min_m(e_{m_1}^{t_1}.\text{lower}), max_m(e_{m_\ell}^{t_\ell}.\text{upper})]$. The remaining task is to compute the confidence.

For a *skip till any match* query, the confidence $C_{any}(\alpha)$ equals:

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

This calculation is correct because the probability of the point match $e_{m_1}^{t_1}, \ldots, e_{m_\ell}^{t_\ell}$ is the product of the probabilities of its individual point 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_1^2, b_3^3, c_2^5 to be a match of the query in Fig. 1, we require that event c_4 does not occur at time 4. Formally, a potential point match $m = (e_{m_1}^{t_1}, \ldots, e_{m_\ell}^{t_\ell})$ is a true match iff (1) $t_1 < \ldots < t_\ell$, and (2) for each $e_{m_j}^{t_j}$ $(2 \le j \le \ell)$, no point event matching E_j occurs between $e_{m_{j-1}}^{t_{j-1}}$ and $e_{m_j}^{t_j}$. 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 skip-till-next match, $C_{next}(\alpha)$, equals:

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

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 of the pattern components. Thus, any event can occur in at most one $\Theta_j(m)$ set, and these sets being empty are independent of each other. Hence, we can rewrite Eq. 2 as:

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

The equation above leads to a memoization-based algorithm to compute $C_{next}^1(\alpha)$. For all point matches in S_α, it computes the quantity $\mathbb{P}\left[e_{m_{j-1}}^{t_{j-1}}\right] \mathbb{P}\left[e_{m_j}^{t_j}\right] \mathbb{P}\left[\Theta_j(m) = \emptyset\right]$ 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 $e_{m_{j-1}}^{t_{j-1}}$ and $e_{m_j}^{t_j}$. This algorithm is detailed in Algorithm 3 in Appendix A.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 $\Theta(m) = \emptyset$ based on independence. So,

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

Our algorithm extends that of the first case by using the event index to also compute $\mathbb{P}\left[\Theta_j(m) = \emptyset | \{e_{q_i}^{t_i}\}\right]$ as well as memoization.

5. 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 cost of enumerating a potentially large number of point matches. It is not obvious how to efficiently find the exact set of query matches in this way. Below, we present evaluation methods and optimizations that together achieve this goal. It is worth noting that the key ideas developed in the point-based framework, such as those for supporting skip till next match queries and computing the confidence, are shared in the event-based framework.

5.1 The Query Order Evaluation Method

To focus on the main idea, we start with two temporary assumptions about the evaluation of a pattern $p = (E_1, \ldots, E_\ell)$ on an event stream: (1) Each event can match only one of the ℓ components. (2) If two events match two different pattern components, E_i and E_j $(i < j)$, and overlap in time, then the event matching E_i is presented

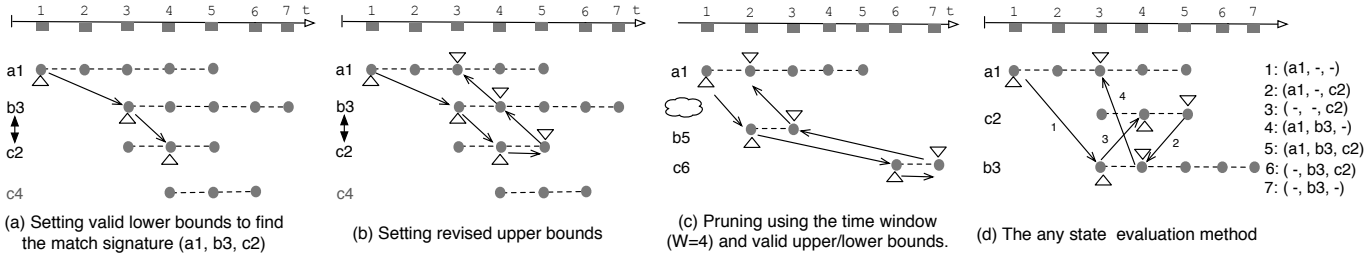


Figure 3: Illustration of the event-based evaluation (assuming that events are presented in query order).

before the event matching E_j in the stream. These assumptions will be eliminated later using a flexible evaluation algorithm.

A Three-Pass Algorithm. Even with these simplifying assumptions, it is still quite subtle to find event-based matches. To do so, we will walk through the events in a potential match 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 for pruning of the potential match, and third backwards again, pruning the match further by the constraint that all the matched events must fit in the query window, W.

1. *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 by both the event type and the predicates applied to 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 at 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 ($a_1$,$b_3$,$c_2$).

2. *Pruning based on Upper Bounds*. 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, that is, $e_{m_j}.\text{vlb} < e_{m_j}.\text{rub}$; otherwise, the match m is pruned.

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 $a_1.\text{rub} = 3$. The match is preserved in this step.

3. *Pruning based on the Window*. Finally we consider the query window size, W. We introduce the notion of *valid upper bound* (vub) to bound the range of each event that can form a valid match. We formally define it in two cases. 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 $T_m = e_{m_1}.\text{rub} + W - 1$. In the first case, $e_{m_\ell}.\text{rub} \le T_m$. Then the revised upper bounds are in fact the valid upper bounds, and we have validated the match m. Otherwise, we walk back down the third time computing the valid upper bounds as follows: $e_{m_\ell}.\text{vub} = T_m$. Inductively, assuming $e_{m_{j+1}}.\text{vub}$ is defined, we let $e_{m_j}.\text{vub} = \min(e_{m_{j+1}}.\text{vub} - 1, e_{m_j}.\text{upper})$. At any time during this pass, if some event e_{m_j} has $e_{m_j}.\text{vub} < e_{m_j}.\text{vlb}$, 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: $a_1.\text{vlb} = 1$, $b_5.\text{vlb} = 2$, and $c_6.\text{vlb} = 6$. After the second pass, we have: $c_6.\text{rub} = 7$, $b_5.\text{rub} = 3$, and $a_1.\text{rub} = 2$. Then we have $T_m = a_1.\text{rub} + W - 1 = 5$. Since $c_6.\text{rub} = 7 > T_m = 5$, we start the third pass, in which we set $c_6.\text{vub} = T_m$ and can immediately see that $c_6.\text{vub} = 5 < c_6.\text{vlb} = 6$. So the match is pruned.

An Incremental Algorithm. To prune non-viable matches as early as possible, our implementation actually uses an incremental algorithm that runs ext() forward on the event stream, building the match signature and pruning the match simultaneously. The main idea is that as we scan events forward to extend the partial match, $m = (e_{m_1}, \ldots, e_{m_j})$, we can already run backwards over m, treating the event e_{m_j} as if it were the last event in the pattern and computing the revised upper bounds and valid upper bounds as described above. At any time during this process, if an event in m has an empty valid range, this partial match can be pruned immediately. The details of this algorithm and its correctness proof are presented in §B.1.

Given events presented in query order, the incremental algorithm evaluates *skip till any match* queries using ext() and the skip till any match strategy, and *skip till next match* queries by further augmenting ext() using NELT as proposed in §4. For details see §B.1.

Computing the Confidence. We last compute the confidence of a match. For a *skip till any match* query, in the point-based evaluation framework we can simply sum up the probabilities of the point matches sharing the signature. In the event-based framework, we are only given the events in the match, so we need to enumerate valid point matches in those events' valid intervals and sum up their probabilities. For a skip till next match query, we can reuse the confidence algorithm in the point-based framework, again by quickly constructing point matches from those events in the match.

5.2 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, $\text{ext}(\emptyset, a_1)$ will select a_1, $\text{ext}((a_1), c_2)$ will skip c_2, and $\text{ext}((a_1), b_3)$ 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 then select any event that can potentially match another pattern component until the match completes or fails—we call this new method *"any state" evaluation*. In our work, we refer to the partial processing result using this method as a "run". 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 so that the previous run can be extended in a different way later. Fig. 3(d) shows the any state evaluation method for events a_1, c_2, and b_3, including the evolution of runs and computation of upper and lower bounds. The details of the method are given in §B.2.

5.3 Optimizations

Sorting for Query Order Evaluation. We observe that the any state evaluation method, which evaluates events in arrival order, is much more complex than the query order evaluation method, which assumes events to be presented in query order, If we can sort the input stream to present events in query order, 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. To do so, we use buffering and available ordering information. See the appendix (§B.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.

6. PERFORMANCE EVALUATION

We have implemented both evaluation frameworks using the SASE pattern query engine [1]. In this section, we evaluate these frameworks using both synthetic data streams with controlled properties and real traces collected from MapReduce cluster monitoring. The details of our experimental setup are given in Appendix C.

6.1 Evaluation using Synthetic Streams

We implemented a data generator that creates events in increasing order of time t but assigns to each event an uncertainty interval $[t - \delta,\ t + \delta]$; we call δ the half uncertainty interval size. The query pattern $(E_1, \ldots, E_\ell)$ is controlled by the time window size W (default 100 units), the pattern length ℓ (default 3), the event selection strategy, and the selectivity of each pattern component.

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 high cost. The event-based method is not very 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 were observed for varied W and ℓ values.

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.

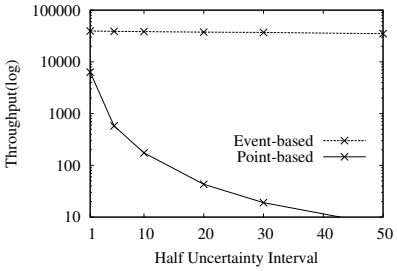
We then examine the effect of event frequencies. We keep the query selectivity roughly the same, increase the percentage of events matching the first pattern component E_1 by adjusting its predicate, and decrease that for the last pattern component E_ℓ accordingly. As a result, more events can match 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. This strategy aims to find the "first" match of each pattern component in a deterministic world. Under temporal uncertainty, however, it requires more work to handle such first matches, including the use of the Next Event's Latest Time (NELT) and the more complex confidence computation. 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 somewhat. It is because 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 $\delta \geq 30$.

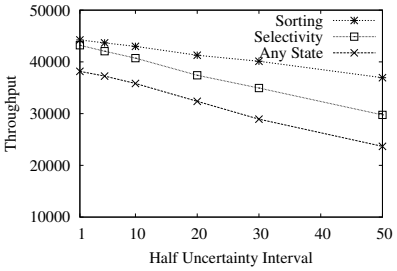
6.2 Evaluation in Cluster Monitoring

To evaluate our techniques in real-world settings, we performed a case study of Hadoop cluster monitoring (as detailed in Appendix C). The Hadoop system logs events, such as the start and end times of map and reduce tasks, in unix time (us). This cluster also uses the Ganglia monitoring tool [11] 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 pattern queries, similar to Query 1 in §3, to study the effects of Hadoop operations on the load on each compute node. These queries require the use of uncertainty intervals because of (1) the granularity mismatch between Hadoop events (in us) and Ganglia events (once 15 seconds) and (2) the clock synchronization problem in the cluster. 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 queries on the merged trace of the Hadoop log and the Ganglia event stream.

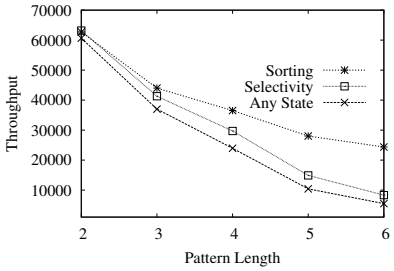
For each query, we used four combinations of the event selection strategy and the selectivity of the predicate on the last pattern component (the predicates on other pattern components were fixed and



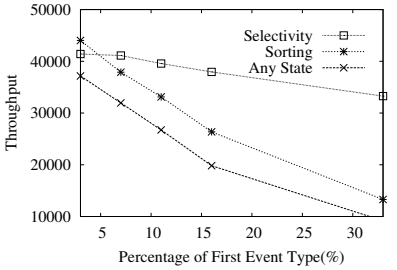
(a) Varying uncertainty interval (skip-till-any)



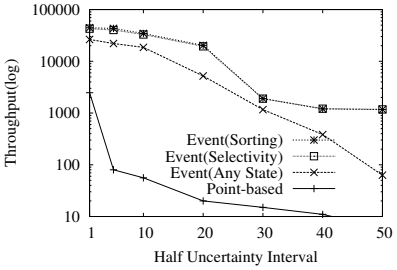
(b) Event-based optimizations with varied uncertainty intervals (skip-till-any)



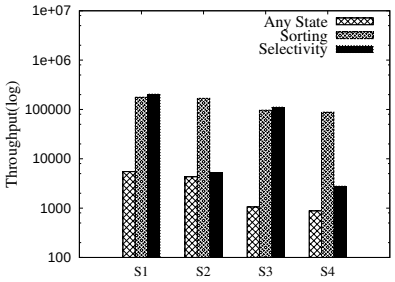
(c) Event-based optimizations with varied pattern lengths (skip-till-any)



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



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



(f) Case study of cluster monitoring (Query1, time unit: 0.1 sec)

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

not so selective). (S1) a skip till *any* match query with a *selective* predicate; (S2) skip till *any* match and a *nonselective* predicate; (S3) skip till *next* match and a *selective* predicate; and (S4) skip till *next* match and a *nonselective* predicate. Fig. 4(f) shows the results. We can see that skip till any strategy queries are faster than skip till next match queries for the reasons explained above. Moreover, we see that sorting always works well. Selectivity-based optimization works well for S1 and S3 where the last predicate is selective. In these cases, this method can prune many expired events when the last stack remains empty. For S2 and S4 where the last predicate is nonselective, this method cannot remove many events to save time. Other queries show similar results as shown in [23].

7. CONCLUSIONS

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

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] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB J.*, 16(4):523–544, 2007.

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

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

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

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

[12] E. Koskinen and J. Jannotti. Borderpatrol: isolating events for black-box tracing. In *EuroSys*, 191–203, 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*, 649–660, 2008.

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

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

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

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

[19] T. Tran, C. Sutton, et al. Probabilistic inference over RFID streams in mobile environments. In *ICDE*, 1096-1107, 2009.

[20] 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.

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

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

[23] H. Zhang, Y. Diao, et al. Recognizing Patterns in Streams with Imprecise Timestamps *UMass Tech Report* UM-CS-2010-025, 2010.

APPENDIX

A. POINT-BASED EVALUATION

In this appendix, we give the pseudocode of algorithms in the point-based evaluation framework and prove their correctness.

A.1 Pseudocode of Algorithms

Algorithm 1 shows the point-based evaluation procedure.

Algorithm 1 Point-based Evaluation

Input: Event Stream S **, Pattern** $(E_1, ...E_\ell)$
  for Each event e_i in S **do**
    Set Now to $[max^i_{j=1}(e_j.lower), max^i_{j=1}(e_j.upper)]$
    Set t_{emit} to $Now.lower - U_{max}$
    for $t = e_i$.lower to e_i.upper **do**
      Generate the point event e^t_i
      Add e^t_i to the event buffer
    end for
    for Each point event e^t_j in the event buffer **do**
      if $t = t_{emit}$ **then**
        Emit e^t_j to the pattern matching engine
        if The query uses skip-till-any-match **then**
          Run the engine using skip-till-any-match strategy
        end if
        if The query uses skip-till-next-match **then**
          Run the engine using skip-till-any-match strategy and new $next()$ with NELT by calling Algorithm 2
        end if
      end if
    end for
    for Each point match $m : (e^{t_1}_{m_1}, ..., e^{t_\ell}_{m_\ell})$ in the match buffer **do**
      if $t_{emit} > e_{m_1}$.upper $+ W$ **then**
        Collapse matches with the same signature as m
        Compute the time range and the confidence of the match by calling Alogirhtm 3
      end if
    end for
  end for

Algorithm 2 shows the function $next()$ extended with the use of Next Event's Latest Time (NELT). In this algorithm, we incrementally compute the NELT of an event e. Every time that a partial match $m : (e_{m_1}, ...e_{m_j})$ decides whether to select event e that can potentially match E_{j+1}, it compares e_{m_j}.NELT with e.lower. If the e.lower $< e_{m_j}$.NELT, m will select e. Then it will compare its e_{m_j}.NELT with e.upper. If the e_{m_j}.NELT is larger, then we update e_{m_j}.NELT to $e.upper$. At the same time, we need to check runs that have passed the previous NELT in case that they fail in the check using the new NELT. We will keep updating NELT of each event until t_{emit} has advanced the point that no future events can change the NELT. When a match has selected events for all pattern components, we will not return it until we are sure that there is no chance to change NELT's of its events.

Algorithm 3 shows the computation of the match confidence for a *skip till next match* query in the point-based framework. This 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. Then we can enumerate these events when we compute the confidence as shown in Eq. (4). For details, please see our tech report [23].

A.2 Correctness Proofs

Skip till any match. We first prove the correctness of our point-based evaluation algorithm for skip till any match queries.

Algorithm 2 Pattern Matching using $next()$ with NELT

Input: Event e**, Pattern** $(E_1, ...E_\ell)$
  for Each partial match $m : (e_{m_1}, ...e_{m_j})$ in the buffer **do**
    if $e_{m_j}.NELT$ has not been initialized **then**
      Initialize $e_{m_j}.NELT$ to $+\infty$
    end if
    if e matches E_{j+1} **then**
      if e.lower $< e_{m_j}.NELT$ **then**
        $next(m, e) := true$
        if e.upper $< e_{m_j}.NELT$ **then**
          $e_{m_j}.NELT := e$.upper
          for Every other partial match m' that contains e_{m_j} **do**
            if $e_{m'_{j+1}}$.lower $> e_{m_j}.NELT$ **then**
              Remove m'
            end if
          end for
        end if
        if $j + 1 = \ell$ and $e_{m_j}.NELT < t_{emit}$ **then**
          Return m as a complete match
        end if
      end if
    end if
  end for

PROOF. For a skip till any match query, pattern matching 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 will 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 $(e^{i_1}_{m_1}, e^{i_2}_{m_2}, ...e^{i_\ell}_{m_\ell})$ 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 point event stream, point events are ordered by timestamps, so, we have $(e^{i_1}_{m_1} \prec e^{i_2}_{m_2} \prec ... \prec e^{i_\ell}_{m_\ell})$. By definition, skip till any match will have one such run that first selects $e^{i_1}_{m_1}$, ignores other point events until $e^{i_2}_{m_2}$ arrives, selects $e^{i_2}_{m_2}$, ignores other point events until $e^{i_3}_{m_3}$ 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 for skip till next match queries. Recall that our algorithm handles such queries by using the skip till any match strategy and extending $next()$ with the Next Event's Latest Time (NELT) to prune potential matches.

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

(1) Any point event, denoted by e^t_i, that starts **after** e_{m_j}'s NELT cannot be used to extend the partial match $(e^{t_1}_{m_1}, \ldots, e^{t_j}_{m_j})$ in any possible world. 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^t_i occurs after the e_{m_j}'s NELT, it will surely be preceded by the point event e^{NELT}_k in any possible world, and hence cannot be the next to e_{m_j}.

(2) Every point event, e^t_i, that can potentially match the pattern

Algorithm 3 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
    $Confidence_m$ += $\frac{\prod_{i=1}^{\ell-1} \mathbb{P}\left[e_{m_i}^{t_i} \prec e_{m_{i+1}}^{t_{i+1}}\right] \times \mathbb{P}\left[\text{None of } \bar{S}_{m_{i+1}} \text{ between } [t_i, t_{i+1}]\right]}{\prod_{i=2}^{\ell-1} \mathbb{P}\left[e_{m_i}^{t_i}\right]}$
  end for
end if

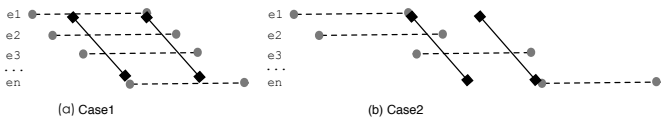


Figure 5: Two cases of constructing point-matches

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. 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 start 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 few that occur at the same time NELT). □

B. EVENT-BASED EVALUATION

In this appendix, we give details and optimizations of evaluation methods in the event-based framework, and prove their correctness.

B.1 Query Order Evaluation

B.1.1 Three Pass Algorithm

To prove the correctness of our three pass algorithm (in §5.1), we show that it obtains 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, and the timestamps of point events should be larger than or equal to their corresponding valid lower bounds. We show this by induction. When the pattern length is one, obviously, for a point-match $e_1^{t_1}$, we can capture the event e_1 in the event-based framework, and have that $t_1 \geq e_1$.vlb and $t_1 \leq e_1$.rub. Then we assume when the sequence length is n, for a point match $(e_1^{t_1}, e_2^{t_2}, ...e_n^{t_n})$, we can find an event match $(e_1, e_2, ...e_n)$, and we have $t_i \geq e_i$.vlb and $t_i \leq e_i$.rub $(1 \leq i \leq n)$. When the sequence length is $n+1$, if the point-based framework gets a match $(e_1^{t_1}, e_2^{t_2}, ...e_n^{t_n}, e_{n+1}^{t_{n+1}})$, by the assumption we know the event-based framework can capture the first n events $(e_1^{t_1}, e_2^{t_2}, ...e_n^{t_n})$ and $t_n \geq e_n$.vlb, $t_n \leq e_n$.rub. From the point-match, we know $t_{n+1} > t_n$. Also we know $t_{n+1} \geq e_{n+1}$.lower and $t_{n+1} \leq e_{n+1}$.upper. By the definition of valid lower bound, we know $e_{n+1}.\text{vlb} = max(e_n.\text{vlb} + 1, e_{n+1}.lower)$, and e_{n+1}.rub $= e_{n+1}$.upper. So we can get $t_{n+1} \geq e_{n+1}$.vlb and $t_{n+1} \leq e_{n+1}$.rub. So e_{n+1} will be selected for the match.

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. Our proof above did not consider the time window constraint. We next show that the event-based framework can support the time window correctly.

PROOF. First, we show if a point match satisfies the time window W, the event match with the same signature will also pass the time window check. If the point match is $(e_1^{t_1}, e_2^{t_2}, ...e_n^{t_n})$, we have $t_n - t_1 < W$. If we do not need the third pass, then e_i.vub $= e_i$.rub, and thus t_i will be in e_i's valid range $(1 \leq i \leq n)$. If we need the third pass to compute the valid upper bounds, e_n.vub $= e_1$.rub $+ W - 1 > t_1 + W - 1 \geq t_n$, so t_n is still in e_n's valid interval. Then $e_{n-1}.\text{vub} = min(e_n.\text{vub} - 1, e_{n-1}.upper)$. Since $t_{n-1} < t_n \leq e_n$.vub and $t_{n-1} \leq e_{n-1}$.upper, we have $t_{n-1} \leq e_{n-1}$.vub and t_{n-1} is in e_{n-1}'s valid range. Repeating this, we can prove that t_i is in e_i's valid range for other events in this match. So the event match will be retained from the time window check in both cases.

Then we show that given any event match m satisfying the time window W, at least one point match with the same signature will pass the time window check. We can prove this by constructing a point match with the points from an event match. We consider two cases that are distinguished by e_n.vlb $- e_1$.vub.

Case 1: e_n.vlb $- e_1$.vub $< n - 1$, as shown in Fig. 5(a).

In this case, we will pick $(e_1^{e_1.\text{vub}}, e_2^{e_1.\text{vub}+1}, ...e_n^{e_1.\text{vub}+n-1})$ as the point match. Since these points are consecutive on timestamps, we only need to prove that these timestamps are in valid ranges of these events, i.e., $e_i.\text{vlb} \leq e_1.\text{vub} + i - 1 \leq e_i.\text{vub}$. We can show this by contradiction: Assume that $e_1.\text{vub} + i - 1$ is out of e_i's valid range. Then $e_1.\text{vub} + i - 1 > e_i.\text{vub}$ or $e_1.\text{vub} + i - 1 < e_i.\text{vlb}$. In the former case, it will contradict with the valid upper bound definition. In the latter case, we can get $e_1.\text{vub} + i - 1 < e_i.\text{vlb} \leq e_n.\text{vlb} - n + i$, then we can get $e_n.\text{vlb} - e_1.\text{vub} \geq n - 1$, which contradicts the case condition.

Case 2: e_n.vlb $- e_1$.vub $\geq n - 1$, as shown in Fig. 5(b).

In this case, we will pick $e_1^{e_1.\text{vub}}$ as the first point event of the point match, and pick $e_n^{e_n.\text{vlb}}$ as the last point event of the point match. For $e_i (1 < i < n)$, we will choose the point at $t_i =$

$min(e_i.\text{vub}, e_n.\text{vlb} - n + i)$. We need to show the timestamps of these points are monotonically increasing. First, we show that the valid range of $e_i (1 < i < n)$ overlaps with range $[e_1.\text{vub} + i - 1, e_n.\text{vlb} - n + i]$. We can show this by contradiction. Assume that there is no overlap. Then $e_i.\text{vlb} > e_n.\text{vlb} - n + i$ or $e_i.\text{vub} < e_1.\text{vub} + i - 1$. According to the definition of the valid lower bound and valid upper bound, $e_i.\text{vlb} \leq e_n.\text{vlb}$ and $e_i.\text{vub} \geq e_1.\text{vub} + i - 1$. The contradictions are obvious. And actually t_i is the upper bound of the overlap between its valid range and the range $[e_1.\text{vub} + i - 1, e_n.\text{vlb} - n + i]$. Then we need to show that $t_i > t_{i-1} (1 < i < n)$. If $t_i = e_n.\text{vlb} - n + i$, then t_i is larger than all points during $[e_{i-1}.\text{vub} + i - 1, e_n.\text{vlb} - n + i - 1]$, and so $t_i > t_{i-1}$. If $t_i = e_i.\text{vub}$, then we need to consider two cases: if $t_{i-1} = e_{i-1}.\text{vub}$, by definition we can get $t_i > t_{i-1}$; if $t_{i-1} = e_n.\text{vlb} - n + i - 1$, we know $t_{i-1} = e_n.\text{vlb} - n + i - 1 < e_n.\text{vlb} - n + i \leq t_i$. So t_i is always larger than t_{i-1}. □

Time Range. For the correctness of time range, we need to prove that the valid range bounded by the valid lower bound and valid upper bound is correct. It means that all the points that can form a point match are included in the valid range, and all the points in the valid range can construct a point match.

PROOF. In the proofs for the match signature and time window constraint, we have already shown that any point that can form a point match is in the valid range of the event. Then we need to show that any point in the valid range can construct a point match. We prove this by contradiction. Assume that we have an event match $(e_1, e_2, ...e_n)$. We assume that there exists a point $e_j^{t_j}$ in e_j's valid range that cannot form a match by selecting points from the other events' valid range. If it cannot pick a point from e_i $(i < j)$, it means that either $t_j - i + 1 < e_i.\text{vlb}$ or $t_j - e_i.\text{vub} > W$. The former case contradicts the definition of valid lower bound, which can tell us $e_i + i - 1 < e_j.\text{vlb} \leq t_j$. The latter case contradicts the time window constraint, by which we can get $t_j - e_i.\text{vub} \leq e_j.\text{vub} - e_i.\text{vub} \leq e_n.\text{vub} - e_1.\text{vub} < W$. If $e_j^{t_j}$ cannot pick a point from e_i $(i > j)$, we can obtain similar contradictions. □

B.1.2 Incremental Algorithm

To prune non-viable matches early, our implementation actually uses incremental computation as the ext function runs forward on the event stream. Our algorithm incrementally computes valid lower and upper bounds of events and evaluates the window constraint. While the valid lower and upper bounds initially may not be as tight as the true ones defined in the three-pass algorithm, they will converge to the true ones when the match becomes complete.

Consider a partial match $m = \emptyset$ or $(e_{m_1}, \ldots, e_{m_j})$, and the current event e in the input. The incremental algorithm, as shown in Algorithm 4, takes four main steps:

1. Compute e's valid lower bound given m. Initialize e's valid upper bound using its own upper bound and check whether its valid interval is empty. (Lines 4-6).
2. Compute the rub of the events in reverse pattern order, i.e., from e_{m_j} down to e_{m_1}. Check whether the valid interval of each event is empty (Lines 7-10).
3. If $e_{m_1}.\text{rub} + W < e.upper$, compute the vub in a third pass. Again, check whether the valid interval of each event is empty (Lines 11-19).
4. If the partial match passes all checks, perform the pattern matching (Lines 20-26).

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 $c_6.\text{vlb} = 6$ and $c_6.\text{vub} = 7$. Step 2 sets $b_5.\text{rub} = 3$ and $a_1.\text{rub} = 2$. Then in Step 3, the window constraint $W = 4$ is expressed as $c_6.\text{rub} > a_1.\text{vub} + 4 - 1 = 5$. So we should set $c_6.\text{vub} := 5$, and then $c_6.\text{vub} - c_6.\text{vlb} < 0$ That is c_6's valid interval is negative, and c_6 cannot be included in a match starting with a_1. In this example, c_6 is pruned.

Algorithm 4 Incremental Method for Query Order Evaluation

```
Input: Event Stream S, Pattern (E_1, ..., E_ℓ)
 1: for Each event e in s do
 2:   for Each partial match m (e_{m_1}, e_{m_2}, ..., e_{m_j}) in the buffer do
 3:     if e satisfies query component E_{j+1} then
 4:       e.vlb := min(e_{m_j}.lower + 1, e.vlb)
 5:       if e.vlb ≤ e.upper then
 6:         e.rub := e.upper
 7:         for each event e_{m_i} (1 ≤ i ≤ j) in r do
 8:           e_{m_i}.rub := min(e_{m_{i+1}}.rub − 1, e_{m_i}.upper)
 9:           Check if e_{m_i}.rub − e_{m_i}.vlb < 0
10:         end for
11:         if e_{m_i}.rub ≥ e_{m_i}.vlb (1 ≤ i ≤ j + 1) then
12:           if e_{m_1}.rub + W < e.rub then
13:             e.vub := e_{m_1}.rub + W − 1
14:             Check if e.vub − e.vlb < 0
15:             if e.vub = e.vlb > 0 then
16:               for each event e_{m_i} (1 ≤ i ≤ j) in r do
17:                 e_{m_i}.vub := min(e_{m_{i+1}}.vub − 1, e_{m_i}.upper)
18:                 Check if e_{m_i}.vub − e_{m_i}.vlb < 0
19:               end for
20:               if e_{m_i}.vub ≥ e_{m_i}.vlb (1 ≤ i ≤ j + 1) then
21:                 if Using skip till any match strategy then
22:                   ext(m, e) := true
23:                 end if
24:                 if Using skip till next match strategy then
25:                   Call Pattern Matching with NELT for Event-
                      based Framework
26:                 end if
27:               end if
28:             end if
29:           end if
30:         end if
31:       end if
32:     end if
33:   end for
34: end for
```

Then we need to show the equivalence of the incremental method and the three pass method. The valid lower bounds computed by the two methods are exactly the same. For the valid upper bounds, when we see later events we either keep them or shrink them so these bounds are non-increasing. Since the temporary valid upper bounds in the incremental method will not be smaller than the final valid upper bounds, our early pruning will not cause loss of results. For the final valid upper bounds, the two methods both start the computation from the last event's upper bound so will produce the same results. The formal proof is given in our tech report [23].

B.1.3 Support of Skip Till Next Match

The previous proofs show that the event-based framework can produce the same results for skip till any match queries. To support skip till next match queries, we also compute the NELT to filter events that cannot form a match. In the point-based framework, we do not select events that happen after the NELT. In the event-based framework, we will shrink the next event's valid upper bound to the current event's NELT, and if this causes the next event's valid upper bound to be less than its valid lower bound, then we prune this partial match. The detailed algorithm is similar to Algorithm 2, hence omitted. Next we show that the results of the event-based

framework remain the same as the point-based framework.

PROOF. Since we use the same method to compute the NELT, it is the same under two frameworks. First we show that when we remove a point by NELT in the point-based framework, the point will not appear in the valid range of the event in the event-based framework. From the definition, this is obvious. In the other direction, after we shrink the valid upper bound by the NELT, of course we will not pick points after the NELT to build the point match. Since we prove the correctness of time range and window constraints with the assumption that we already have the valid ranges, and the NELT operation only shrinks the valid ranges, the correctness proof will still hold for the remaining part after shrinkage. □

B.2 The Any State Evaluation Method

The any state evaluation method is an incremental algorithm that runs directly on the event stream, without the assumption that events are presented in query order. Given an event e, a run γ, and the set of events m selected in γ , this 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.
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'.

The pseudocode and the correctness proof of this method is given in our tech report [23].

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 $(-, b_3, -)$. 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 §4 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 $\gamma_{tll} < now.lower - U_{max}$.

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 $e_{m_j}.\text{upper} < now.lower - U_{max}$, we know that no future event can match E_{j-1}, and hence can safely prune γ.

B.3 Sorting for Query Order Evaluation

We propose an optimization for sorting for query order evaluation: 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, $e.\text{upper} < now.lower - U_{max}$ (the earliest time of an unseen event given the arrival order constraint). (2) Many stream systems use heartbeats [18] or punctuations [13, 20] 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. EXPERIMENTAL SETUP

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.

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 have an uncertainty interval $[t - \delta, t + \delta]$, with δ called the half uncertainty interval size. Each stream contains 0.1 to 1 million events. 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

Query workloads are controlled by the following parameters: 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 ($1 \leq j \leq \ell$).

Case Study. Our case study of MapReduce cluster monitoring ran a Hadoop job for inverted index construction on 457GB of web pages using a 11-node research cluster. 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 [11] to measure the max and average load on each compute node, once every 15 seconds.

Our monitoring queries study the effects of Hadoop operations on the load on each compute node. These queries require the use of uncertainty intervals. The first reason is the granularity mismatch between Hadoop events (in us) 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 generated uncertainty intervals as described in §6.