ABSTRACT

Pattern queries are widely used in complex event processing (CEP) systems. Existing pattern matching techniques, however, can provide only limited performance for expensive queries in real-world applications, which may involve Kleene closure patterns, flexible event selection strategies, and events with imprecise timestamps. To support these expensive queries with high performance, we begin our study by analyzing the complexity of pattern queries, with a focus on the fundamental understanding of which features make pattern queries more expressive and at the same time more computationally expensive. This analysis allows us to identify performance bottlenecks in processing those expensive queries, and provides key insights for us to develop a series of optimizations to mitigate those bottlenecks. Microbenchmark results show superior performance of our system for expensive pattern queries while most state-of-the-art systems suffer from poor performance. A thorough case study on Hadoop cluster monitoring further demonstrates the efficiency and effectiveness of our proposed techniques.

1. INTRODUCTION

In Complex Event Processing (CEP), event streams are processed in real-time through filtering, correlation, aggregation, and transformation, to derive high-level, actionable information. CEP is now a crucial component in many IT systems in business. For instance, it is intensively used in financial services for stock trading based on market data feeds; fraud detection where credit cards with a series of increasing charges in a foreign state are flagged; transportation where airline companies use CEP products for real-time tracking of flights, baggage handling, and transfer of passengers [17]. Besides these well-known applications, CEP is gaining importance in a number of emerging applications, including

- 

- Logistic management, enabled by sensor and RFID technology advances, is gaining adoption in hospitals [26], supply chains [17], and aerospace applications. While pattern queries have been used for complex event processing in this area, query evaluation is often complicated by the uncertainty of the occurrence time and value of events because they are derived through probabilistic inference from incomplete, noisy raw data streams [9, 27].

Challenges. Among many challenges in CEP, this paper focuses on efficient evaluation of pattern queries. Pattern query processing extends relational stream processing with a sequence-based model (in contrast to the traditional set-based model). Hence it supports a wide range of features concerning the temporal correlation of events, including sequencing of events; windowing for restricting a pattern to a specific time period; negation for non-occurrence of events; and Kleene closure for collecting a finite yet unbounded number of events. While various subsets of these features have been supported in prior work on pattern matching in CEP [1, 11, 20, 21, 23, 28] and regular expression matching, this work is motivated by our observation that two unique features of CEP can dramatically increase the complexity of pattern queries, rendering existing solutions insufficient:

- Event selection strategies: A fundamental difference between pattern queries in CEP and regular expression matching is that the set of events that match a particular pattern can be widely dispersed in one or multiple input streams— they are often not contiguous in any input stream or in any

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 20XX ACM X-xxxxxx-xx-x/xx/xx ...$15.00.
simple partition of the stream. The strategy on how to select those events relevant to a pattern is called **event selection strategy** in the literature. Event selection strategies can vary widely depending on the application, from the most strict form of selecting events only continuously in the input (**strict or partition contiguity**), to the more flexible form of skipping irrelevant events until finding the relevant events to match the pattern (**skip till next match**), to the most flexible form of finding all possible ways to match the pattern in the input (**skip till any match**). As shown later in this study, the increased flexibility in event selection leads to significantly increased complexity of pattern queries, with most existing solutions [1, 20, 21, 28] unable to support the most flexible strategy for Kleene closure or even simple pattern queries.

**Imprecise timestamps:** The timestamps in input events can be imprecise for several reasons [30]: (i) The events are inferred using probabilistic algorithms from incomplete, noisy sensor streams, such as in the logistics application. Hence, the inferred occurrence time in an event is behind the actual occurrence time with an unknown lag. (ii) Event occurrence times in different inputs are subject to granularity mismatch. In cluster monitoring, for instance, Ganglia returns peak CPU utilization every 15 seconds while Hadoop returns task progress reports at the granularity of a microsecond. Understanding which task causes a CPU spike requires ordering the events on CPU utilization and the events on Hadoop task progress by occurrence time, but here it is hard to order them because one cannot tell exactly where a CPU spike occurs in a 15-second period. (iii) There is also the clock synchronization problem in distributed environments. For these reasons, CEP systems cannot arrange the events from all inputs into a single stream with the right order property (total order or strict partial order) required for pattern matching. As we shall show, techniques for handling imprecise timestamps [30] work only for simple pattern queries and quickly deteriorate for more complex queries.

**Contributions.** In this paper, we perform a thorough analysis of pattern queries in CEP, with a focus on the fundamental understanding of which query features make them "expensive", formally, which set of query features correspond to which complexity class. As such, our analysis yields a hierarchy of query features and the corresponding complexity classes. Our analysis also includes mapping existing CEP systems into the same hierarchy, hence providing a unified theoretical framework for comparing existing CEP systems. The results of our theoretical study also offer key insights for optimizing those expensive pattern queries. More specifically, our contributions include:

1. **Descriptive Complexity** ([§3]): We begin our study by addressing the question of which features of pattern queries make them "expensive", formally, which set of query features correspond to which complexity class. As such, our analysis yields a hierarchy of query features and the corresponding complexity classes. Our analysis also includes mapping existing CEP systems into the same hierarchy, hence providing a unified theoretical framework for comparing existing CEP systems.

2. **Runtime Complexity** ([§4]): We then extend the problem formulation from checking the existence of a query answer to finding all query answers in an input. This analysis, which we call "runtime analysis", reveals two types of expensive queries: (i) Pattern queries that use Kleene closure under the most flexible event selection strategy, skip till any match, are subject to an exponential number of pattern matches from a given input, hence an exponential cost in computing these matches; (ii) The solution to evaluating Kleene+ pattern queries on events with imprecise timestamps can be constructed based on a known algorithm for evaluating simple pattern queries, but always has to use the skip till any match strategy to avoid missed results, hence incurring a worst-case exponential cost. It has an additional cost of confidence computation for each pattern match, which is also exponential in the worst case. In summary, two bottlenecks in pattern query processing are Kleene closure evaluated under the skip till any match strategy (1) and confidence computation in the case of imprecise timestamps (2).

3. **Optimizations** ([§5]): To address bottleneck (1), we derive an insight from the observed difference between the low-level complexity classes in descriptive complexity analysis (which considers only one match) and exponential complexity in runtime analysis (which considers all pattern matches). Our optimization breaks query evaluation into two parts: pattern matching, which can be shared by many matches, and result construction, which constructs individual results. We propose a series of optimizations to reduce shared pattern matching cost from exponential to polynomial time (sometimes close-to-linear). This also offers a compact way to return all matches if explicit enumeration of them is not needed by the user. To address bottleneck (2), we provide a dynamic programming algorithm to expedite confidence computation and to improve performance when the user increases the confidence threshold for desired matches.

4. **Evaluation with a case study** ([§6]): We compare our new system with a number of state-of-the-art pattern query systems including SASE [1, 28], ZStream [20], and XSeq [21]. Our microbenchmark results show that our system can mitigate performance bottlenecks in most workloads, while other systems suffer from poor performance for the expensive pattern queries mentioned above. In addition, we perform a case study in cluster monitoring using real Hadoop workloads, system traces, and a range of monitoring queries. We show that our system can automate cluster monitoring using declarative pattern queries, return very insightful results, and support real-time processing even for expensive queries.

## 2. BACKGROUND

In this section, we define a “core language” for pattern queries, introduce its formal semantics, and present an extension to imprecise timestamps. This discussion offers a technical context for our study in the subsequent sections.

### 2.1 A Core Language for Pattern Queries

A number of languages for CEP have been proposed, including SQL-TS [23], Cayuga [11], SASE [1, 28], and CEDR [6]. Although designed with different grammar and syntax, the core features for pattern matching are similar. Below, we define a core language, $L$, for pattern queries, which includes necessary constructs to be useful in real-world applications, but leaves out derived features that do not change the com-
plexity classes shown below.

The core language $L$ employs a simple event model: Each event represents an occurrence of interest; it includes a time-stamp plus other attributes. All input events to the CEP system can be merged into a single stream, ordered by the occurrence time. Then over the ordered stream, a pattern query seeks a series of events that occur in the required temporal order and satisfy other constraints. The constructs in $L$ include:

- **Sequencing** (SEQ) lists the required event types in temporal order, e.g., SEQ(A, B, C), and may assign a variable to refer to each event selected into the match.
- **Kleene closure (+)** collects a finite yet unbounded number of events of a particular type. It is used as a component of the SEQ construct, e.g., SEQ(A, B+, C).
- **Negation (~ or !)** verifies the absence of certain events in a sequence. It is also used as a component of the SEQ construct, e.g., SEQ(A, ~B, C).
- **Value predicates** further specifies value-based constraints on the events addressed in SEQ. For Kleene+, they can be applied to each event ‘e’ considered in Kleene+ by placing a constraint on (a) only e, (b) between e and a fixed number of previous events, or (c) over all the events previously selected in Kleene+ by the use of an aggregate function (see below for examples.). Aggregate functions include standard functions (max, min, count, sum, avg) and user-defined functions.
- **Closure under union, negation and Kleene closure.** Union ($\cup$) can be applied to two patterns, e.g., SEQ(A, B, C) $\cup$ SEQ(A, D, E). Negation (~ or !) can be applied to a SEQ pattern, e.g., ~SEQ(A,B, C). Kleene closure (+) can also be applied to a pattern, e.g., SEQ(A,B,C)+.
- **Windowing** (WITHIN) restricts a pattern to a specific time period.
- **Return** (RETURN) constructs new events for output.

There are other useful constructs such as UNORDERED, at least, and at most [6], however, they can either be derived from the core constructs or do not affect the complexity classes, so we do not include them in $L$.

Table 1 shows two example queries used in our case study on Hadoop cluster monitoring. The queries are written using the syntax used in [1, 20, 26, 28]. Query 1 computes the statistics of running times of mappers in Hadoop: The ‘Pattern’ clause specifies a SEQ pattern with three components: a single event indicating the start of a Hadoop job, followed by a Kleene+ for collating a series of events representing the mappers in the job, followed by an event marking the end of the job. Each component declares a variable to refer to the corresponding event(s), e.g., a, b[i], and c, with the array variable b[i] declared for Kleene+. The ‘Where’ clause uses these variables to specify value-based predicates. Here the predicates require all events to refer to the same job id; such equality comparison across all events can be writing with a shorthand, ‘[job_id]’. The ‘Within’ clause specifies a 1-day window over the pattern. Finally, the ‘Return’ clause constructs each output event to include the average and maximum durations of mappers in each job.

Query 6 finds reducers that cause increasingly imbalanced load across the nodes in a cluster. It has a similar structure as Query 1. A notable difference is the use of an iterator predicate on the Kleene+: b[i] refers to each event of type ‘LoadStd’ considered by Kleene+, and it is required to have a value no less than the value of the previously selected event in option 1, or the maximum value of all previously selected events in option 2 (using aggregate max). These options are equivalent here but show different types of predicates used.

**Event Selection Strategy.** The event selection strategy expresses how to select the events relevant to a pattern from an input mixing relevant and irrelevant ones. Three strategies can be chosen based on the application needs:

- **S1:** Strict or partition contiguity ‘$\exists$’. The most stringent event selection strategy requires the selected events to be contiguous in the input. A close variant is partition contiguity, which partitions the input stream based on a logical condition, e.g., the same task_id, and requires selected events to be continuous in each partition.

- **S2:** Skip till next match ‘$\rightarrow$’. The strategy removes the contiguity requirements and instead, has the ability to skip irrelevant events until it sees the next relevant event to match more of the pattern. Using this strategy, Query 1 can conveniently ignore all irrelevant events, e.g., the reducer events, which are only “noise” to pattern matching but commonly exist in input streams.

- **S3:** Skip till any match ‘$\exists$’. The last strategy offers more flexibility by allowing non-deterministic actions on relevant events: Once seeing a relevant event, this strategy clones the current partial match to a new instance, then it selects the event in the old instance and ignores the event in the new instance. This way, the new instance skips the current event to reserve opportunities for additional future matches. Consider Query 6 using option 1 and a sequence of load std values (0.1, 0.2, 0.15, 0.19, 0.25). The strategy of skip to next match can find only one sequence of non-decreasing values (0.1, 0.2, 0.25). In contrast, skip to any match produces not only the same sequence, (0.1, 0.2, 0.25), by selecting the value 0.2 in one instance, but also a new sequence, (0.1, 0.15, 0.19, 0.25), by skipping 0.2 in a new instance.

### 2.2 Formal Semantics by NFA$^b$ Automata

The formal semantics of pattern queries is usually based on some form of automaton [1, 11, 20]. In this work, we adopt the NFA$^b$ model in [1] to explain the formal semantics. In this model, each query could be represented by a composition of automata where each is a nondeterministic finite automaton (NFA) with a buffer (b) for computing and storing matches. Figure 1 is the NFA$^b$ for Query 6.

**Figure 1:** An NFA$^b$ automaton for Query 6.
ponent of a pattern is represented by one state, and a Kleene+ component by two consecutive states. In Figure 1, the matching process begins at the first state, \( a \). The second state \( b[1] \) is used to start the Kleene closure, and it will select an event into the \( b[1] \) unit of the match buffer. The next state \( b[i] \) selects each additional relevant event into the \( b[i] \) (\( i > 1 \)) unit of the buffer. The next state \( c \) processes the last pattern component after the Kleene closure has been fulfilled. The final state, \( F \), represents a complete match.

**Edges.** Edges associated with a state represent the actions that can be taken at the state. The conditions for these actions are compiled from the event types, value predicates, the time window, and the selection strategy specified in the pattern query. In the interest of space, we will not present detailed compilation rules, but point out that (1) the looping ‘take’ edge on the \( b[i] \) state is where Kleene+ selects an unbounded number of relevant events; (2) all the looping ‘ignore’ edges are set based on the event selection strategy, often to skip irrelevant events.

**NFA\(^a\)** runs. A run of an NFA\(^b\) automaton is an instance of the automaton, and represents a unique partial match of the pattern. A run that reaches the final state yields a complete match. This concept will be used intensively when we analyze runtime complexity in Section 4.

Finally, the language \( \mathcal{L} \) is closed under union, negation, Kleene+, and composition. Any formula in the language can thus be evaluated by a set of NFA\(^a\) automata combined using these four operations.

### 2.3 Extension to an Imprecise Temporal Model

As discussed earlier, due to granularity mismatch, clock synchronization problems, etc., in many applications we do not have precise timestamps to produce a combined stream that is guaranteed to be sorted by occurrence times. For example, Query 6 has a granularity mismatch: the ‘Std_Load’ events generated by Ganglia every 15 seconds while the Hadoop generated ‘ReducerStart’ and ‘ReducerEnd’ events are precise to a few microseconds.

To deal with imprecise timestamps in pattern evaluation, recent work [30] proposed a temporal uncertainty model: Assume that the time domain is a sequence of positive integers. The owner of each event stream assigns a time interval to each event to cover all of its possible occurrence times, e.g., \([10,15]\), and a probability distribution over this interval, e.g., a uniform distribution over \([10,15]\). Then the query evaluation system defines a set of possible worlds, where each possible world is a unique combination of the possible occurrence time of each event, and has a probability computed from all the included events. Then in each possible world, each event has a fixed timestamp, one can run the NFA\(^a\) automaton as before, and each match produced in this possible world has the probability of the possible world. Finally, the **confidence value** of a match is the sum of the probabilities of all the possible worlds that produced it.

### 3. DESCRIPTIVE COMPLEXITY

Given the core language, \( \mathcal{L} \), an important question is what features make the language\(^1\) more expressive and at the same time computationally more expensive? To answer this question, we leverage the theory of Descriptive Complexity\(^2\) on a CRCW-PRAM, see [12] for details on descriptive complexity and [25] for more information about complexity.

**Figure 2:** Results on expressibility and complexity of \( \mathcal{L} \).
aggregation from consideration.

Let $L$ (w.o. aggregation) and $NFA^b$ (w.o. aggregation) be the restriction of these two models to have no occurrences of aggregation. In this case we can think of the input alphabet, $\Sigma = D_1 \times \cdots D_n$, as the product of the domains of possible attribute values in the event stream. It is not surprising that without Kleene+ and aggregation we are in $AC^0$, and after adding Kleene+ we are limited to the regular sets:

**Theorem 3.1.** Let $A \subseteq \Sigma^*$. The following conditions are equivalent:

1. $A$ is regular
2. $A$ is recognizable by an $L$ (w.o. aggregation, $\sim$) query
3. $A$ is recognizable by an $L$ (w.o. aggregation) query
4. $A$ is in $cl(NFA^b$ (w.o. aggregation), $\circ, +$)
5. $A$ is in $cl(NFA^b$ (w.o. aggregation), $\circ, +, \sim$)

**Proof Sketch:** Since $L$ (w.o. aggregation, $\sim$) contains single letter alphabets and is closed under concatenation, union, and Kleene+, it contains the regular languages. Obviously, $2 \rightarrow 3 \rightarrow 5$ and $2 \rightarrow 4 \rightarrow 5$. Since an NFA$^b$ (w.o. aggregation) automaton can obviously accept single letter alphabets, $5 \rightarrow 1$.

In the presence of aggregation, we can express non-regular properties, e.g., a simple, strictly contiguous $L$ query can accept exactly the strings over $(a \cup b)^*$ that have more a's than b's. The aggregation operations that we consider are the standard count, min, max, sum, avg, together with any user-defined finite state aggregator. Recall that $S_n$ is the group of permutations of $n$ objects where multiplication is composition. It is known that the word problem for the finite group $S_n$ — given a sequence of elements of $S_n$ is their product the identity — is complete for NC$^1$ [7].

We show in Theorem 3.2 that $L$ with only contiguous queries expresses a rich subset of NC$^1$. First we show,

**Lemma 3.1.** The word problem for $S_5$ — an NC$^1$-complete problem — is expressible in a simple $L$ query of the form strict_contiguity.

**Proof.** The word problem for $S_5$ can be represented as a simple strict-contiguity $a+$ query: we define an aggregate that keeps track of a value, $v$, from 1 to 5 and combines that with the input $\pi$, an element of the fixed, finite alphabet, $S_5$ and computes the next value, $\pi(v)$. The beginning and ending condition of the $L$ query is that $v = 1$.

**Theorem 3.2.** $L$ with only strict contiguity or with only strict and partition contiguity expresses a subset of NC$^1$ that includes complete problems for NC$^1$.

**Proof.** The NC$^1$ completeness comes from Lemma 3.1. For containment in NC$^1$, the $L$ with partition contiguity query can be simulated in NC$^1$ as follows: first replace any input from an event not in the partition by the identity element for the aggregation operation in question. Then do a partial-prefix computation of the aggregation operation.

The ordered graph reachability problem, oREACH, consists of the set of directed graphs on vertices numbered 1 to $n$ such that there is a path from 1 to $n$ and all edges $(i, j)$ are increasing, i.e., $i < j$. It is well known that oREACH is complete for NSPACE[log $n$]. Similarly, oREACH$_a$, the restriction of oREACH in which there is at most one edge from each vertex is complete for DSPACE[log $n$].

It is not hard to see that

**Lemma 3.2.** oREACH$_a$ is expressible in a simple $L$ query of the skip till next match form. Similarly, oREACH is expressible in a simple $L$ query of the skip till any match form.

**Proof.** In both cases the input stream consists of a sequence of edge events with attributes head and tail. A simple $a+$ query checking that $a[i].tail = a[i-1].head$ finds the path. In the deterministic case this is of the skip till next match form because there is at most one edge with a given tail, but in the general case this is a skip till any match case because nondeterminism is involved in finding the right path.

**Theorem 3.3.** $L$ (without skip-till-any-match) expresses a subset of the DSPACE[log $n$] queries including some that are complete for DSPACE[log $n$].

**Proof.** The DSPACE[log $n$] completeness comes from Lemma 3.2. The only subtlety about containment in DSPACE[log $n$] comes with the possible nondeterminism between (Ignore or Take) versus Proceed edges of NFA$^b$. Since there are only a bounded number of places where this nondeterminism can occur in any $L$ query, we remain in logspace by sequentially trying each possible choice. This involves adding a log n-bit counter for each of the states of the NFA$^b$ where such a non-deterministic move could occur.

Finally, for the $L$ language with skip_till_any_match, a theorem in [1] gives the upper bound of its expressive power. It is included below for the sake of completeness:

**Theorem 3.4.** $L$ expresses a subset of NSPACE[log $n$] including some queries that are complete for NSPACE[log $n$].

### 3.3 Expressibility of Other Languages

We also map the expressibility of a wide range of existing pattern query languages to the complexity hierarchy in Fig. 2, with the main results summarized in the left column. **Temporal logic** is equivalent to first-order logic and thus the star-free regular languages on words [14, 19].

**CQL** [5] is a well-known stream language. It maps streams to relations using windows, and applies SQL to compute a result for each window. If the subset of SQL used is limited to selection-join-aggregation, it is first-order logic extended with a counting quantifier, thus equal to ThC$^o$. If the subset of SQL used is relaxed to the bigger set with recursion, its expressiveness and complexity is way up in P-time — this level of complexity is not needed for pattern queries in CEP.

**SQL-TS** [23] provides a stream-processing addition to SQL. Just looking at that stream processing facility, its expressive power — assuming the same set of aggregates as $L$ — is the same as $L$ without negation and restricted to uses of strict or partition contiguity. It thus follows that this stream language is restricted to at most the same subset of NC$^1$.

**Cayuga** [11] is built from an algebraic stream processing language. A least-fixed-point operator is described to express Kleene+. The semantics of simple, i.e., not composed, queries is given via an automaton model similar to NFA$^b$. Its expressive power is the same as $L$ without negation and
Table 2: Notation in runtime complexity analysis.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l$</td>
<td>Number of components in a seq pattern.</td>
</tr>
<tr>
<td>$k$</td>
<td>Number of Kleene closure components in seq.</td>
</tr>
<tr>
<td>$W$</td>
<td>Size of the time window.</td>
</tr>
<tr>
<td>$R$</td>
<td>$R_i$ is the arrival rate of events satisfying the constraints on the $i^{th}$ component of a pattern. A simplifying assumption is: $R_1 = R_2 = \ldots = R_i = R$.</td>
</tr>
<tr>
<td>$U$</td>
<td>Size of the uncertainty interval for events with imprecise timestamps, assumed to be the same for all.</td>
</tr>
<tr>
<td>$c_r$</td>
<td>Average cost for a run, including the cost for run creation, event evaluation, etc.</td>
</tr>
<tr>
<td>$c_m$</td>
<td>Average cost to compute the probability for a (point-based) match in the imprecise case.</td>
</tr>
<tr>
<td>$S_i - S_j$</td>
<td>Event selection strategy of Contiguity, Skip-till-next-match, Skip-till-any-match, respectively.</td>
</tr>
</tbody>
</table>

restricted to skip till next match queries. Thus, Cayuga is contained in the same subset of DSPACE[log n].

XSeq [21] supports pattern queries over hierarchical data with sequencing elements, e.g., in XML. It claims that every “core XSeq” formula can be translated to an equivalent Visibly Pushdown Automata (VPA) and vice-versa. Thus core XSeq can express exactly the languages accepted by VPAs, which was characterized as MSO$_\mu$ in [4]. Monadic second-order (MSO) over words expresses exactly the regular languages. Adding a binary relation $\mu$ that has an edge from each call site (push) to its corresponding return site (pop) gives us MSO$_\mu$ which expresses exactly the visibly pushdown languages. It is strictly between the regular languages and the context free languages.

Summary. The above results give a good picture of the expressibility of sub-languages of $\mathcal{L}$ and other languages, as well as their complexity classes. Our main conclusions are:

1. Understanding pattern languages means understanding the interaction between Kleene+, aggregation, and the event selection strategy. As they are included in successively larger subsets of $\mathcal{L}$, they can be mapped into low-level complexity classes ranging from AC$^0$ to NSPACE[log n].

2. Existing stream pattern languages can be mapped to different levels of the complexity hierarchy. Some of these languages are not able to express all pattern operators and selection strategies. Some others like CQL with recursion are more powerful than what pattern queries in CEP need, hence having to pay a cost for a higher complexity class. After comparing with other languages in the complexity hierarchy, we can see that the core language $\mathcal{L}$ achieves a good balance between expressive power and complexity.

4. RUNTIME COMPLEXITY

We next extend our problem formulation from checking the existence of a query answer to finding all query answers in an input. This analysis, which we call “runtime analysis”, follows the methodology used in the previous section: it shows how the runtime complexity changes as we add more key language features that were shown to lead to different classes in descriptive complexity. The runtime analysis will help us find intuitions for optimization later.

Preliminaries. The runtime cost is mainly reflected by the number of simultaneous runs of an NFA$^\delta$ automaton. A run represents a unique partial match of the pattern. It is either initiated when a new event is encountered to match the first component of the pattern, or cloned from an existing run due to nondeterminism in the strategy of skip to any match. A run is terminated when it forms a complete match or expires before reaching a complete match. The symbols used in the analysis are listed in Table 2.

Figure 3 illustrates the possible runs for different selection strategies. Figure 3(a) shows a simple pattern with Kleene+. In Figure 3(b), the first two rows include the id and timestamp of 5 events for a sample stream. In the id, the letter specifies the satisfied pattern component, and the number is used to distinguish from events of the same type. The lower part of Figure 3 shows the possible runs for different selection strategies. Each row represents a possible run: in the the cell under each event, an arrow means the event is selected for this run, while a dotted line means the run skipped this event. In the “Result” column, a circle means that the run is terminated before it reaches the final state, while a black dot means that the run reaches the final state to make a match. For $S_1$, there are only 2 possible runs, and only the second ($\#2$) reaches the final state and generates a match. The first run ($\#1$) terminates immediately when the next event does not satisfy the pattern. For $S_2$, two matches are returned. $\#4$ is the same as $\#2$ in $S_1$. The $\#3$ match, which is $(a_1,b_1,b_2,c_1)$ skips $a_2$ during pattern matching because it is irrelevant after the run selects $a_1$. For $S_3$, obviously the number of runs is many more than the other two strategies. There are 14 runs triggered in total, and 6 of them generate matches. All possible runs are triggered in this case. This illustration is drawn to provide some sense of the number of possible matches before our analysis.
The relations of the five cases are summarized in Table 3.

**Base case.** Consider a simple pattern without Kleene+, evaluated under $S_1$ or $S_2$. The runtime complexity for $S_1$ and $S_2$ are the same in number of runs. (In practice, the cost for $S_2$ may be higher because these runs can produce longer matches.) Here the only trigger to generate a new run is an event qualified for the first component of the pattern. So the total number of runs is exactly the same as the number of events matching the first component, i.e., $RW$. After multiplying the cost $c_r$, we get the runtime cost.

**Skip-till-any-match.** Then consider a pattern without Kleene+, evaluated under $S_3$. $S_3$ is chosen to capture all event sequences that match the pattern, ignoring irrelevant events in between. Given a pattern of $l$ components, each component can have $RW$ matching events in the time window, so there can be $(RW)^l$ matches. At runtime we need at least this number of runs: some runs lead to complete matches, while others are intermediate runs that fail to complete. It is not hard to show that considering all, the number of runs is $(RW)^{l+1} / (RW - 1)$, hence polynomial in $W$.

**Kleene Closure.** Next consider a Kleene+ pattern evaluated under $S_3$. Under $S_3$, any combination of the $RW$ events for a Kleene+ component can potentially lead to a match, hence requiring a run. So the cost is exponential, $2^{RW}$. Even worse, $k$ Kleene+ components will make the factor $2^{kRW}$. As a result, the total number of runs would be $(RW)^{l+1} / (RW - 1)^k$ times $2^{kRW}$, exponential in $W$.

**Imprecise Timestamps.** Finally consider all patterns in $L$ in the presence of imprecise timestamps. Recent work [30] proposed a solution for simple pattern queries like $\text{SEQ}(A, B, C)$, where input events all carry an uncertainty interval to represent possible occurrence times. The algorithm employs (1) an efficient online sorting method that presents events in the current time window in “query order”; that is, in the current window ‘a’ events are presented before ‘b’ events which are before ‘c’ events; (2) after sorting, an efficient method to check the temporal order of events for a simple pattern, without enumerating all possible worlds.

Our work aims to further support Kleene+ patterns like Query 6 on events with imprecise timestamps. Take Query 6 and the sequence of events with values, (0.1, 0.2, 0.15, 0.19, 0.25). The goal is to look for a series of events that have increasing timestamps and non-decreasing values. Since each event has an uncertainty time interval, finding a series of events with increasing timestamps cannot be restricted to the order of events in the input sequence. Instead, we can (1) apply the sorting method in [30] to re-arrange the events in a time window by query order, in this case that is, arranging the events by order of non-decreasing values; (2) enumerate every subset of this sorted sequence using skip till any match strategy; and (3) check temporal order of each subset of events using the method in [30]. More details of the algorithm are shown in Appendix A. In summary, the solution to evaluating Kleene+ pattern queries on events with imprecise timestamps can be constructed based on the known algorithm for evaluating simple pattern queries [30], but always has to use $S_3$ to avoid missed results.

In addition, there is an extra cost caused by imprecise timestamps, confidence computation in the match construction process. Assume that a matching algorithm, as sketched above, has returned a sequence of events, $(e_{i_1}, e_{i_2}, \ldots, e_{i_m})$ where each has an uncertainty internal, as a potential match. The model for imprecise timestamps, described in §2.3, requires computing the confidence of this sequence being a true match and comparing it with a threshold. To do so, the confidence is computed based on timestamp enumeration: pick one possible point timestamp for each event from its uncertainty interval, validate whether the point timestamps of the $m$ events satisfy the desired sequence order, and if so, compute the probability for this point match. After enumerating all instances, sum the probabilities of all validated instances as confidence. So without Kleene+, the total cost is, $\left(\frac{(RW)^{l+1}}{RW-1}\right)(c_r + t^l \times c_m)$, where the first factor is the number of runs and the second is the time cost per run.

For queries with Kleene+ components, there are two different cases. The simpler case is that events can satisfy a Kleene+ independently, which is called the uncorrelated case. In the correlated case, events collect by a Kleene+ must satisfy an ordering constraint, e.g., increasing in time and non-decreasing in ‘LoadStd’ value for Q6 in Table 1. In this case, let the set of events collected by each Kleene+ be $RW$. They have to participate in the enumeration process in confidence computation. So the total cost for $k$ Kleene+ components is given by the number of runs, $\left(\frac{(RW)^{l+k+1}}{RW-1}\right) \times 2^{kRW}$, times the cost per run, $(c_r + U^{l-k+WR} \times c_m)$.

**Summary.** The main results of our runtime analysis include: (i) Pattern queries that use Kleene+ under skip till any match, is subject to an exponential cost in the window size; (ii) The solution to evaluating Kleene+ pattern queries on events with imprecise timestamps can be constructed based on a known algorithm for evaluating simple pattern queries, but always has to use $S_3$ to avoid missed results. It also includes an additional cost of confidence computation for each pattern match, which is also exponential in the worst case.

As such, two bottlenecks in pattern query processing are (1) Kleene+ evaluated under $S_3$ and (2) confidence computation under imprecise timestamps. We focus on the two bottlenecks in optimization. In particular, optimizing Kleene+ under $S_3$ not only expedites such queries, but also enables the evaluation of all queries with imprecise timestamps.

<table>
<thead>
<tr>
<th>#</th>
<th>Language Features</th>
<th>Selection Strategy</th>
<th>Timestamp Class in W</th>
<th>Complexity in W</th>
<th>Formula (using notation in Table 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$L$ w.o. Kleene+</td>
<td>S1/S2</td>
<td>Precise</td>
<td>Linear</td>
<td>$RW \times c_r$</td>
</tr>
<tr>
<td>2</td>
<td>$L$ w.o. Kleene+</td>
<td>S3</td>
<td>Precise</td>
<td>Polynomial</td>
<td>$\left(\frac{(RW)^{l+1}}{RW-1}\right) \times c_r$</td>
</tr>
<tr>
<td>3</td>
<td>$L$ w. Kleene+</td>
<td>S3</td>
<td>Precise</td>
<td>Exponential</td>
<td>$\left(\frac{(RW)^{l+k+1}}{RW-1}\right) \times 2^{kRW} \times c_r$</td>
</tr>
<tr>
<td>4</td>
<td>$L$ w. Kleene+, uncorrelated</td>
<td>S1/S2/S3</td>
<td>Imprecise</td>
<td>Exponential</td>
<td>$\left(\frac{(RW)^{l+k+1}}{RW-1}\right) \times 2^{kRW} \times (c_r + U^{l-k} \times c_m)$</td>
</tr>
<tr>
<td>5</td>
<td>$L$ w. Kleene+, correlated</td>
<td>S1/S2/S3</td>
<td>Imprecise</td>
<td>Exponential</td>
<td>$\left(\frac{(RW)^{l+k+1}}{RW-1}\right) \times 2^{kRW} \times (c_r + U^{l-k+WR} \times c_m)$</td>
</tr>
</tbody>
</table>

Table 3: Main results of runtime complexity analysis.
5. OPTIMIZATIONS

Our key insight for optimization is derived from the observed difference between the low-level complexity classes in descriptive complexity analysis, which considers only one match, and exponential complexity in runtime analysis, which considers all matches. Our idea is to break query evaluation into two parts: pattern matching, which can be shared across matches, and result construction, which constructs individual results. We propose several optimizations to reduce shared pattern matching cost (§5.1 and §5.2).

To address the overhead in confidence computation, we provide a dynamic programming algorithm to expedite the computation and enable improved performance when the user increases the confidence threshold to filter matches (§5.3).

5.1 Sharing with Postponed Operations

Let us consider the evaluation of a Kleene+ pattern under $S_3$. For ease of composition, we use a simplified version of Query 6, shown in Fig. 4(a), and a small event stream in Fig. 4(b). Each event is labeled with a letter specifying the pattern component satisfied, and the number for distinguishing it from other events of the same type. The events are also listed with contained attributes. The NFA$^b$ model for this pattern is in Fig. 4(c). An initial set of operations according to the NFA$^b$ execution model are shown in Fig. 4(d). In the diagram, each box shows an operation in NFA$^b$ execution (the upper part) and the run after this operation (the lower part). We call such a diagram a “pattern execution plan”.

To better explain it, we introduce the primitive operations based on the NFA$^b$ model:

- **Edge evaluation** evaluates the condition on taking the transition marked by the edge, where the condition is compiled from the event type, time window constraint, and value predicates—this can be broadly considered a “predicate evaluation” step.
- **Run initialization** is used to start a new run.
- **Run extension** adds a new event to an existing run.
- **Run cloning** duplicates an existing run to enable nondeterministic actions.
- **Run proceeding** moves to the next automaton state without consuming events.
- **Run termination** terminus a run when it arrives at the final state or it fails to find any possible transition.

Then a pattern execution plan $\Gamma$ is a tree of primitive operations, where each unique path in the tree is a run ($\rho$) of the NFA$^b$. Next we state some key properties of this execution plan, which enable later optimization.

First, we observe that $S_3$ allows edge (predicate) evaluation operations to be postponed until later in the execution plan, which is a special type of “commutativity” allowed in the NFA$^b$ model. For instance, consider the evaluation of the ‘[’ edge in the NFA$^b$ in Fig. 4(c), where Kleene+ is trying to select more ‘b’ events. Let $e$ denote the current event. The predicates in this edge evaluation are: $e.type = B \land e.time < W \land e.val \geq b[i - 1].val$. If we postpone the value predicate, $e.val \geq b[i - 1].val$, until the end of the plan, it is not hard to show that the plan still produces the same matches as before.

Second, we observe that $S_3$ also allows some suffix paths of the plan to be postponed altogether. To explain that, we introduce the concept of “consecutive operations”: Some of the primitive operations in the plan have to be performed consecutively. In Fig. 4(d), after step 1 is executed, step 2 must be performed immediately; otherwise this run will not be initialized and the following $b_1$ will not be evaluated properly. We call such a pair of operations as consecutive operations (denoted by “$\leftrightarrow$”), meaning that other operations are not allowed between the two operations.

In contrast, there are operations that do not need to be performed consecutively. This happens when a run is cloned in $S_3$. In Fig. 4(d), after step 3 finishes, due to the nondeterminism two actions are triggered: step 4 extends the current run with a new event, which needs to be performed right after step 3. In contrast, step 5 clones the current run to a new independent run for further processing, and thus even if it is not performed immediately, it will not affect the other run. We call a pair of primitive operations like steps 3 and 5 “non-consecutive operations” (denoted by $\rightarrow$), e.g., $3 \rightarrow 5$, $6 \rightarrow 9$ and $7 \rightarrow 11$ in Fig. 4(d). In the plan $\Gamma$, all the pairs of non-consecutive operations allow us to decompose some suffix paths from the main path (which is highlighted in green in Fig. 4(d)). We denote the main path as $\rho_1$, and each suffix path as $\rho_3 = \rho_1 + \Delta \rho$, with some $1 \leq i < j$. The observations above lead to two propositions key to our optimization.

**Proposition 5.1.** Given a pattern execution plan $\Gamma$ evaluated under $S_3$, if the run corresponding to the main path $\rho_1$ is evaluated with value predicates removed, and if it produces an intermediate match, $M = (e_1, e_2, \ldots, e_m)$, then $M$ is a superset of every match that can be produced by $\Gamma$.

**Proof.** Since the evaluation of value predicates is removed, all events of the (1)satisfied event type (2)during the period defined by the first and last event of $\Gamma$ will be selected to $\mathcal{M}$. Any match $m$ produced by $\Gamma$ satisfies (1) event type requirement and (2)time window constraint, thus any event of $m$ will be included by $\mathcal{M}$. \qed

**Proposition 5.2.** Given a pattern execution plan $\Gamma$ evaluated under $S_3$, the complete set of matches produced by $\Gamma$ is the same as first obtaining the intermediate match $\mathcal{M}$ by running the main path $\rho_1$ with value predicates postponed, and then enumerating all subsets of $\mathcal{M}$ while evaluating the postponed predicates.

**Proof.** In the enumeration part, the system will perform postponed predicate evaluations over each enumeration instance, which is a subset of events in $\mathcal{M}$ in their temporal order.

According to Proposition 5.1, $\mathcal{M}$ is the superset of every match of $\Gamma$. So for any match $m$ produced by $\Gamma$, there is one enumeration instance containing the same events as $m$. After evaluating value predicates on this enumeration instance, $m$ will be generated.

For any enumeration instance, if it passes the predicate evaluation, it get a match $m'$. Since $\Gamma$ is running under $S_3$, the events of $m'$ will be matched together.

In summary, enumerating all subsets of $\mathcal{M}$ generates the same results as $\Gamma$ under $S_3$. \qed

**Postponing Algorithm.** We now present the postponing algorithm that breaks the evaluation according of a plan $\Gamma$ into two parts: pattern matching, which is shared by all of the original runs of $\Gamma$, and result construction.
1. **Pattern matching.** It follows directly from Proposition 5.1: we take the main run $\rho_1$ and run it with all value predicates removed. This is the only cost incurred.

2. **Result construction.** This step follows directly from Proposition 5.2: We take the match $M$ produced by the main run $\rho_1$ with value predicates postponed. Then we enumerate all subsets of $M$ while applying the postponed predicates, and return all the matches produced in this process. A simple optimization can be added to the enumeration process, e.g., ensuring that there is at least one event matching each pattern component in order to be a match.

Fig. 4(e) illustrates the postponing algorithm. Using the original plan, there are 15 runs. Using the postponing algorithm, there is only 1 run in pattern matching, producing an intermediate match $M = (a_1, b_1, b_2, b_3, c_1)$, and 7 cases in enumeration, leading to 7 final matches (in bold). Note that the benefits of the postponing algorithm are usually more than illustrated in this simple example: First, it can filter non-viable runs effectively. For example, a run that collects a large number of events for a Kleene+ component without finding an event for the last component is completely avoided in the postponing algorithm. Second, many fewer runs also mean the reduced evaluation cost for each event. Third, when a run reaches the result construction phase, the enumeration cost is still cheaper than the cost of cloning runs on the fly and repeated operations like edge evaluation on the same event can be carefully shared.

5.2 Postponing with Early Filters

A main drawback of the postponing algorithm is that the pattern matching phase removes value predicates and hence loses the ability to prune many irrelevant events early. To improve the filtering power, we would like to improve the postponing algorithm by performing edge evaluation, including the value predicates, on the fly as events arrive. However, it is incorrect to simply evaluate all predicates on the fly because it may not produce an intermediate match $M$ that is a superset of every final match. Consider a Kleene+ on a sequence of values, $(0.1, 0.2, 0.15, 0.19, 0.25)$, and two correct results for non-decreasing subsequences, $(0.1, 0.2, 0.25)$ and $(0.1, 0.15, 0.19, 0.25)$. If we evaluate the value predicate $b[i].val \geq b[i-1].val$, in the main run $\rho_1$ as events are scanned, we can produce an intermediate match $M = (0.1, 0.2, 0.25)$, which is not a superset of $(0.1, 0.15, 0.19, 0.25)$.

Therefore, the decision on whether to evaluate a predicate on the fly should be based on its correctness, which is related to the consistency of evaluation results on a power set of an event sequence. Regarding consistency, we observe that all predicates applied to Kleene+ fall into one of four categories:

- **True-value consistent predicates.** A predicate in this category satisfies the following property: if the predicate evaluation result of comparing the current event, $e$, against all selected events is true, then it is always true when comparing the event $e$ against any subset of the selected events. Consider $b[i].val > max(b[i], \ldots )$.val for Pattern(a, b+, c). If $e.val$ is larger than the maximum of all selected events for the Kleene+, it will be larger than the maximum of any subset. So the “true” value is consistent. If an event fails the check, it is still possible to be larger than the maximum value of some subsets. So events validated by true-value consistent predicates on the fly do not need to be checked again in result construction; they can be labeled as “SAFE” to avoid redundant evaluation. Other events cannot be discarded and should be labeled as “UNSAFE” for additional evaluation in result construction.

- **False-value consistent predicates.** The property for this category is: if the predicate evaluation result of comparing $e$ against all selected events is false, then it is always false for comparing $e$ against any subset of the selected events. $c.val < max(b[i], \ldots )$ val for Pattern(a, b+, c) is an example. Events evaluated to false by such predicates can be discarded immediately because they will never qualify. Other events must be kept for additional checking in result construction.

- **True and false-value consistent predicates** are predicates that are both true-value and false-value consistent predicates. An example is $b[i].val > 5$ for Pattern(a, b+, c). Since it does not compare $b[i]$ with any of the selected events by Kleene+, the evaluation result will never vary with the subset of the events chosen. Events evaluated to true by true-false consistent predicates can be labeled as “SAFE”,

![Figure 4: A running example for the postponing algorithm.](source_url)
and those evaluated to false can be discarded immediately. For this kind of predicates, we can output the generated intermediate matches as collapsed format, which collect all satisfied events and are superset of final matches. The collapsed format provides a compact way of results before enumerating every detailed match, and the user may opt to pay the enumeration cost only when needed.

Inconsistent predicates are predicates that are neither true-value consistent or false-value consistent. An example is $b[i].val > \text{avg}(b[i+1..i].val)$ for Pattern((a, b+, c)). This type of predicates should be postponed until result construction.

With the knowledge of the four categories, the postponing algorithm can make a judicious decision on whether to perform predicate evaluation on the fly to filter events early.

5.3 Optimization on Confidence Computation

As mentioned in §4, there is an extra cost to compute the confidence of each pattern match in the presence of imprecise timestamps. This operation is prohibitively expensive for queries with Kleene closure, because the cost is exponential in the number of selected events. So we optimize it in this section. Our main idea is that existing work [30] finds all possible matches with confidence greater than zero. However, matches with low confidence are not of little interest to the user. Setting a confidence threshold to prune such matches is of more value to the user, and it provides opportunities for optimization. The confidence of a partial match is non-increasing as more events are added to extend a partial match. In result construction, we can begin the enumeration with shorter runs (matches), and add events to validated matches one by one. If a shorter match has confidence lower than a threshold, then all longer matches with the same prefix will not need to be considered again.

Dynamic programming optimization. Based on the above intuition, a dynamic programming method is designed to optimize the performance of confidence computation. The pseudocode is in Algorithm 1. The input is $\tau$ which already collects a set of events $e[0..n]$ for a Kleene+ component. We first initialize a list $S$ to hold all sub-list $L_i$, where $L_i$ holds all enumerations starting with $e[i]$. These are done in Lines 1-4. Then we start with a new enumeration (newEnum) with only $e[i]$ selected for the Kleene+ component (Line 4). If newEnum passes the confidence threshold check and predicate check, it will be added to $L_i$ as one of the matches. Then inside the loop between Lines 7 to 14, it tries to extend every stored match in $L_i$ with a new event $e_j$, and valid enumerations will be added to $L_i$. Any invalid enumerations will be ignored as there is no chance for them to be a prefix of future enumerations. Finally, $S$ will be the whole qualified matches with confidence higher than $c$.

6. EVALUATION

In this section, we evaluate our new system, called SASE++, with the proposed optimizations, and compare it with several state-of-the-art pattern evaluation systems. Our evaluation uses both microbenchmarks with controlled properties and a detailed case study in Hadoop cluster monitoring.

6.1 Microbenchmarks

Queries in the microbenchmarks use the template, seq(A a, B+, b ], C c), unless stated otherwise, and $S_{16}$. We vary two parameters: The selectivity ($\theta$) defined as, 

$$\frac{\# \text{Matches}}{\# \text{Events}}$$

is controlled by changing the value predicates in the pattern.

```matlab
Algorithm 1 Dynamic programming optimization
Input: A run $\tau$, whose buffers have been all filled, events for its Kleene closure component are $e[0..n]$; the confidence threshold $c$
Output: All enumerations that will make a match with confidence higher than $c$
1: Initialize an array of enumeration space (List) $S$
2: for $i = 0$ to $n$ do 3: Add new initialized (List)$L_i$ to $S$
4: Initialize new ENUM newEnum = {$e_i$}
5: if Confidence(newEnum) $> c$ then 6: Add newEnum to $L_i$
7: for $j = i + 1$ to $n$ do 8: for each ENUM in $L_i$, enum do
9: if Confidence(enum $\cup e_j$) $> c$ then
10: Initialize newEnum2 = (enum $\cup e_j$)
11: Add newEnum2 to $L_i$
12: end if
13: end for
14: end for
15: end if
16: end for
17: return $S$
```

It is varied from $10^{-6}$, which is close to the real selectivity in our case study, up to 1.6, which is a very heavy workload to test our optimizations. The other parameter is the time window ($W$), varied from 25 to $10^5$. Our event generator creates synthetic streams where each event contains a set of attributes with pre-defined value ranges, and a timestamp assigned by an incremental counter or an uncertainty interval if the timestamp is imprecise. We use 0.1 million events when varying $\theta$, and 100 million events when varying $W$.

We run SASE++ with the following settings: (1) Postponing, which applies postponing ([5.1] only; (2) On-the-fly, which applies early filters ([5.2] based on postponing; (3) Collapsed, which returns results in collapsed format based on on-the-fly; (4) DP $x$ it applies dynamic programming ([5.3]) with $x%$ as the threshold based on on-the-fly. In addition to running SASE++ with different optimization settings, we also compare it with (5) ZStream [20], which applies the optimization of placing a buffer of events at each NFA state and triggers pattern evaluation only when all the buffers become non-empty; (6) SASE+ [1, 28], which strictly follows the execution of the NFA model, and (7) XSeq [21], which we describe in detail shortly. Table 4 lists all the algorithms and systems compared in our study.

All experiment results were obtained on a server with an Intel Xeon Quad-core 2.3GHZ CPU and 8GB memory. System SASE++ runs on Java HotSpot 64-Bit Server VM.

6.2 Evaluation with Precise Timestamps

We first evaluate the two optimizations, postponing ([5.1] and on-the-fly ([5.2]), using streams with precise timestamps.
Throughput. Figure 5(a) and 5(c) show the throughput while varying \( \theta \) and \( W \) for the true-value consistent predicates. The y-axis is in logarithmic scale. We see that the throughput of SASE+ drops very fast as \( \theta \) and \( W \) increase. ZStream’s performance degrades similar to SASE+. Our postponing algorithm works well; its performance goes down only slightly. On-the-fly has a similar performance as postponing in this workload. Figure 5(b) and 5(d) show the number of runs created with varied \( \theta \) and \( W \). The plots confirm our runtime analysis that the numbers of runs in SASE+ and ZStream can go up exponentially and thus their throughput drops quickly. On the contrary, the number of runs in postponing algorithm increases much more gradually.

We further show the throughput when varying \( \theta \) and \( W \) for the false-value consistent predicates in Figure 5(e) and 5(f). Here, on-the-fly performs better than postponing because it can discard more events earlier when evaluating them on the fly. Results for the other types of predicates are omitted because they exhibit similar trends as shown in these plots.

Cost breakdown. We further break down the cost of each system by profiling time spent on each operation. The breakdown of SASE+ is shown in Figure 5(g). The run initialization cost stays the same because only events qualified for the first component trigger this operation and the same stream is used. The rest of the cost is attributed to pattern matching, which is exponential in \( W \) and becomes dominant as the time window increases. The cost breakdown for ZStream is shown in Figure 5(h). The additional cost compared to SASE+ is the buffering cost, which is also constant as the stream stays the same. With the filtering power of the buffering, the cost for run initialization and pattern matching is smaller than that of SASE+. The cost breakdown for the postponing algorithm with predicate evaluation on-the-fly is shown in Figure 5(i). Using the run initialization as a reference, the cost for pattern matching stays low all the time. The cost for result construction increases because runs tend to collect more events as \( W \) increases. However, it is still lower than the run initialization cost for most time.

Summary. Overall the postponing algorithm can pro-
provide up to 2 orders of magnitude improvement (max. 383x) over SASE+ and ZStream. The pattern matching phase can reduce the cost from exponential to polynomial, and sometimes close-to-linear cost. Although the result construction phase may still generate an exponential number of matches, which are determined by the query, the cost is much smaller than SASE+ and ZStream, and returning them in a collapsed format is an option for further reduction of the cost.

### 6.3 Evaluation with Imprecise Timestamps

In this set of experiments, we generate streams where each event has an uncertainty interval size of 10. Fig. 6(a) shows the throughput for varying $W$ with true-false value consistent predicates. The postponing algorithm without dynamic programming optimization is dominated by the cost of confidence computation, which is highly sensitive to $W$. It fails to run when $W > 3000$, which is too small for practical uses. The dynamic programming (DP) optimization can support larger windows and improve performance as the confidence threshold increases. The collapsed format returns results in a compact way, without enumerating all the matches, hence setting the upper bound of performance. The cost on confidence computation for different algorithms is as shown in Fig. 6(d). Note that the DP method is based on the postponing algorithm; without the intermediate matches, such optimization on confidence computation is not feasible.

Figure 6(b) shows the cost breakdown of the SASE+ algorithm, while Figure 6(c) show the cost breakdown of the dynamic programming algorithm with 0.999 as the confidence threshold. We could see confidence computation cost is dominant in both figures. Please note that the value range of $X$-axis in Figure 6(c) is much wider than that in Figure 6(b), this is because the SASE+ is unable to support larger window, while the dynamic programming makes it feasible for windows which is more than one order of magnitude larger.

### 6.4 Comparison with XSeq

We further compare the performance of our system with XSeq, an engine for high-performance complex event processing over hierarchical data like XML, JSON etc. (which won the best paper award at SIGMOD 2012). For comparison, the same synthetic stream is used, and it is converted to the SAX format required by XSeq. Since we use $S_3$, XSeq is set to the All_Match_Skip_One mode, which finds all possible matches for each starting point. The optimization method of XSeq is set to VP_OPS_OPTIMIZATION, which gives the best performance.

We first vary the query length $l$ for seq($A_1, \ldots, A_l$). The result is in Fig. 6(e). A line marked by “XSeq $n$” means that the input includes $n$ events. XSeq is sensitive to the input size so it does not scale well, while our system is stable with the input size and its throughput is about four to ten magnitudes higher. Then we compare to XSeq by varying time window $W$ for the usual Kleene+ pattern, which is shown in Fig. 6(f). The throughput of XSeq is still much lower and sensitive to the input size. We observe the performance of XSeq is always low and not affected by $W$.

A main observation is that XSeq is not optimized for the time window. From the observation of its output, we learn that XSeq treats the timestamp as a general attribute and misses some necessary optimizations. For example, if the query is, seq($a, b$) within 25, XSeq will compare every $a$ with every $b$ in the input, instead of terminating when no future events can fall into the time window. This can be a straightforward optimization but we were given only a binary executable of XSeq without the source code. Second, XSeq is optimized for different selection strategies. Among 13 sample queries with Kleene closure in the paper [21], 5 queries are applied to children of nodes, the depth of which can be limited; the other 8 queries are applied to on immediate following siblings, and this is like $S_1$. XSeq lacks optimizations for more flexible selection strategies, $S_2$ or $S_3$.

Overall, XSeq is not optimized for the ability to “skip” events, which is one of the core features of CEP. It is largely due to the fact that XSeq is designed for processing hierarchical data instead of general event streams.

### 6.5 Case study: Hadoop Cluster Monitoring

As stated in recent work [22], Hadoop cluster monitoring is still in its adolescence. By working with Hadoop experts, we perform a detailed case study to demonstrate that our system can help automate cluster monitoring using declarative pattern queries and provide real-time performance.

#### Data collection

We collect two types of logs in real-time: the logs of system metrics, e.g., CPU usage, network activity, etc, and the logs of Hadoop jobs, e.g., when a job starts and ends. Ganglia [18], a popular distributed system monitor tool, is used as the core part of our real-time data collection system as shown in Fig. 8. In Ganglia, gmond is the monitoring daemon installed on every node. We use gmond to grab performance metrics from the OS, and we also customize it to parse Hadoop logs. Then gmond records the collected data, and broadcasts the records to the gmond on peer nodes. This way, each node has all metrics for the cluster. Gmeta is the polling daemon, which runs on the machine where our system, SASE++, is running. Gmeta connects to one node of the cluster, polls the data, and saves to round-robin databases (RRD). Then our system can read the data for pattern evaluation.

#### Queries

We develop 6 queries together with Hadoop experts to analyze Hadoop performance. They all use Kleene+ patterns and some use uncertainty intervals. As Q1 and Q6 are already shown in Table 1, we discuss other queries below, which are listed in Table 5.

Recall that Q1 computes the statistics of lifetime of mappers in Hadoop. Similarly, Q2 does it for reducers. Fig. 7(a) and Fig. 7(b) show the average lifetime of mappers and reducers for three different workloads in Table 6. Twitter, which counts statistics for tri-grams from tweets; Worldcup U, analyzes the frequent users from the logs for clicks on 1998 FIFA Worldcup website; Worldcup S, divides user clicks into sessions. In Fig. 7(a), the Twitter job has much longer running time than the other two workloads because
Table 5: Other pattern queries for Hadoop monitoring.

<table>
<thead>
<tr>
<th>Q</th>
<th>Pattern Query</th>
<th>Workload</th>
<th>Raw data (GB)</th>
<th>Map output (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Pattern seq(JobStart a, Mapper b</td>
<td>JobEnd c)</td>
<td>Twitter</td>
<td>53.5</td>
</tr>
<tr>
<td>Q2</td>
<td>Pattern seq(JobStart a, Reducer b</td>
<td>JobEnd c)</td>
<td>Worldcup U</td>
<td>252.9</td>
</tr>
<tr>
<td>Q3</td>
<td>Pattern seq(ReducerStart a, DataPull b</td>
<td>)</td>
<td>Worldcup S</td>
<td>252.9</td>
</tr>
</tbody>
</table>

Table 6: Hadoop workload statistics

As mappers output intermediate results, reducers may not consume them immediately, which leads to data queuing. The data queuing in the lifetime of Twitter workload is shown in Fig. 7(c). The first peak implies that most mappers have completed their tasks; then the queuing size starts to reduce as data is consumed by reducers. Fig. 7(d) is the queuing size for the Worldcup U workload, which is different. The job has not really started until 2300 seconds passed. This is because concurrent jobs are running and it has to wait. Our Hadoop experts find these results very helpful.

Q5 and Q6 are used to find tasks that cause cluster imbalance. As Q6 is described above, we simply note that they provide some public datasets, where the data rate is 0.758 events/second/node in the busiest month, and even lower in other months. Fig. 7(f) compares the optimization algorithms for Q5 and Q6. It shows the effectiveness of the optimizations.

7. RELATED WORK

CEP languages. In §3 we analyzed and compared a
large number of CEP languages with different descriptive complexity. Other languages such as TESLA [10] do not introduce new features beyond those surveyed languages.

**Temporal models.** The discussion for CEP over streams with imprecise timestamps is based on the uncertain temporal model in [30]. Other temporal models [3, 2, 6, 11] use time intervals to represent precise event durations, instead of uncertain occurrence time, and hence do not address uncertainty in pattern matching and related complexity.

**Optimizing CEP performance.** Improving the performance of CEP queries has been a focus of many works. Recent studies make use of sharing among similar queries to reduce cost [15, 29]; optimize for performance given out of order streams [13]; optimize the performance of nested pattern queries by pushing negation into inner subexpressions [16]; and rewrite queries in a more efficient form before translating them into automata [24]. In distributed systems, the work [2] applies plan-based techniques to minimize event transmission costs and efficiently perform CEP queries across distributed event sources.

8. CONCLUSIONS

This paper presented theoretical results on expressive power and computation complexity of pattern query languages in CEP. These results offer insights for developing three optimization techniques. Comparison with existing systems shows the efficiency and effectiveness of a new system using these optimizations. A thorough case study on Hadoop cluster monitoring also demonstrates its practical value. In future work, we will consider ways to integrate with approximate pattern matching when events carry uncertain values and evaluate our techniques in large scale CEP systems.

9. REFERENCES


APPENDIX

A. ALGORITHM FOR IMPRECISE TIMES-TAMPS

Here we show the details to construct the algorithm to evaluate stream with imprecise timestamps. The new algorithm is constructed from a base algorithm for evaluating simple patterns without Kleene+ over streams with imprecise timestamps[30]. We use the event-based framework with sorting for query order evaluation as the base algorithm. Following are the changes we made to adapt it for queries with Kleene+.

The first change is sorting events in the preprocessing part. Similar as the base algorithm, for events overlap in time, they are sorted by their satisfied component order in the query. If events overlap in time are both for the same Kleene+ component, they are ordered to satisfy the value predicates for the component.

The second change is in the pattern matching part. The base algorithm only deals with NFA\(^{b}\) models for simple patterns, now we change it to adapt for states representing Kleene+ such that it can select finite yet unbounded events. Other settings stay the same: it is always running under skip till any match strategy; checking temporal order by shrinking the uncertain intervals of selected events.

Another change is in the match confidence part. For correlated case, it is the same method: enumerate all possible point matches, compute the probability of each match, then sum them up to get the confidence. For uncorrelated case, it is easier because the order of events for Kleene+ does not matter. So events selected for Kleene+ needs to participate in enumeration only if their uncertainty intervals overlap with events for other components. Otherwise they do not need to participate in enumeration. After get the probability from enumeration, we only need to multiply it by the probability of events which do not participate in enumeration one by one.

In summary, we construct the solution to evaluating Kleene+ pattern queries on events with imprecise timestamps based on the known algorithm for evaluating simple pattern queries [30], always using \(S_1\).