# HIGH-PERFORMANCE COMPLEX EVENT PROCESSING FOR DECISION ANALYTICS

Haopeng Zhang

# HIGH-PERFORMANCE COMPLEX EVENT PROCESSING FOR DECISION ANALYTICS

A Dissertation Presented

by

HAOPENG ZHANG

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2017

College of Information and Computer Sciences

# HIGH-PERFORMANCE COMPLEX EVENT PROCESSING FOR DECISION ANALYTICS

A Dissertation Presented

by

HAOPENG ZHANG

Approved as to style and content by:

Yanlei Diao, Chair

Neil Immerman, Member

Alexandra Meliou, Member

Ana Muriel, Member

James Allan, Chair of the Faculty
College of Information and Computer Sciences

# ACKNOWLEDGMENTS

I would like to express my deepest thanks to my PhD advisor, Prof. Yanlei Diao. She helps me so much during my years at UMass. I learnt from her not only kills to conduct research, but also ways of scientific thinking, insistence on quality of work and positive attitude towards difficulties. She is so nice to forgive my stupid mistakes and offers help generously. I feel so lucky and proud to have her as my advisor.

I am also grateful to Prof.Neil Immerman, who made significant contribution in our collaboration and joins my committee. Neil is so nice and patient when we discussed problems which were too difficult for me to understand quickly, I really appreciate his help and patience.

I am grateful to Prof. Alexandra Meliou, who collaborates with me on my last piece of work intensely and joins my committee member. Her expertise helps me so much on an unfamiliar area. She offers detailed explanations and gives references for concepts, which helps me so much. I enjoying working with her.

I am grateful to Prof. Ana Muriel for being my committee member and for her valuable comments and suggestions in our discussion. She is so responsive and kind in our communication and willing to offer favor when I scheduled my defense. I really appreciate her time and kindly help.

I would like to thank my colleagues and friends at UMass. I am grateful to Prof. Gerome Miklau for input and discussions about many aspects of database research. I am grateful to Boduo Li, Liping Peng, Wentian Lu, Yue Wang, Chao Li and Bo Jiang for active discussion with me on my research problems. I thank Rachel Lavery and Leeanne Leclerc for making my PhD study much easier by taking care of administrative issues.

I am deeply obliged to my parents who are always supportive whenever I was facing difficulties. Although I have not been home for a few years, I can always feel their love, which has encouraged me to keep moving forward. I love my parents so much.

# ABSTRACT

## HIGH-PERFORMANCE COMPLEX EVENT PROCESSING FOR DECISION ANALYTICS

MAY 2017

HAOPENG ZHANG

B.Sc., BEIHANG UNIVERSITY, BEIJING, CHINA

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Yanlei Diao

Complex Event Processing (CEP) systems are becoming increasingly popular in domains for decision analytics such as financial services, transportation, cluster monitoring, supply chain management, business process management, and health care. These systems create or collect high volumes event streams, and often require such event streams to be processed in real-time. To this end, CEP queries are applied for filtering, correlation, aggregation, and transformation, to derive high-level, actionable information. Tasks for CEP systems fall into two categories: passive monitoring and proactive monitoring. For passive monitoring, users know their exact needs and express them in CEP queries, and then CEP engines evaluate those queries against incoming data events. For proactive monitoring, users cannot specify exactly what they are looking for and need the assistance of the CEP engine to derive a formal description of their data interests. This thesis makes contributions for both areas of passive monitoring and proactive monitoring.

For passive monitoring, the first contribution I make is to apply CEP queries over streams with imprecise timestamps, which was infeasible before this work. Existing CEP systems assumed that the occurrence time of each event is known precisely. However I observe that event occurrence times are often unknown or imprecise due to loss of information from derived data, granularity mismatch and clock synchronization problem in distributed systems. Therefore, I propose a temporal model that assigns a time interval to each event to represent all of its possible occurrence times. Under the uncertain temporal model, I further propose two evaluation frameworks, a point-based framework which convert events with time intervals into events with point timestamps before pattern matching, and an event-based framework which matches patterns over events with time intervals directly. I also propose optimizations in these frameworks. My new approach achieves high efficiency for a wide range of workloads tested using both both real traces and synthetic datasets. While existing systems cannot process this type of streams, the throughput of my algorithm achieves as high as tens of thousands of events per second for the MapReduce case study. This contribution enabled CEP techniques for more application scenarios.

Another contribution for the passive monitoring is that I identify expensive queries in CEP, analyze their runtime complexity, and propose effective optimizations to improve their performance significantly. Those expensive queries involve Kleene closure patterns, flexible event selection strategies, and events with imprecise timestamps. I analyze the runtime complexity of each language component and identify two performance bottlenecks: Kleene closure under the most flexible event selection strategy and confidence computation in the case of imprecise timestamps. For the first bottleneck, I break query evaluation into two parts: pattern matching, which can be shared by many matches, and result construction. Optimizations for the shared pattern matching cut cost from exponential to polynomial time and even close-to-linear. To address the second bottleneck, I design a dynamic programming algorithm to improve performance. Microbenchmark results show state-of-the-art systems suffer poor performance, while my system can provide 2 to 10 orders of magnitude improvement. A thorough case study on Hadoop cluster monitoring further demonstrates the efficiency and effectiveness of my proposed techniques: the throughput is over 1 million events per second.

The last problem in this thesis is proactive monitoring: explaining anomalous behaviors that users annotate on CEP-based monitoring results. Given the new requirements for explanations, namely, conciseness, consistency with human interpretation, and prediction power, most existing techniques cannot produce explanations that satisfy all three of them. The key technical contributions include a formal definition of optimally explaining anomalies in CEP monitoring, and three key techniques for generating sufficient feature space, characterizing the contribution of each feature to the explanation, and selecting a small subset of features as the optimal explanations, respectively. Our entropy distance function outperforms state-of-the-art distance functions on time series by reducing the features considered by 94.6%. Our system outperforms existing techniques by improving consistency from 10.7% to 87.5% on average, and reduces 90.5% of features on average to ensure conciseness. Our implementation is also efficient: with 2000 concurrent monitoring queries, the triggered explanation analysis returns explanations within half a minute and affects the performance only slightly, delaying events processing by 0.4 second on average.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

Complex Event Processing (CEP) systems are becoming increasingly popular in domains for decision analytics such as financial services, transportation, cluster monitoring, supply chain management, business process management, and health care. These systems collect or create high volumes event streams, and often require such event streams to be processed in real-time. To this end, CEP queries are applied for filtering, correlation, aggregation, and transformation, to derive high-level, actionable information.

As a new stream processing paradigm that addresses the 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 a wide range of pattern queries that address temporal correlations of events. Prior research [3] has shown that such pattern queries are more expressive than selection-join-aggregation queries and regular languages. This thesis addresses challenges arising in the context of complex event processing.

In the introduction, I present some motivating applications in Section 1.1. In Section 1.2, I briefly discuss the three research challenges. Following are the research contributions in Seciont 1.3. Finally, I show the organization of this thesis in Section 1.4.

## 1.1 Motivating Applications

CEP is now a crucial component in many IT systems in businesses. 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. Besides these well-known applications, CEP is gaining importance in a number of emerging applications, which particularly motivated my work in this thesis:

1) Logistics management: Logistics management enabled by sensor and RFID technology, is gaining adoption in supply chain mangement [32], hospitals [48], etc. We are talking these specific areas in the following paragraphs.

Supply chains connect manufacturers and retailers through a distribution network. At each manufacturer, an RFID tag is affixed to each product. Subsequent scans in the supply chain will generate new readings that report the primitive information including tag id, reader id and the time. All these readings can be sent to a CEP system that transforms the raw data into information of interest as the data arrives. CEP queries can be applied to this data stream to detect events such as contaminated shipments, expired or spoiled products, shoplifting activities, and misplaced inventory.

RFID-based object tracking are also useful in hospitals. Hospitals are busy environments where many medical devices need to appear in different locations in a particular order. For example, some medical tools need to be disinfected in several steps by machines at locations in a predefined order. If the tools to be disinfected went through different locations or machines in a wrong order, the tools might not be ready for use. Similar to the supply chain use case, RFID tags are attached to medical tools for tracking purposes. Readings from readers deployed at different locations form an event stream, and the order requirement on medical tools can be expressed by CEP queries. Then the CEP engine evaluates these queries against the continuously arriving stream in real-time, and generates warnings when violations occur.

However the timestamp of the data in logistic management is imprecise. Raw RFID data provides primitive information such as tag ids, reader ids and timestamp, which is known to be lossy. Meaningful events such as object movements and containment changes are often derived using probabilistic inference[46]. Thus the occurrence time of those events is imprecise, and it causes problems when the CEP engine transforms those events into a temporal sequence.

2) Cluster monitoring: Cluster computing has gained wide-spread adoption in big data analytics. Monitoring a computer cluster, such as a Hadoop cluster, has become crucial for understanding performance issues and managing resources properly [8].

Pattern queries can be issued to various system logs to monitor large clusters and help achieve better performance and resource management. In cloud computing, Internet data centers (IDC), and content-delivery networks (CDN), popular cluster monitoring tools such as Ganglia [33] provide system measurements regarding CPU, memory, and I/O from outside user programs. However, there is an increasing demand to correlate such system measurements with workload-specific logs (e.g., the start, progress, and end of Hadoop tasks) in order to identify unbalanced workloads, task stragglers, queuing of data, etc. Manually writing programs to do so is very tedious and hard to reuse. Hence, the ability to express monitoring needs in CEP queries becomes the key to freeing users from manual programing. However, the imprecise timestamp issue arises again here due to granularity mismatch: for example, system metrics by Ganglia are logged in the unit of 15 seconds, while Hadoop logs are often in the unit of milliseconds. In addition, many monitoring queries require the correlation of a series of events, which can be widely dispersed in a trace or multiple traces from different machines. This requirement needs the Kleene closure operator (which will be defined in Chapter 2) to select a finite yet unbounded sequence for a match. Applying Kleene closure operators over streams with imprecise timestamps in real time brings big challenges on the performance of CEP systems.

In cluster monitoring, there is another challenging request regarding explicability of monitoring results. When users observe unexpected anomalies in the visualized monitoring results there is often no obvious explanation. Users do not know what is going on in the monitored system, whether they can simply ignore them or they should take actions to stop the anomalies. In order to explain the anomalies, users have to examine a large volume of logs. The process is tedious and time consuming, leading to missing the best time to correct errors in the system. Even after spending time investigating the explanations, it is highly possible that users fail to find the true explanation.

## 1.2 Research Challenges

The applications above motivate three research challenges, and I classify them into two categories: passive monitoring and proactive monitoring. The challenges are as follows, the first two of which belong to passive monitoring, while the last one falls into the category of proactive monitoring.

- Existing technologies process event streams with precise timestamps, and they are unable to process event streams with imprecise timesatmps. It is challenging to apply CEP queries over streams with imprecise timestamps.

- The performance suffers when existing CEP engines evaluate expensive queries involving Kleene closure and imprecise timestamps. It is difficult to improve the throughput for such kind of queries and data.

- When users observe anomalies in monitoring results generated by existing CEP engines, they need help to explain the anomalies however existing engines lack such functionality. It is challenging to automatically generate human readable explanations and use that for proactive monitoring.

## 1.3 Contributions

I solved all three challenges mentioned above. For passive monitoring, I extend CEP techniques for events with imprecise timestamps. I propose optimizations for improving evaluation performance expensive queries involving Kleene closure operators and imprecise timestamps. For proactive monitoring, I build a system to generate explanations for user annotated anomalies. I will briefly introduce the three problems in the following paragraphs of this section.

### 1.3.1 Recognizing Patterns over Streams with Imprecise Timestamps

The first problem I solved is to apply CEP queries over streams with imprecise timestamps. Existing work 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 [3, 16, 31, 41, 52] or a strict partial order [4, 5, 6, 15, 34, 51] on the event stream. However, I 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 as I show in the RFID-based tracking example. Event occurrence times are subject to granularity mismatch as shown in the Hadoop cluster monitoring example. Events collected from a distributed system are subject to the clock synchronization problem, which also exists in the Hadoop cluster monitoring example.

A basic idea underlying my work is to employ a *temporal uncertainty model* that assigns a time interval to each event for representing all of its possible occurrence times. This model is easy to adopt and allows us to enumerate all possible orders of events and hence find potential pattern matches given each particular order of events. The research challenge lies in the high cost of enumeration: if every event in the stream has several possible occurrence times, the total search space will be exponential in the number of events. By ways of addressing pattern evaluation under the temporal uncertainty model and particularly the research challenge, I make the following technical contributions:

*Formal Semantics.* I 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 the matches from those possible worlds into a succinct result format where each match signature is associated with a confidence. This formal semantics offers a foundation for reasoning about the correctness of implementations.

*Evaluation Frameworks and Optimizations*. I 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. I present evaluation methods in these frameworks, prove their correctness, and further devise optimizations to improve efficiency.

*Complexity Analysis*. To provide a better understanding of the above two evaluation frameworks, I analyze their complexities including both the pattern matching cost and the confidence computation cost (which dominates in the result collapsing step). The complexity of pattern matching depends not only on parameters like the window size but also on how events are arranged in a window and whether an event can match different pattern components. I analyze the pattern marching complexity by considering all of these cases. The cost of confidence computation depends on the event selection strategy used in the query and I analyze its complexity by considering two common event selection strategies.

*Evaluation*. I evaluate my evaluation frameworks and algorithms using data traces collected from the applications of MapReduce cluster monitoring and RFID-based object tracking as well as synthetic event streams.While existing systems cannot process this type of streams, the throughput of my algorithm achieves as high as tens of thousands of events per second for MapReduce case study.

To the best of my knowledge, this is the first work that solves the problem of recognizing patterns over streams with imprecise timestamps. Existing systems can adopt the proposed new model and the point-based framework with minimum modifications. The optimizations are highly effective and can enable real-time processing on large volume of data.

### 1.3.2 Complexity and Optimization of Expensive Queries in Complex Event Processing

The second contribution is that I improve the evaluation performance significantly for expensive queries involving Kleene closure patterns, flexible event selection strategies, and events with imprecise timestamps. Without the optimizations in this work, existing systems are very slow to evaluate those expensive queries in production systems. The challenges of the problems come from the combination of the following factors:

*Kleene closure* is a special component in CEP, which can be used to collect a finite yet unbounded number of events.

*Event selection strategies*: The strategy on how to select those events relevant to a pattern is called *event selection strategy* in the literature. The most strict form selects events only continuously in the input (*strict*), a relaxation of the previous one selects events only continuously in the same partition(*partition contiguity*), a more flexible form skips irrelevant events until finds the relevant events to match the pattern (*skip till next match*), and the most flexible form finds all possible ways to match the pattern in the input (*skip till any match*). The increased flexibility in event selection leads to significantly increased complexity of pattern queries, with most existing solutions [3, 34, 35, 52] unable to support the most flexible strategy for Kleene closure or even simple pattern queries.

*Imprecise timestamps*: as I discussed in previous problem, imprecise timestamps exist due to varieties of reasons. In the solution of dealing with imprecise timestamps, one important configuration is that the CEP system has to be run under the most flexible selection strategy no matter what strategy is specified in the original query. Such that the CEP engine can detect all possible matches.

When dealing with these challenges, my technical contributions include:

*Runtime Complexity*: I begin my study by analyzing the runtime complexity of different CEP queries. This analysis shows how the runtime complexity changes as I add more key language features to queries. This "runtime analysis" reveals two types of expensive queries:

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).

*Optimizations*: To address bottleneck (1), I derive an insight from the observed difference between the low-level complexity classes in descriptive complexity analysis[56] (which considers only one match) and exponential complexity in runtime analysis (which considers all pattern matches). My optimization breaks query evaluation into two parts: pattern matching, which can be shared by many matches, and result construction, which constructs individual results. I propose a series of optimizations to reduce shared pattern matching cost from exponential to polynomial time (sometimes close-to-linear). To address bottleneck (2), I provide a dynamic programming algorithm to expedite confidence computation and to improve performance when the user increases the confidence threshold for desired matches.

*Evaluation with a case study*: I compare my new system with a number of state-of-the-art pattern query systems. Microbenchmark results show state-of-the-art systems suffer poor performance, while my system can provide 2 to 10 orders of magnitude improvement. In addition, I perform a case study in cluster monitoring using real Hadoop workloads, system traces, and a range of monitoring queries. I show that my system can automate cluster monitoring using declarative pattern queries, return very insightful results, and support real-time processing even for expensive queries. The throughput is over 1 million events per second.

Overall, the optimization techniques improve the evaluation performance of expensive queries by 2 to 10 orders of magnitude, making it possible to evaluate those queries in real-time. The use case study on Hadoop cluster monitoring provides a good way for Hadoop users to monitor and diagnose their jobs.

### 1.3.3 Explaining Anomalies in Event Stream Monitoring

The last problem in this thesis is proactive monitoring, specifically, explaining anomalies in event stream monitoring. With expressive query languages and high performance processing power, CEP is now the contributing technology under the hood of real-time monitoring systems in varieties of areas. For instance, monitoring communication between the controllers in a chip fabrication line to ensure correct progress of the pallets of wafers through the processing steps[32]; monitoring the progress of mortgage applications across a global network of cooperating financial institutions[32]; monitoring the progress of Hadoop jobs[56]. Users of these applications sit and watch the visualized results to know the status of monitored systems in real-time.

However, today's CEP technology supports only *passive monitoring* by requesting the monitoring application (or user) to explicitly define patterns of interest. There is a recent realization that many real-world applications demand a new service beyond passive monitoring, that is, the ability of the monitoring system to identify interesting patterns (including anomalous behaviors), produce a concrete explanation from the raw data, and based on the explanation enable a user action to prevent or remedy the effect of an anomaly. We broadly refer to this new service as *proactive monitoring*.

In this thesis we present *EXstream* system for proactive monitoring. The overall goal of *EXstream* is to provide good explanations for anomalous behaviors that users annotate on CEP monitoring results. We assume that an enterprise information system has CEP monitoring functionality: a CEP monitoring system offers a dashboard to illustrate high-level metrics computed by a CEP query. When a user observes an abnormal value in the monitoring results, he annotates the value in the dashboard and requests to search for an explanation from the archived raw data streams. *EXstream* generates high quality explanation by quickly replaying a fraction of the archived data streams. Then the explanation can be encoded into the system for proactive monitoring for similar anomalies in the future.

The challenges in the design of *EXstream* arise from the requirements for such explanations. Informed by the two real-world applications mentioned above, we consider three requirements in this work: (a) *Conciseness*: The system should favor smaller explanations, which are easier for humans to understand. (b) *Consistency*: The system should produce explanations that are consistent with human interpretation. In practice, this means that explanations should match the true reasons for an anomaly (*ground truth*). (c) *Prediction power*: We prefer explanations that have predictive value for future anomalies.

To solve this problem, I make the following contributions:

*Formalizing explanations* : I provide a formal definition of optimally explaining anomalies in CEP monitoring as a problem that maximizes the information reward provided by the explanation.

*Sufficient feature space*: A key insight in our work is that discovering explanations first requires a sufficient feature space that includes all necessary features for explaining observed anomalies. $EXstream$ includes a new module that automatically transforms raw data streams into a richer feature space, **F**, to enable explanations.

*Entropy-based, single-feature reward*: As a basis for building the information reward of an explanation, we model the reward that each feature, $f \in \mathbf{F}$, may contribute using a new entropy-based distance function.

*Optimal explanations via submodular optimization*: We next model the problem of finding an optimal explanation from the feature space, **F**, as a submodular maximization problem. Since submodular optimization is NP-hard, we design a heuristic algorithm that ranks and filters features efficiently and effectively.

*Evaluation*: We have implemented $EXstream$ on top of the SASE stream engine [3, 56]. Experiments using two real-world use cases show promising results: (1) Our entropy distance function outperforms state-of-the-art distance functions on time series by reducing the features considered by 94.6%. (2) $EXstream$ significantly outperforms logistic regression [2], decision tree [2], majority voting [28] and data fusion [37] in consistency and conciseness of explanations while achieving comparable, high predication accuracy. Specifically, it outperforms others by improving consistency from 10.7% to 87.5% on average, and reduces 90.5% of features on average to ensure conciseness. (3) Our implementation is also efficient: with 2000 concurrent monitoring queries, the triggered explanation analysis returns explanations within half a minute and affects the performance only slightly, delaying events processing by 0.4 second on average.

In summary, we present $EXstream$, a system that provides high-quality explanations for anomalous behaviors that users annotate on CEP-based monitoring results. Formulated as a submodular optimization problem, which is hard to solve, we provide a new approach that integrates a new entropy-based distance function and effective feature ranking and filtering methods. Evaluation results show that $EXstream$ outperforms existing techniques significantly in conciseness and consistency, while achieving comparable high prediction power and retaining a highly efficient implementation of a data stream system.

## 1.4 Thesis Organization

The reminder of this thesis is organized as follows. In Chapter 2, I introduce the necessary background about CEP. In Chapter 3, I present the techniques for recognizing patterns over imprecise timestamps. In Chapter 4, I show the complexity analysis and optimizations of expensive queries in complex event processing. In Chapter 5, I show the solution to explain anomalies in CEP-based monitoring. I conclude in Chapter 6 and discuss future work.

# CHAPTER 2
# BACKGROUND

In this chapter, I define a “core language” for pattern queries, introduce its formal semantics, and present the formal semantics by an NFA$^b$ model. This discussion offers a technical context for our study in the subsequent chapters.

## 2.1 A Core Language for Pattern Queries

A number of languages for CEP have been proposed, including SQL-TS [41], Cayuga [15], SASE [3, 52], and CEDR [6]. Although designed with different grammar and syntax, the core features for pattern matching are similar. Below, I define a core language, $\mathcal{L}$, for pattern queries, which includes necessary constructs to be useful in real-world applications, but leaves out derived features that do not change the complexity classes shown below.

The core language $\mathcal{L}$ employs a simple event model: Each event represents an occurrence of interest; it includes a timestamp 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 $\mathcal{L}$ 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 ($\sim$ 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, $\sim$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* ($\sim$ or !) can be applied to a SEQ pattern, e.g., $\sim$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 I do not include them in $\mathcal{L}$.

The overall structure of a pattern query is as follows:

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

Table 2.1 shows three example queries to illustrate the language. The queries are written using the syntax used in [3, 34, 48, 52]. The *Mapper Statistics(Q1)* query 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, $a$, $b$[ ] and $c$, with the array variable $b$[ ] 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.

The Shoplifting Detection query detects shoplifting activity in RFID-based retail management [52, 3]: it reports items that were picked at a shelf and then taken out of the store without being checked out. The 'Pattern' clause also specifies a SEQ pattern with three components: the occurrence of a shelf reading, followed by the non-occurrence of a check-out reading, followed by the occurrence of an exit reading. Non-occurrence of an event, denoted by '!' or '$\sim$', is also referred to as negation. The predicate requires all events to refer to the same tag id. Such equality comparison across all events is referred to as an equivalence test (a shorthand for which is [tag_id]). Finally, the query uses a 'within' clause to specify a 12-hour time window over the entire pattern.

The Load Imbalance (Q6) 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.

| **Query Name** | **Pattern Query** |
|---|---|
| Mapper Statistics (Q1) | Pattern SEQ(JobStart $a$, Mapper+ $b$[ ], JobEnd $c$)<br>Where $a$.job_id = $b[i]$.job_id $\wedge$ $a$.job_id=$c$.job_id<br>Within 1 day<br>Return $avg$($b$[ ].period), $max$($b$[ ].period) |
| Shoplifting Detection | Pattern SEQ(Shelf $a$, !(CheckOut $b$), Exit $c$)<br>Where [tag_id]<br>Within 12 hours<br>Return $c$.location, $c$.tag_id |
| Load Imbalance (Q6) | Pattern SEQ(ReducerStart $a$, LoadStd+ $b$[ ], ReducerEnd $c$)<br>Where [task_id] $\wedge$ ($b[i]$.val $\geq$ $b[i$-1].val //option 1)<br>($b[i]$.val $\geq$ $max$($b$[1..$i$-1].val //option 2)<br>Within 10 minutes<br>Return $a$.task_id |

**Table 2.1.** Example pattern queries.

### 2.1.1 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:

$S_1$: Strict or partition contiguity '—'. 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.

$S_2$: 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.

$S_3$: Skip till any match '$\Rightarrow$'. 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

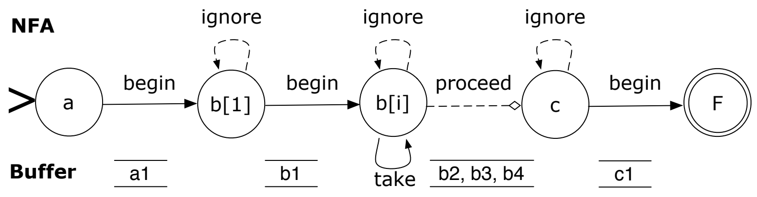


**Figure 2.1.** An NFA$^b$ automaton for Query 6.

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 [3, 15, 34]. In this work, I adopt the **NFA$^b$** model in [3] 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 2.1 is the NFA$^b$ for Q6.

States: In the NFA$^b$ automaton, a non-Kleene+ component of a pattern is represented by one state, and a Kleene+ component by two consecutive states. In Figure 2.1, 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, I 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$^b$ 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 I analyze runtime complexity in later chapters.

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$^b$ automata combined using these four operations.

# CHAPTER 3

# RECOGNIZING PATTERNS OVER STREAMS WITH IMPRECISE TIMESTAMPS

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) [3, 4, 6, 15, 16, 34, 41, 51, 52] 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 [3] has shown that such pattern queries are more expressive than selection-join-aggregation queries and regular languages.

Existing work, however, fundamentally relies on two assumptions. First, the occurrence time of each event is known precisely. Second, events from various sources can be merged into a single stream such that a binary relation (denoted by $\prec$) based on the event occurrence time gives rise to a total order [3, 16, 31, 41, 52] or a strict partial order [4, 5, 6, 15, 34, 51] 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.

I 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 [38, 46]. The actual occurrence time of object movement or containment change is unknown and can only be estimated to be in a range with high confidene.

Event occurrence times are subject to granularity mismatch. In cluster monitoring, for instance, a commonly used monitoring system, Ganglia [21], 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 I shall show formally in Section 3.2).

Events collected from a distributed system are subject to the clock synchronization problem. Consider causal request tracing in large concurrent, distributed applications [7, 25], which involve 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 from different machines into a single stream with a total or partial order [25].

In this chapter, I address pattern query evaluation in streams with imprecise occurrence times of events. Such events preclude the use of existing systems that assume a total order or strict partial order of events from various data sources. A basic idea underlying 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. This model is easy to adopt and allows us to enumerate all possible orders of events and hence find potential pattern matches given each particular order of events. The research challenge lies in the high cost of enumeration: if every event in the stream has several possible occurrence times, the total search space will be exponential in the number of events. By ways of addressing pattern evaluation under the temporal uncertainty model and particularly the research challenge, I make the following technical contributions:

Formal Semantics: I 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 the matches from those possible worlds into a succinct result format where each match signature is associated with a confidence. This formal semantics offers a foundation for reasoning about the correctness of implementations.

Evaluation Frameworks and Optimizations:. I 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. I present evaluation methods in these frameworks, prove their correctness, and further devise optimizations to improve efficiency.

Complexity Analysis: To provide a better understanding of the above two evaluation frameworks, I analyze their complexities including both the pattern matching cost and the confidence computation cost (which dominates in the result collapsing step). The complexity of pattern matching depends not only on parameters like the window size but also on how events are arranged in a window and whether an event can match different pattern components. I analyze the pattern marching complexity by considering all of these cases. The cost of confidence computation depends on the event selection strategy used in the query and I analyze its complexity by considering two common event selection strategies.

Evaluation: I evaluate our evaluation frameworks and algorithms using data traces collected from the applications of MapReduce cluster monitoring and RFID-based object tracking as well as synthetic event streams. Our evaluation yields a number of 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 a 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 studies of cluster monitoring and RFID object tracking, despite the large uncertainty intervals used.

## 3.1 Related work

Temporal databases: Temporal databases are surveyed in [9]. The most relevant work is supporting valid-time indeterminacy [17], whose indeterminate semantics shares the basic idea as our semantics. However, the work in [17] 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 [17], the proposed technique uses multi-way joins, which is less efficient than either of the two evaluation frameworks I 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.

Interval-based event processing: Several event processing systems [5, 4, 6, 15, 51] 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 [5, 6, 29, 43] 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.

Probabilistic Databases: Our work also differs from probabilistic databases and stream systems, such as [14, 38], which address the uncertainty of the *values* in events but not the *timestamps*. If I 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.

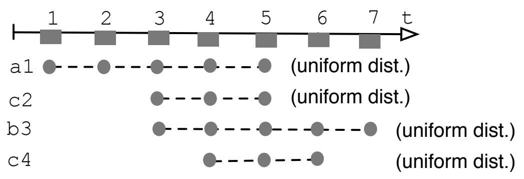
## 3.2 Model and Semantics

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

(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: 15/225 )
```

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

**Figure 3.1.** Semantics of pattern query evaluation under our temporal uncertainty model.

### 3.2.1 Temporal Uncertainty Model

I now consider events with uncertain occurrence times and propose an event model that accommodates temporal uncertainty. As in most temporal data model research [9], I assume a discrete, totally ordered time domain $T$; without loss of generality, I 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 [17]: 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, I 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 [43]. In our problem, there is no clear notion of "increasing order of the occurrence time" due to imprecise timestamps. So I 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, I assume that using buffering or advanced techniques for out-of-order streams [29, 43], I 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.

### 3.2.2 Formal Semantics under the Model

I 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, I refer to $S_k$ as a *possible world* for pattern evaluation, and compute its probability as $\mathbb{P}[S_k] = \prod_{i=1}^{n} f(t_{ij})$. I then perform pattern matching in every possible world $S_k$, as in any existing event system.

*Example*: Fig. 3.1(a) shows a sequence pattern with a 4-second time window (assuming that a time unit is a second). Fig. 3.1(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. 3.1(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 my work , I instead present these match sets in a succinct way. More specifically, I 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. 3.1, 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. 3.1(d).

## 3.3 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

Stream Expansion: The first issue is that existing pattern query engines [3, 15, 34] require that events be arranged in total or partial order based on their occurrence times. As stated in §3.2.1, 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. 3.1(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 §3.2.1 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 $e_1, \ldots, e_{n-1}, e_n$ 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 $now.\text{lower} - U_{max}$; otherwise, the unseen event will violate the arrival order constraint with the earlier event $e_i$ that sets $now.\text{lower} = e_i.\text{lower}$ . So we can safely output the buffered point events up to $now.\text{lower} - U_{max}$, labelled as the **emit time** $t_{emit}$. Then the time range used to bound the buffer size can be defined as follows: $max_{i=1}^{n}(e_i.\text{upper}) - t_{emit} = max_{e_i.\text{upper}} - max_i(e_i.lower) + U_{max}$.

Pattern Matching: We next evaluate pattern queries over the point event stream by leveraging an existing pattern query engine such as [3, 34]. 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. 3.2(a) shows all the time points of the four events in Fig. 3.1(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

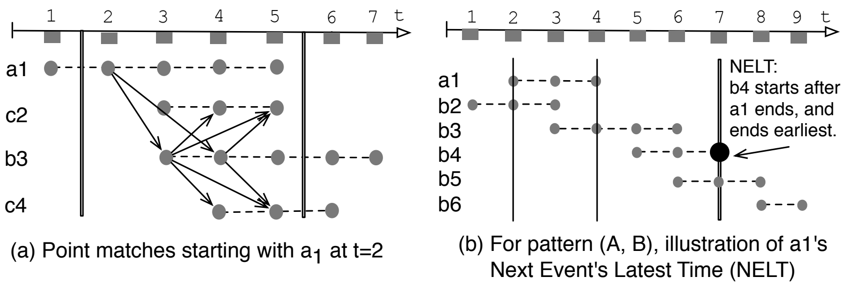

1 2 3 4 5 6 7 t
a1
c2
b3
c4
(a) Point matches starting with a1 at t=2
1 2 3 4 5 6 7 8 9 t
a1
b2
b3
b4
b5
b6
NELT: b4 starts after a1 ends, and ends earliest.
(b) For pattern (A, B), illustration of a1's Next Event's Latest Time (NELT)

**Figure 3.2.** The point-based evaluation framework.

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 3.11.1. Algorithm 1 shows the point-based evaluation procedure.

*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. 3.2(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

**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 $[max_{j=1}^{i}(e_j.lower), max_{j=1}^{i}(e_j.upper)]$
    Set $t_{emit}$ to $Now.lower - U_{max}$
    for $t = e_i$.lower to $e_i$.upper do
      Generate the point event $e_i^t$
      Add $e_i^t$ to the event buffer
    end for
    for Each point event $e_j^t$ in the event buffer do
      if $t = t_{emit}$ then
        Emit $e_j^t$ 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 $m : (e_{m_1}^{t_1}, ..., e_{m_\ell}^{t_\ell})$ 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 confidence of the match by calling Algorithm 3
      end if
    end for
  end for
```

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 $b_5.\mathtt{upper}$ 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.

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.

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 3.11.1. Finally, note that due to temporal uncertainty, skill till next match queries cannot be run directly on the

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

```
Input: Event $e$, Pattern $(E_1, ...E_\ell)$
  if $e$ is a point event then
    $e$ := e's original event
  end if
  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
```

point event stream using the same strategy. For example, starting from $a_1^3$ in Fig. 3.2(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 3.2.2. 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 $\alpha$, denoted by $S_\alpha$. The tightest time range for all the point matches is $[min_m(e_{m_1}^{t_1}$.lower), $max_m(e_{m_\ell}^{t_\ell}$.upper)$]$. The remaining task is to compute the confidence.

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

$$C_{any}(\alpha) = \sum_{m \in S_\alpha} \mathbb{P}\left[(e^{t_1}_{m_1}, \ldots, e^{t_\ell}_{m_\ell})\right]) = \sum_{m \in S_\alpha} \prod_{j=1}^{\ell} \mathbb{P}\left[e^{t_j}_{m_j}\right] \tag{3.1}$$

This calculation is correct because the probability of the point match $e^{t_1}_{m_1}, \ldots, e^{t_\ell}_{m_\ell}$ is the product of the probabilities of its individual 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 $a^2_1, b^3_3, c^5_2$ to be a match of the query in Fig. 3.1, we require that event $c_4$ does not occur at time 4. Formally, a potential point match $m = (e^{t_1}_{m_1}, \ldots, e^{t_\ell}_{m_\ell})$ is a true match iff (1) $t_1 < \ldots < t_\ell$, and (2) for each $e^{t_j}_{m_j}$ $(2 \leq j \leq \ell)$, no point event matching $E_j$ occurs between $e^{t_{j-1}}_{m_{j-1}}$ and $e^{t_j}_{m_j}$. Let $\Theta_j(m)$ be the set of all such excluded point events. Thus condition (2) may be written $\Theta_j(m) = \emptyset$ for $2 \leq j \leq \ell$, or $\Theta(m) = \emptyset$ for short. Then the confidence of skip-till-next match, $C_{next}(\alpha)$, equals:

$$C_{next}(\alpha) = \sum_{m \in S_\alpha} \mathbb{P}\left[(e^{t_1}_{m_1}, \ldots, e^{t_\ell}_{m_\ell})\right] = \sum_{m \in S_\alpha} \prod_{j=1}^{\ell} \mathbb{P}\left[e^{t_j}_{m_j}\right] \cdot \mathbb{P}\left[\Theta(m) = \emptyset\right] \tag{3.2}$$

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. Now let us consider the *intervening events* that do not belong to the match $m$ but can match a pattern component. It is evident that each intervening event can occur in at most one $\Theta_j(m)$ set, so these sets being empty are independent of each other. Hence, we can rewrite Eq. 3.2 as:

$$C^1_{next}(\alpha) = \sum_{m \in S_\alpha} \frac{\prod_{j=2}^{\ell} \mathbb{P}\left[e^{t_{j-1}}_{m_{j-1}}\right] \cdot \mathbb{P}\left[e^{t_j}_{m_j}\right] \cdot \mathbb{P}\left[\Theta_j(m) = \emptyset\right]}{\prod_{j=2}^{\ell-1} \mathbb{P}\left[e^{t_j}_{m_j}\right]} \tag{3.3}$$

The equation above leads to a memorization-based algorithm to compute $C^1_{next}(\alpha)$. For all point matches in $S_\alpha$, it computes the quantity $\mathbb{P}\left[e^{t_{j-1}}_{m_{j-1}}\right] \cdot \mathbb{P}\left[e^{t_j}_{m_j}\right] \cdot \mathbb{P}\left[\Theta_j(m) = \emptyset\right]$

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 build an index on the fly to remember those events that can potentially match the $j$-th 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 $e_{m_j}^{t_j}$.

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 the intervening events, $\bar{S}$, that do not belong to the match $m$ but can match more than on pattern component, and hence belong to more than one $\Theta_j(m)$ set. In the enumeration process, as we become conditioned on the specific points of events in $\bar{S}$, we can factorize $\Theta(m) = \emptyset$ based on independence. Therefore,

$$C_{next}^2(\alpha) = \sum_{e_k \in \bar{S}} \prod_{t_k} \mathbb{P}\left[e_k^{t_k}\right] \cdot \sum_{m \in S_\alpha} \prod_{j=1}^{\ell} \mathbb{P}\left[e_{m_j}^{t_j}\right] \prod_{j=2}^{\ell} \mathbb{P}\left[\Theta_j(m) = \emptyset | \{e_k^{t_k}\}\right] \tag{3.4}$$

Our algorithm for the second case extends that of the first case by using the event index to also compute the conditional probability, $\mathbb{P}\left[\Theta_j(m) = \emptyset | \{e_k^{t_k}\}\right]$, in addition to implementing memoization.

Algorithm 3 shows the computation of the match confidence of skip-till-next-match queries for both of the cases described above. When it is applied to the first case, the set $\bar{S}$ is empty, which means that there are no intervening events that can match multiple components. In this case, we just need to compute the confidence with given intervening sets, $\Theta_2(m), \Theta_3(m)..., \Theta_\ell(m)$, without enumeration, so the loop in Line 9 will have only one order. When it is applied to the second case, the set $\bar{S}$ will contain intervening events that can match multiple components. The algorithm then will enumerate the specific points of events in $\bar{S}$, and conditioned on these points of events, we can compute the confidence. Finally, the sum of the confidence computed from each combination of the points of the events will yield the final confidence of the match.

Benefits: The point-based evaluation framework offers three key benefits: First, it has tremendous performance benefits over an evaluation method based on the formal semantics—

**Algorithm 3** Compute the confidence of skip-till-next-match queries for the point-based framework.

```
Input: match m:$(e_{m_1}, e_{m_2}, ...e_{m_\ell})$, $\bar{S}$: intervening events that have matched multiple components, $\acute{S}_i(m)$: the set of intervening events that can potentially extend a partial match ending at $e_{m_{i-1}}$ but do not appear in $m$, excluding the events in $\bar{S}$.
 1: if $\bar{S} \neq \emptyset$ then
 2:     Enumerate the points of events in $\bar{S}$ in all possible orders, listed as $O = \{O_1, O_2, \ldots, O_R\}$
 3:     $\mathbb{P}[O_r] := \prod_{e_k \in \bar{S}} \mathbb{P}\left[e_k^{t_k}\right]$
 4: else
 5:     $O := \{\perp\}$
 6:     $\mathbb{P}[\perp] := 1$
 7: end if
 8: $Conf(m) := 0$
 9: for Order $O_r$ in $O$ do
10:     for Point match $m_p \in m$ do
11:         for i=1 to i=$\ell$-1 do
12:             $e_{m_i}^{t_i}$ = point event of $m_p$ at state $i$
13:             $e_{m_{i+1}}^{t_{i+1}}$ = point event of $m_p$ at state $i+1$
14:             if $\mathbb{P}\left[e_{m_i}^{t_i} \prec e_{m_{i+1}}^{t_{i+1}}\right]$ is not computed yet then
15:                 $\mathbb{P}\left[e_{m_i}^{t_i} \prec e_{m_{i+1}}^{t_{i+1}}\right] = \mathbb{P}\left[e_{m_i}^{t_i}\right] \times \mathbb{P}\left[e_{m_{i+1}}^{t_{i+1}}\right]$
16:                 $\mathbb{P}\left[e_{m_{i+1}}^{t_{i+1}} \text{ is the first match after } e_{m_i}^{t_i} | O_r\right]$
                    $= \prod_{e_k \in \acute{S}_{i+1}(m)} \mathbb{P}\left[e_k \text{ is not in } [t_i, t_{i+1}]\right]$
                    $\times \mathbb{P}\left[\text{None of } \bar{S} \text{ matching } E_{i+1} \text{ occurs in } [t_i, t_{i+1}] | O_r\right]$
17:             end if
18:         end for
19:     end for
20:     $Conf(m|O_r) := 0$
21:     for Point match $m_p \in m$ do
22:         for $i = 1$ to $i = \ell$ do
23:             $e_{m_i}^{t_i}$ = point event of $m_p$ at state $i$
24:         end for
25:         $Conf(m|O_r)$ += $\frac{\prod_{i=1}^{\ell-1} \mathbb{P}\left[e_{m_i}^{t_i} \prec e_{m_{i+1}}^{t_{i+1}}\right] \times \mathbb{P}\left[e_{m_{i+1}}^{t_{i+1}} \text{ is the first match after } e_{m_i}^{t_i} | O_r\right]}{\prod_{i=2}^{\ell-1} \mathbb{P}\left[e_{m_i}^{t_i}\right]} \times \mathbb{P}[O_r]$
26:     end for
27:     $Conf(m)$ += $Conf(m|O_r)$
28: end for
```

the latter is infeasible in most workloads. More precisely, the point-based evaluation method dynamically finds the possible worlds in which the order of point events matches the query-specified order, and simply ignores other possible worlds. Second, it requires the minimum change of an existing pattern query engine, hence easy to use. Third, when an application receives the query match (which is a collapsed format) but wants the detailed point matches, it can run the point-based evaluation method over the events in the query match, if the query uses the skip till any match strategy. This re-evaluation incurs little cost because it involves only a few events, as opposed to many more in the window on the input stream. However, we note that if the query uses the skip till next match strategy, it is incorrect to rerun the point-based evaluation because we also need the intervening events in addition to the events in the query match. In this case, the application cannot recover the point matches.

## 3.4 An Event-based Framework

In this section, we present a second evaluation framework which is event based rather than point based. This way, we can eliminate the 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.

### 3.4.1 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 $\ell$ 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 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 $\mathrm{ext}(m,e)$ is true iff event $e$ may extend the partial match $m$ of pattern $p$. To compute $\mathrm{ext}(m,e)$, 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 $e$.vlb $=$ $e$.lower. Inductively, assume that $m = (e_{m_1}, \ldots, e_{m_j})$ and $e_{m_j}$.vlb is defined. If $e \models E_{j+1}$, define $e.\mathrm{vlb} = \max(e_{m_j}.\mathrm{vlb} + 1, e.\mathrm{lower})$. Thus, $e$.vlb is the first time that $e$ might occur in the match $m$.

Using vlb we can immediately define ext:

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

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. 3.3(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, $a_1$.vlb $= 1$, $\mathrm{ext}(\emptyset, a_1) = \mathit{True}$; $b_3$.vlb $= 3$, $\mathrm{ext}((a_1), b_3) = \mathit{True}$; and $c_2$.vlb $= 4$, $\mathrm{ext}((a_1, b_3), c_2) = \mathit{True}$, 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, $e_{m_\ell}.\mathrm{rub} = e_{m_\ell}.\mathrm{upper}$. Inductively, assume that $e_{m_{j+1}}$.rub is defined, and let $e_{m_j}.\mathrm{rub} = \min(e_{m_{j+1}}.\mathrm{rub} - 1, e_{m_j}.\mathrm{upper})$. As we compute the revised upper bounds, we check that each interval is nonempty, that is, $e_{m_j}.\mathrm{vlb} < e_{m_j}.\mathrm{rub}$; otherwise, the match $m$ is pruned.

*Example*. Fig. 3.3(b) shows the computation of the revised upper bounds after the match $(a_1,b_3,c_2)$ is recognized. That is, $c_2$.rub $= 5$, $b_3$.rub $= 4$, and $a_1$.rub $= 3$. The match is preserved in this step.

3. Pruning based on the Query 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 $e_{m_1}$.rub, the last possible time for $e_{m_\ell}$ is at most $T_m = e_{m_1}.\text{rub} + W - 1$. In the first case, $e_{m_\ell}.\text{rub} \leq T_m$. 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 $e_{m_{j+1}}$.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 $e_{m_j}$ in the current match has $e_{m_j}.\text{vub} < e_{m_j}.\text{vlb}$, then the match fails.

*Example*. Fig. 3.3(c) shows an example using three events $a_1$, $b_5$, and $c_6$. In the first pass, we compute the valid lower bounds as: $a_1$.vlb $= 1$, $b_5$.vlb $= 2$, and $c_6$.vlb $= 6$. After the second pass, we have: $c_6$.rub $= 7$, $b_5$.rub $= 3$, and $a_1$.rub $= 2$. Then we have $T_m = a_1.\text{rub} + W - 1 = 5$. 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. While the valid lower and upper bounds initially may not be as tight as

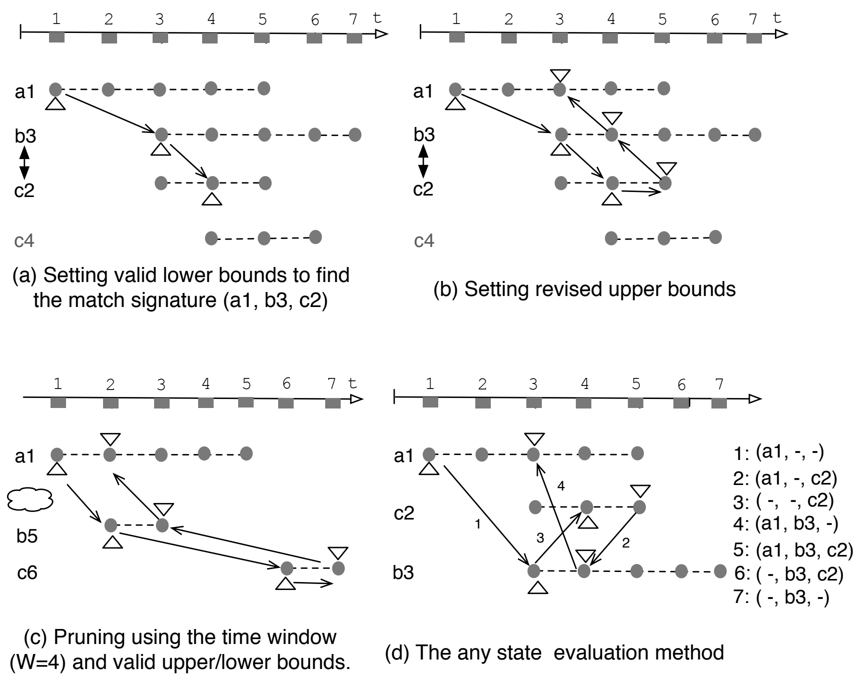

1 2 3 4 5 6 7 t
a1
b3
c2
c4
(a) Setting valid lower bounds to find the match signature (a1, b3, c2)
(b) Setting revised upper bounds
a1
b5
c6
(c) Pruning using the time window (W=4) and valid upper/lower bounds.
a1
c2
b3
1
2
3
4
1: (a1, -, -)
2: (a1, -, c2)
3: ( -, -, c2)
4: (a1, b3, -)
5: (a1, b3, c2)
6: ( -, b3, c2)
7: ( -, b3, -)
(d) The any state evaluation method

**Figure 3.3.** Illustration of the event-based evaluation (assuming that events are presented in query order).

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 6-9).

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 10-21).

3. If $e_{m_1}.\text{rub} + W < e.upper$, the partial match is validated, go to next step(Line 22-25). Otherwise, compute the vub in a third pass. Again, check whether the valid interval of each event is empty (Lines 26-38).

4. If the partial match passes all checks, perform the pattern matching (Lines 39-46).

*Example*. Fig. 3.3(c) shows an example using three events $a_1$, $b_5$, and $c_6$. Upon arrival of $c_6$, we have a partial match $(a_1, b_5)$. Step 1 above sets $c_6$.vlb $= 6$ and $c_6$.vub $= 7$. Step 2 sets $b_5$.rub $= 3$ and $a_1$.rub $= 2$. Then in Step 3, the window constraint $W = 4$ is expressed as $c_6$.rub $> a_1$.vub $+ 4 - 1 = 5$. So we should set $c_6$.vub $:= 5$, and then $c_6$.vub $- c_6$.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.

Given events presented in query order, the incremental algorithm evaluates *skip till any match* queries using ext() and the skip till any match strategy. It supports *skip till next match* queries by further augmenting ext() using the Next Event's Latest Time (NELT) as proposed in Section 3.3, using the same procedure defined in Algorithm 2.

We have proved the following proposition, which shows the equivalence of the three-pass algorithm and the incremental algorithm. The correctness proof is provided in 3.12.

**Proposition 3.4.1.** The incremental algorithm can obtain the same results as the three-pass algorithm when it evaluates the same query over the same event stream.

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. Algorithm 5 shows the computation of the match confidence for a *skip till any match* query in the event-based framework.

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. This is a different case from that when we discuss the benefits of the point-based framework in Section 3.3 because we have the intervening events with the matches now.

### 3.4.2 The "Any State" Evaluation Method

We next relax the assumption that events are presented in query order. Instead, we consider events in their arrival order. Fig. 3.3(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. More specifically, given an event $e$, a run $\gamma$, and the

**Algorithm 4** Incremental Method for Query Order Evaluation

```
Input: Event Stream $S$, Pattern $(E_1, ..., E_\ell)$
 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 $e$ does not satisfies query component $E_{j+1}$ then
 4:             break
 5:         end if
 6:         $e$.vlb := $min(e_{m_j}.\text{lower} + 1, e.\text{vlb})$
 7:         if $e.\text{vlb} > e.\text{upper}$ then
 8:             break
 9:         end if
10:         $e$.rub := $e$.upper
11:         $rubCheckSucceed := TRUE$
12:         for each event $e_{m_i}(1 \leq i \leq j)$ in $m$ in reverse order as the match do
13:             $e_{m_i}$.rub := $min(e_{m_{i+1}}.\text{rub} - 1, e_{m_i}.\text{upper})$
14:             if $e_{m_i}.\text{rub} - e_{m_i}.\text{vlb} < 0$ then
15:                 $rubCheckSucceed := FALSE$
16:                 break
17:             end if
18:         end for
19:         if $rubCheckSucceed == FALSE$ then
20:             break
21:         end if
22:         $rubWindowCheckSucceed := FALSE$
23:         if $e_{m_1}.\text{rub} + W \geq e.\text{rub}$ then
24:             $e$.vub := $e$.rub
25:             $rubWindowCheckSucceed := TRUE$
26:         else
27:             $e$.vub := $e_{m_1}.\text{rub} + W - 1$
28:         end if
29:         if ($rubWindowCheckSucceed == FALSE$) and ($e.\text{vub} - e.\text{vlb} > 0$) then
30:             $vubCheckSucceed := TRUE$
31:             for each event $e_{m_i}(1 \leq i \leq j)$ in $m$ do
32:                 $e_{m_i}$.vub := $min(e_{m_{i+1}}.\text{vub} - 1, e_{m_i}.\text{upper})$
33:                 if $e_{m_i}.\text{vub} - e_{m_i}.\text{vlb} < 0$ then
34:                     $vubCheckSucceed := FALSE$
35:                     break
36:                 end if
37:             end for
38:         end if
39:         if ($rubWindowCheckSucceed == TRUE$) or ($vubCheckSucceed == TRUE$) then
40:             if Using skip till any match strategy then
41:                 ext$(m, e) := true$
42:             end if
43:             if Using skip till next match strategy then
44:                 Call Algorithm 2
45:             end if
46:         end if
47:     end for
48: end for
```

**Algorithm 5** Compute the confidence of skip till any match queries in the event-based framework.

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

set of events $m$ selected in $\gamma$ , 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. If a predicate of $E_j$ compares to other unmatched pattern components, defer it until it is instantiated later. If $e$ matches $E_j$ and can instantiate predicates between $E_j$ and other matched components, evaluate those predicates to filter $e$.

2. Temporal constraints: Let $E_i, \ldots, E_j, \ldots, E_k$ denote the contiguous matched pattern components involving $E_j$, $i \leq j \leq k$. Compute $e$'s valid lower bound using $e_{m_{j-1}}$'s valid lower bound if existent, or $e$'s lower bound otherwise. Compute $e$'s valid upper bound using $e_{m_{j+1}}$'s valid upper bound if existent, or $e$'s upper bound otherwise. Update the valid lower bound of the subsequent events $e_{m_{j+1}}, \ldots, e_{m_k}$ if present. Update the valid upper bound of the preceding events $e_{m_i}, \ldots, e_{m_{j-1}}$ if present. If

these updates cause any of the events to have an empty valid interval, i.e., vlb $>$ vub, skip $e$. If $e$ is retained, check the time window between the events matching the current two ends of the pattern to further filter $e$.

3. If $e$ is retained, clone $\gamma$ to $\gamma'$ and select $e$ to match $E_j$ in $\gamma'$.

As can be seen, the any state evaluation method is an incremental algorithm that runs directly on the event stream, without assuming that events are presented in query order.

*Example*. Fig. 3.3(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. 3.3(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. 3.3(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 non-viable runs using the window. Consider a run $\gamma$ 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 $\gamma_{tll}$. As more events are selected by $\gamma$, $\gamma_{tll}$ will only decrease but not increase. Recall from §3.3 that our system has a notion $now$ = $[max_{i=1}^{n}(e_i.lower), max_{i=1}^{n}(e_i.upper)]$ defined on all the events we have seen, and the maximum uncertainty interval size $U_{max}$. Further, the arrival order constraint in our system implies that any unseen event must start after $now.lower - U_{max}$. So, a run $\gamma$ can be safely pruned if $\gamma_{tll} < now.lower - U_{max}$.

Another pruning opportunity arises when a run $\gamma$ 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 $\gamma$ based on the arrival order constraint between $e_{m_j}$ and a future event matching $E_{j-1}$. Since any unseen event must start after $now.lower - U_{max}$, when $e_{m_j}$.upper $< now.lower - U_{max}$, we know that no future event can match $E_{j-1}$, and hence can safely prune $\gamma$.

Algorithm 6 shows the details of the incremental method. We have proved another proposition, which shows the equivalence between the any state evaluation method and the query order evaluation algorithm. The correctness proof is provided in 3.12.

**Proposition 3.4.2.** The any state evaluation method can obtain the same results as the query order evalulation method when it evaluates the same query over the same event stream.

## 3.5 Complexity Analysis

We consider a query pattern with $\ell$ components in our complexity analysis. Our analysis below uses the following symbols: (*i*) $U$: the number of instants in an event's uncertainty interval. (*ii*) $W$: the size of the time window used in the query. (*iii*) $\ell$: the number of pattern components of query. (*iv*) $R_i$: the arrival rate of events satisfying the $i$th pattern component by the event type and predicates. (*v*) $N$: the number of events in a time window. We break our complexity analysis into two parts: pattern matching and confidence computation, as detailed below.

### 3.5.1 Pattern Matching

We consider three cases of pattern matching and their respective complexities.

#### 3.5.1.1 Case 1: Mutually Exclusive Pattern Components with a Large Enough Window (i.e., $W \geq \ell U$)

We say that the $\ell$ components of a pattern are mutually exclusive if any given event can match at most one of them. Our analysis aims at a reasonable bound of the worst case

**Algorithm 6** Any State Evaluation Algorithm

```
Input: Event Stream $S$, Pattern $(E_1, ..., E_\ell)$
 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 a query component $E_k$ and $m$ has not selected an event for $E_k$ then
 4:             $rubCheckSucceed := TRUE$
 5:             for each event $e_{m_i}(1 \leq i \leq j)$ in $m$ do
 6:                 $e_{m_i}$.vlb $= max(e_{m_{i-1}}$.vlb $+ 1, e_{m_i}$.lower)
 7:                 $e_{m_i}$.rub $= min(e_{m_{i+1}}$.rub $- 1, e_{m_i}$.upper)
 8:                 if $e_{m_i}$.rub $< e_{m_i}$.vlb $(1 \leq i \leq j + 1)$ then
 9:                     $rubCheckSucceed := FALSE$
10:                     break
11:                 end if
12:             end for
13:             if $rubCheckSucceed := FALSE$ then
14:                 break
15:             end if
16:             $rubWindowCheckSucceed := FALSE$
17:             if  ($e_{m_1}$.rub $+ TW \geq e$.rub) then
18:                 $e$.vub $= e$.rub
19:                 $rubWindowCheckSucceed := TRUE$
20:             else
21:                 $e$.vub $= e_{m_1}$.rub $+ TW - 1$
22:             end if
23:             $vubCheckSucceed := TRUE$
24:             for each event $e_{m_i}(1 \leq i \leq j)$ in $m$ do
25:                 $e_{m_i}$.vub $= min(e_{m_{i+1}}$.vub $- 1, e_{m_i}$.upper)
26:                 if $e_{m_i}$.vub $< e_{m_i}$.lower then
27:                     $vubCheckSucceed := FALSE$
28:                     break
29:                 end if
30:             end for
31:             if ($rubWindowCheckSucceed == TRUE$) or ($vubCheckSucceed == TRUE$) then
32:                 if Using skip till any match strategy then
33:                     ext$(m, e) := true$
34:                 end if
35:                 if Using skip till next match strategy then
36:                     Call Algorithm 2
37:                 end if
38:             end if
39:         end if
40:     end for
41: end for
```

performance. (The exact performance characteristics of the point-based framework are presented in the evaluation section.) In this regard, we make several assumptions to simplify

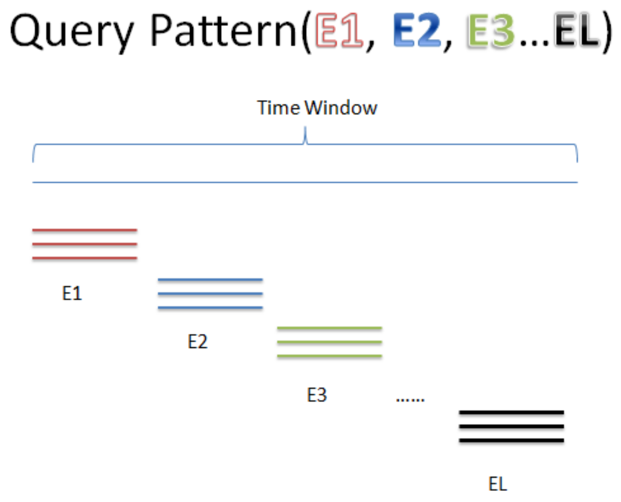


**Figure 3.4.** Event sequences that cause the worst-case performance.

the analysis: We consider different event types for different components in the query. We assume a uniform uncertainty interval size for all events. Furthermore, we assume that events of different types have the same arrival rate ($R_1 = R_2 = R_3 = \ldots = R$), and expect to see roughly the same number of events of each distinct type in a sufficiently large window ($W \geq \ell U$).

Complexity of skip till any match queries: We identify the worst case performance as the largest number of partial matches, also called runs, that the pattern matching process can generate for a sequence of events that fit in a time window. The worst case occurs when the events in the window are arranged in a particular order. More specifically, the largest number of runs occurs if the events are arranged as shown in Fig. 3.4. In this arrangement, the events of the first type (i.e., matching the first pattern component) appear first, before all events of other types. The events of the second type immediately follow the events of the first type. Then the events of the third type immediately follow, and so on. In 3.13, we show the proof that this arrangement can lead to the maximum number of runs. The events of different event types do not overlap. Then,

$\#Runs = (RWU)^{\ell}$, in the point-based framework, and

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

If we follow the possible-world semantics, we have $\#Runs = N^{\ell}$, which is pattern-agnostic. Note that in general $(RW)^{\ell} \ll N^{\ell}$ because $RW$ is much small than $N$.

Complexity of skip till next match queries: The worst case of a skip till next match query is the same as that of a skip till any match query. In this case, the Next Event's Latest Time (NELT) would not invalidate any event.

#### 3.5.1.2 Case 2: Mutually Exclusive Pattern Components with a Small Window ($W < \ell U$)

When the time window is not large enough, $W < \ell U$, intuitively, pattern matching will produce fewer the runs because the $\ell$ events, each of size $U$, overlap in time and not all combinations of their points can satisfy the ordering constraint. Based on the analysis of the previous case, we now subtract some point matches that violate the ordering constraint.

We begin the analysis with a special case, $W = U$, and a query that contains three components: $(A, B, C)$. Since $W = U$, the maximum number of runs occurs when the uncertainty interval of each event covers the whole time window (i.e., they completely overlap in the window). In the window, we use $(1, 2, \ldots U)$ to denote the different time points. At each point, we would have $RW$ events of the type of A, $RW$ events of the type of B, and $RW$ events of the type of C. Let us use $x_i$ to denote a certain event of the type of X at time i. Then for a certain point $a_1$, it can match all $b_2, b_3, \ldots, b_U$, which can be illustrated by Figure 3.5. For a certain point $b_2$, it can match all $c_3, c_4, \ldots, c_U$. Then the prefix $(a_1, b_2)$ can get $(U-2)$ runs. Similarly, for prefix $(a_1, b_3)$, we can get $(U-3)$ runs. And we can calculate the number of runs until we reach $(a_1, b_{U_1})$. So for this point $a_1$, $\#Runs = (U-2) + (U-3) + \ldots + 2 + 1 = \frac{(U-1)(U-2)}{2}$. Similarly, we can get for those points, $a_2, a_3, \ldots, a_{U-2}$, $\#Runs = \frac{(U-2)(U-3)}{2}, \frac{(U-3)(U-4)}{2}, \ldots, 3, 1$. The sum is $\#Runs = \frac{1}{2}\sum_{k=1}^{U-\ell+1}(U-k)(U-k-1)$. Actually, for each event type $X$, we have $RW$ events, so the total number of runs should be $\#Runs = (RW)^3 \times \frac{1}{2}\sum_{k=1}^{U-\ell+1}(U-k)(U-$

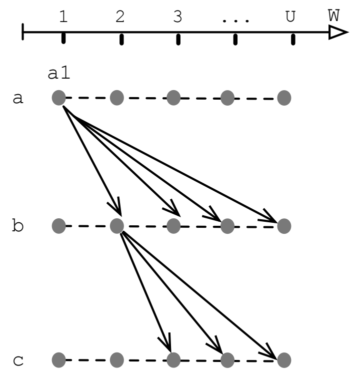


**Figure 3.5.** Enumeration for a particular point $a_1$.

$k-1)$. When we use the Big-O notation, the number of runs would be $O((RWU)^3)$, which is in the same order as Case 1.

Next we would consider more complex cases by increasing the time window and the pattern length. While the time window increases from $U$ to $\ell U$, the number of runs would increase, because we can arrange the events with less overlap. However, at the maximum the number of runs would be exactly the same as in Case 1 above. As we increase the number of pattern components, we can use a similar enumeration as in the Figure 3.5, and obtain the same conclusion. We can prove by induction that for a query with $\ell$ components under the second case, the complexity is $(RWU)^\ell$ as the following:

*Proof. Base case.* We have already seen when $\ell = 3$, the complexity is $\#Runs = (RWU)^3$.

*Induction.* We assume for a query with $\ell$ components, the complexity is $(RWU)^\ell$. When the query has $\ell+1$ components, for each run, it can match with at most $RW(U-\ell)$ events of the $(\ell+1)$-th type. So using the Big-O notation, the complexity should be bounded by $O((RWU)^\ell \times (RWU))$, that is, $O((RWU)^{\ell+1})$. □

The analysis for the event based framework is simpler than the point based framework because we do not need to count the number of point matches. For each component, the event based framework has $RW$ choices so the complexity can be calculated as $\#Runs = (RWU)^\ell$.

#### 3.5.1.3 Case 3: Pattern Components are Not Mutually Exclusive

When the components of a pattern are not mutually exclusive, an event may match multiple components. Let us use $S$ to denote a set of pattern components which can share events, and $|S|$ to denote the size of $S$. Let us use $K$ to denote the number of such sets of a query pattern. Obviously, we have $K \leq \ell$ and $\Sigma_{i=1}^{K}|S_i| = \ell$. An event can match all the components of one set, and only one set.

First, we consider the case that a set $S_i$ contains only consecutive pattern components. The worst case occurs in the following scenario: there is no overlap between events that match different sets; for those events that match the components in the same set, they completely overlap, i.e., having the same start and end timestamps. So when we create runs, for the events in set $S_i$, we only need to randomly pick up $|S_i|$ events to form a permutation. In the following analysis, we would use $< n, k >$ to denote the permutation of choosing k from n. The maximum number of runs for the point based framework would be:

$\#Runs = U^{\ell}\Pi_{i=1}^{K} < |S_i|RW, |S_i| > = U^{\ell}\Pi_{i=1}^{K}(\frac{(|S_i|RW)!}{(|S_i|RW-|S_i|)!})$.

When it comes to the event based framework, we can remove the $U^{\ell}$ part of the cost because there is no need to enumerate the points of each event. So the maximum number of runs for the point based framework would be:

$\#Runs = \Pi_{i=1}^{K} < |S_i|RW, |S_i| > = \Pi_{i=1}^{K}(\frac{(|S_i|RW)!}{(|S_i|RW-|S_i|)!})$.

Then we consider another case where a set can contain non-consecutive pattern components in the set $S_i$, e.g., the first, and the fifth components. We make a claim here for the worst case. And we show the proof for this claim in 3.13.

**Claim 3.5.1.** *The maximum number of runs occurs when the events are distributed evenly across the query components.*

When a set $S_i$ contain more than two non-consecutive subsets, the method to divide the events is the same. The proof is also the same so we omit the details here. Then each subset can be seen as a new set $S_j, j = 1, \ldots, K'$, and the problem is transformed into the former case where a set can only contain consecutive components. So the complexity would be in the same form as the former case, but instead uses $K'$ which denote the number of new sets.

### 3.5.2 Confidence Computation

We next analyze the complexity of confidence computation in the match collapsing step after pattern matches are generated.

#### 3.5.2.1 Skip Till Any Match Queries

Point-based framework: In this framework, for each event match $m$, the worst case occurs when all possible combinations of the points of the events in $m$ can form a point match. So for the event match $m$, the computation cost includes computing the confidence for each point match, for which we need $\ell$-1 multiplications, and after that summing up the confidence of the point matches using $U^\ell - 1$ additions. So, the cost for each point match is $U^\ell(\ell-1) + U^\ell - 1 = U^\ell \ell - 1$, that is, $O(U^\ell \ell)$.

Event-based framework: The difference between the two frameworks is that the event-based framework needs to enumerate the point-matches first, the cost of which is $U^\ell$. So for each match, the cost of confidence computation is: $U^\ell + U^\ell(\ell-1) + U^\ell - 1$ $= U^\ell + U^\ell \ell - 1$, that is, $O(U^\ell \ell)$.

#### 3.5.2.2 Skip Till Next Match Queries

For the skip-till-next-match, we need to consider the intervening sets when we compute the confidence. We assume the average intervening set is $\acute{S}_i$, and the size of the intervening set is $|\acute{S}_i|$.

*Point-based framework:* First we consider the case that no events can match multiple events, i.e. $\bar{S} = \emptyset$. For each event match, the cost of pre-computation (Line 10 -19 in Algorithm 3) is $O(U^2 \ell |\acute{S}_i|)$ , and the cost for the confidence computation (Line 20 - 26 in Algorithm 3) is $O(U^\ell \ell)$. When $\bar{S} \neq \emptyset$, we need to enumerate all possible orders as $O$ (Line

|  | Case 1 | Case 2 | Case 3 |
|---|---|---|---|
| Point-based framework | $O((RWU)^{\ell})$ | $O((RWU)^{\ell})$ | $O(U^{\ell}\frac{(\ell RW)!}{(\ell RW-\ell)!})$ |
| Event-based framework | $O((RW)^{\ell})$ | $O((RW)^{\ell})$ | $O(\frac{(\ell RW)!}{(\ell RW-\ell)!})$ |

**Table 3.1.** Complexity of pattern matching

|  | Skip till *any* match | Skip till *next* match |
|---|---|---|
| Event-based framework | $O(U^{\ell}\ell)$ | $O(U^2\ell\lvert\acute{S}_i\rvert + U^{\ell}\ell)$ when $\bar{S} = \emptyset$ <br> $O(U^{\lvert\bar{S}\rvert}(U^2\ell\lvert\acute{S}_i\rvert + U^{\ell}\ell))$ when $\bar{S} \neq \emptyset$ |
| Event-based framework | $O(U^{\ell}\ell)$ | $O(U^2\ell\lvert\acute{S}_i\rvert + U^{\ell}\ell)$ when $\bar{S} = \emptyset$ <br> $O(U^{\lvert\bar{S}\rvert}(U^2\ell\lvert\acute{S}_i\rvert + U^{\ell}\ell))$ when $\bar{S} \neq \emptyset$ |

**Table 3.2.** Complexity for Confidence Computation

2 in Algorithm 3), and the size of $O$ is $U^{|\bar{S}|}$. The total cost of the confidence computation for the point based framework is:

$$\begin{cases} O(U^2\ell|\acute{S}_i| + U^{\ell}\ell) & \bar{S} = \emptyset \\ O(U^{|\bar{S}|}(U^2\ell|\acute{S}_i| + U^{\ell}\ell)) & \bar{S} \neq \emptyset \end{cases}$$

*Event-based framework:* For event-based framework, it needs an extra cost of $O(U^{\ell})$ to enumerate all point matches first. This, however, does not increase the complexity in the Big-O notation, which stays the same as the point based framework.

Summary: Finally, we summarize the complexities of pattern matching and confidence computation in Table 3.1 and Table 3.2, respectively.

## 3.6 Optimizations

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

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. 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 set as follows:

- 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 ).

- Many stream systems use heartbeats [43] or punctuations [29, 47] to indicate that all future events (or those of a particular type) will have a timestamp greater than $\tau$. 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 $e$.

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 [34], 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.

We note that while we borrow the idea of selectivity based optimization from [34], the approach in that work does not handle events with imprecise timestamps. Hence it needs to use our $next()$ or $ext()$ function to support temporal uncertainty. This approach is further restricted to tree-structure query plans (that is, the evaluation order has to obey the tree structure), whereas our any state evaluation can start and continue the evaluation from any state and hence is more flexible.

We finally note that the two optimization approaches can improve performance when they are used properly. The sorting approach works well when the pattern components require mutually exclusive events, whereas the selectivity based approach works well if the selectivities of different pattern components vary significantly. We compare these two optimizations empirically in Section 3.7.

## 3.7 Performance Evaluation

We have implemented both evaluation frameworks using the SASE pattern query engine [3]. In this section, we evaluate these frameworks using both synthetic data streams with controlled properties and real traces collected from MapReduce cluster monitoring and RFID-based object tracking.

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.

### 3.7.1 Microbenchmarking using Synthetic Streams

We implemented an event generator that creates a stream of events of a single attribute. The events form a series of increasing values from 1 to 1000 and once reaching 1000, wrap around to start a new series. Events arrive in increasing order of time $t$ but each have an uncertainty interval $[t - \delta, t + \delta]$, with $\delta$ 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 $\ell$ (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 $v_j$ $(1 \leq j \leq \ell)$.

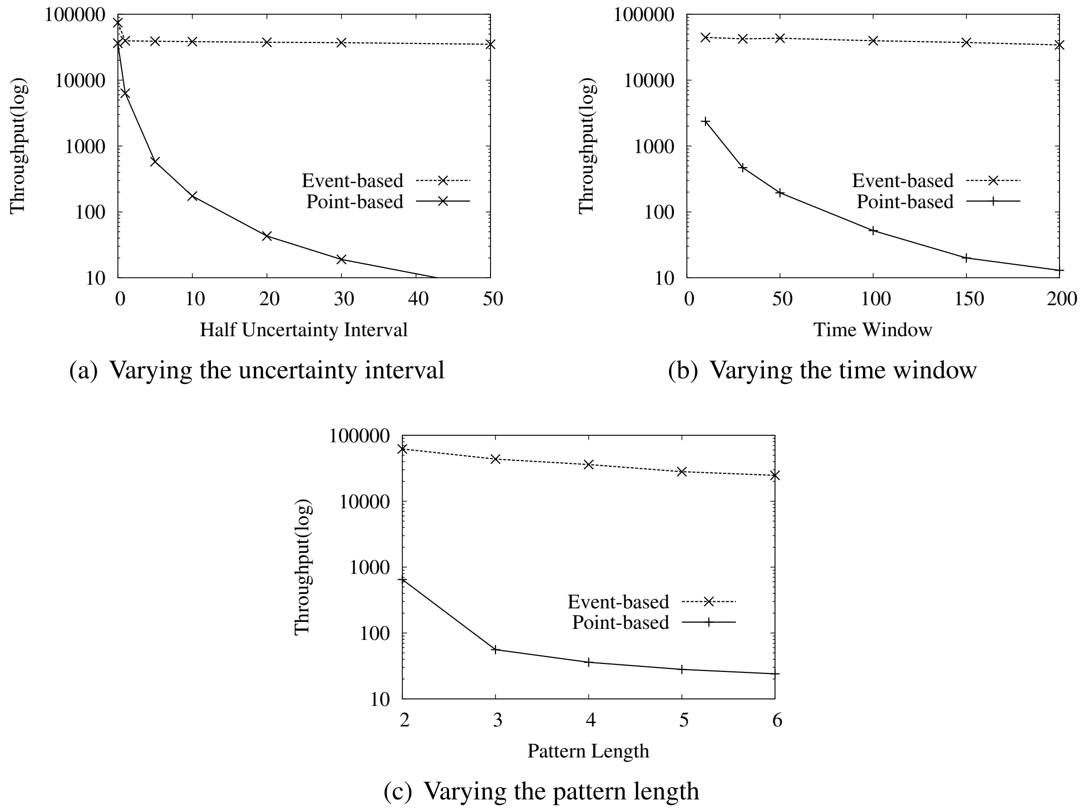


**Figure 3.6.** Comparing the point-based framework and the event-based framework using "skip till any match" queries and synthetic event streams.

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 $\delta$ from 1 to 50. Fig. 3.6(a) shows that the point-based method degrades its performance fast because as $\delta$ 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 $\delta$ 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 $\ell$ values.

To study of the effect of the window size $W$, we increase it from 10 to 200 units. Fig. 3.6(b) shows the results about the point-based and event-based frameworks, demonstrating the performance benefits of the latter. We then consider the effect of the pattern length $\ell$. As we vary $\ell$ from 2 to 6, we also adjust predicate selectivity so that longer patterns still obtain matches. Fig. 3.6(c) shows that while both methods are sensitive to $\ell$, the point-based method suffers much severe performance penalty.

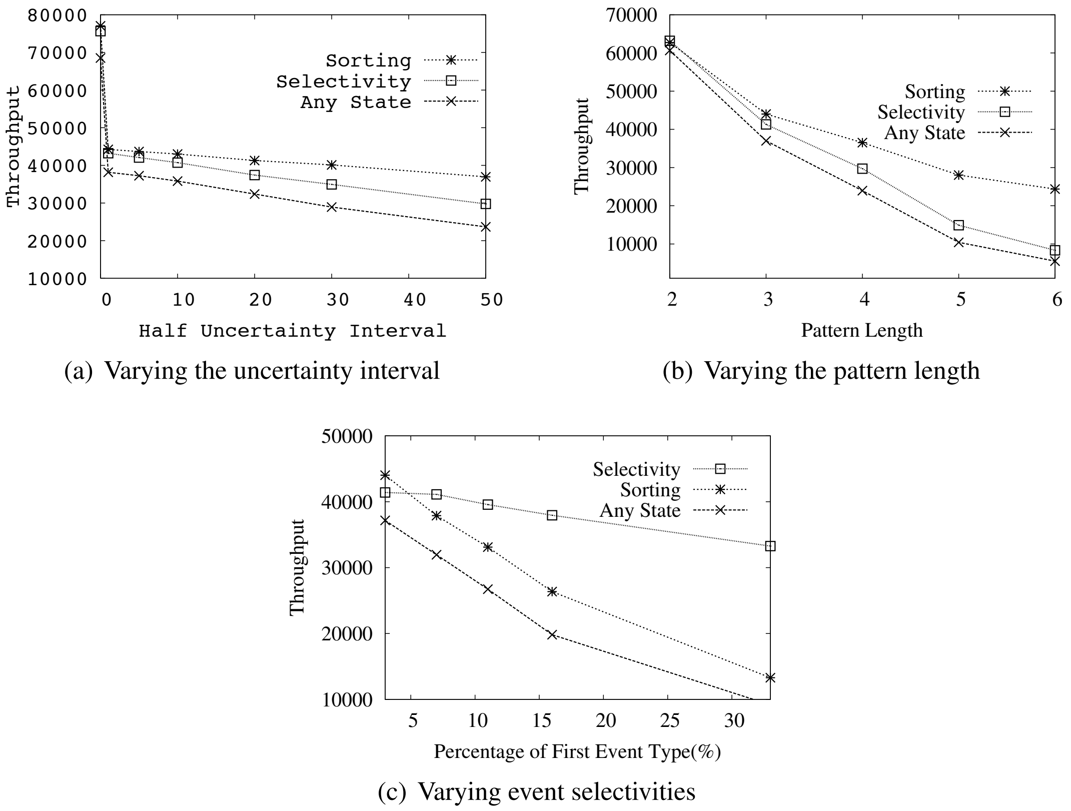


**Figure 3.7.** Comparing the basic event-based algorithm and optimizations using "skip till any match" queries and synthetic streams.

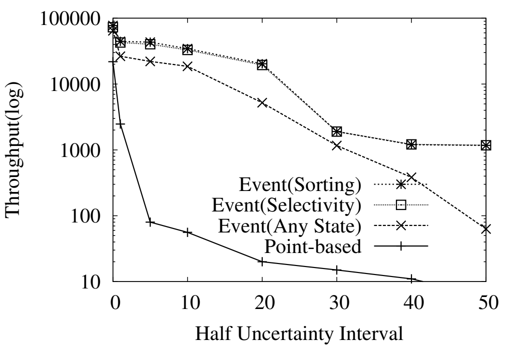
Optimizations of the Event based Method (skip till any match): We next evaluate the two optimizations, sorting for query order evaluation and selectivity order evaluation, for enhancing the basic event-based evaluation method, called the any state method.

Fig. 3.7(a) shows the results with varied $\delta$. The performance of the any state method degrades linearly with $\delta$. This is because as $\delta$ 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 $E_\ell$, 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.

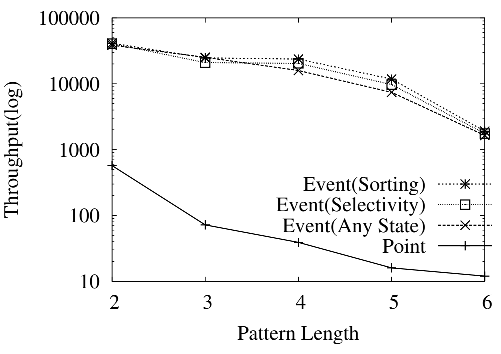
Fig. 3.7(b) compares these methods as the pattern length $\ell$ is increased. The any state method loses performance quickly. Since a run can start by matching any pattern component in this method, a longer pattern means a higher chance for an event to match a component and start a run. Sorting still 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 $E_\ell$ accordingly. As a result, more events can match $E_1$ and fewer can match $E_\ell$. Fig. 3.7(c) 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 for most points tested, because it can remove outdated events from the buffer of $E_1$ before it sees events in other buffers, hence avoiding many runs.

We also note that when the percentage of the first event type is less than 5%, the trend between the two optimizations is reversed. This shows the case when the selectivity-based method does not work as well as the sorting-based method. Since the most selective events come the earliest, both methods benefit from starting pattern matching from the most selective component. At the same time, the sorting-based method generates fewer runs because it strictly follows the query order whereas the selectivity-based method still uses any state evaluation. For example, if we have three events $a_1, b_2, c_3$ as shown in Fig.3.3(d), the sorting-based method will only create 3 runs: $(a_1, -, -)$, $(a_1, b_2, -)$ and $(a_1, b_2, c_3)$, while the any state evaluation method will generate 7 runs, because it will keep partial runs like $(-, -, c_2)$ in memory to match with future $a$'s and $b$'s.



(a) Varying the uncertainty interval



(b) Varying the sequence length

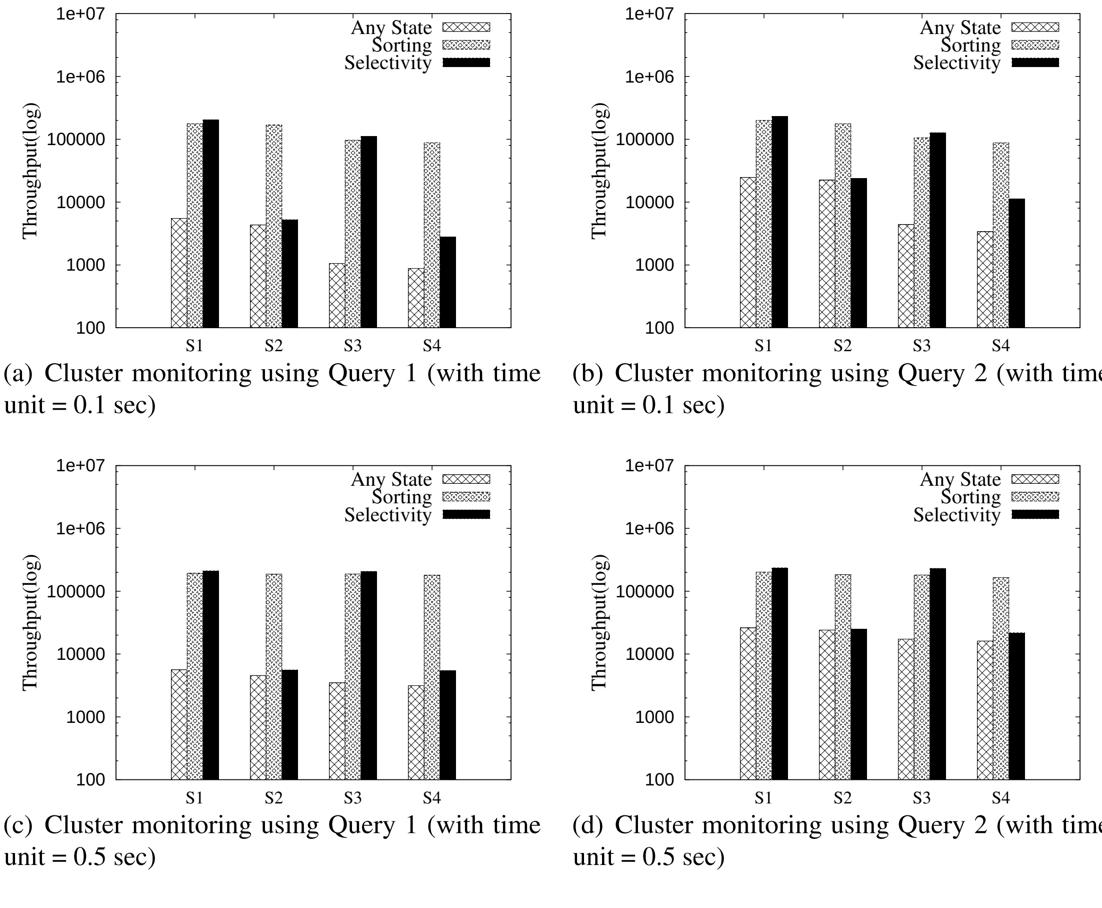
**Figure 3.8.** Comparing the point based and event based algorithms for "skip till next match" queries.

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. 3.8(a) shows the results as $\delta$ is varied. Compared to

Fig. 3.6(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 $\delta$ 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 $\delta \geq 30$.

We further evaluate the performance of our two frameworks for skip till next match queries as the pattern length, $\ell$, varies. Fig. 3.8(b) shows the results as $\ell$ increases: Again, the event-based methods work better than the point-based method. However, with increased $\ell$, the cost of confidence computation also increases fast, which also causes the event-based methods to degrade the performance.

### 3.7.2 Performance Evaluation in MapReduce Cluster Monitoring



(a) Cluster monitoring using Query 1 (with time unit = 0.1 sec)

(b) Cluster monitoring using Query 2 (with time unit = 0.1 sec)

(c) Cluster monitoring using Query 1 (with time unit = 0.5 sec)

(d) Cluster monitoring using Query 2 (with time unit = 0.5 sec)

**Figure 3.9.** Performance results using real traces in MapReduce cluster monitoring.

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 UNIX time) 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 [21] 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 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 of different sizes in our experiments. More specifically, we rounded all Hadoop timestamps using 0.1 or 0.5 second as a time unit. We then 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.

We used four monitoring queries in this case study. The first query is Query 1 in §3.2. The other queries have the same structure but replace the TaskStart/Finish events with MergeStart/Finish, PullStart/Finish and RequestStart/Finish events in Query 2, Query 3 and Query 4, respectively. 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 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. 3.9(a) shows the results for Query 1. 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. Fig. 3.9(b) presents the throughput results of Query 2 and shows similar trends as Query 1.

Fig. 3.9(c), 3.9(d) show the results of Query 1, Query 2 using 0.5 second as the time unit. By comparing Fig. 3.9(a) with Fig. 3.9(c), Fig. 3.9(b) with Fig. 3.9(d), we can see for S1 and S2 (both using the skip till any match strategy), the performance is improved only slightly. For S3 and S4 (both using the skip till next match strategy), the performance is improved more significant (which, however, may not look as pronounced in the plot due to the use of log scale on the y-axis). This is because when we increase the time unit, the uncertainty interval length becomes smaller.

### 3.7.3 Performance Evaluation in RFID-based Object Tracking

We next evaluate our techniques in the second case study, RFID-based object tracking in the hospital setting. Hospitals are busy environments and many medical devices need to appear in different locations in a particular order. For example, some medical tools need to be disinfected in several steps by different machines at different locations in a predefined

|  | $P$ | $f$ | $U$ | $P_n$ |
|---|---|---|---|---|
| Setting1 | 0.9 | 1 | 10 | $1 \times 10^{-10}$ |
| Setting2 | 0.8 | 1 | 14 | $1.6 \times 10^{-10}$ |
| Setting3 | 0.7 | 1 | 19 | $1.2 \times 10^{-10}$ |

**Table 3.3.** Experiment settings used in the RFID case study.

order. If the tools to be disinfected went through different locations or machines in a wrong order, the tools might not be ready to use. Another possible case is the tools may be accepted for use for all orders except some special orders. RFID technology can help assist medical personnel in tracking such medical tools. An RFID tag can be attached to each item to be tracked, and RFID readers can be deployed at the locations (machines) where the items need to visit.

For such object tracking applications, a useful query might look like the following, which is used in our experiment:

```
Query 5:
PATTERN SEQ(ObjectAppear a, ObjectAppear b, ObjectAppear c)
WHERE   a.itemId = b.itemId AND
        b.itemId = c.itemId AND
        a.location = Location1 AND
        b.location = Location2 AND
        c.location = Location3 AND
        skip_till_next_match(a, b, c)
WITHIN  10 minutes
RETURN  a, b, c
```

As we mentioned in the introduction, raw RFID data is known to be lossy and even misleading, Meaningful events such as object movements are often derived using probabilistic inference. The actual occurrence time of object movement is unknown and can only be estimated to be in a range with high probability. Another issue is that if an object moves fast, then the inferred results might be delayed due to the fixed read frequency of an RFID reader. In our experiment, the uncertainty interval was set based on the read frequency and the probability of missed readings. Let us use $f$ to denote the read frequency, i.e. $f$ readings per second, and $p$ to denote the probability of reading an object in the range of an RFID reader. Then the probability that the reader misses the object consecutively for $n$ times is $P_n = (1-p)^n$. If $n$ is very large, $P_n$ would be very small. So we can set the uncertainty interval to $U = n/f$ with a sufficiently large $n$ such that $P_n$ becomes very small. In our experiment, we used three setting as shown in Table 3.3.

We obtained the data trace from a simulated hospital environment by adapting the simulator in [11]. Our data trace contains 300,000 events for 90 medical tools scanned in 8 locations. We ran the above query using our event-based framework. We ran the experiments using three algorithms: the any state evaluation method, the any state evaluation with the sorting optimization, the any state evaluation with the selectivity based optimization.

Fig. 3.10 shows the performance of the three algorithms. We can see the throughput of the any state evaluation is quite good even without optimizations. The reason is that the arrival rate for a particular item is not as high as that in the cluster monitoring use case. The optimizations further improve the performance. In this experiment, the selectivity-based optimization works better because it can remove many events before processing them in pattern matching. These removed events indicate that the object may have never visit some of the locations specified in the query, or may have visited all the locations but in the duration that exceeds the time window. Finally, as we increase the uncertainty interval, the performance of all three methods degrades somewhat, similar to the results of the experiments using synthetic data. However, with optimization, the throughput of our best algorithm remains high, e.g., over 80,000 events/second for the lowest read rate of 70% and the largest uncertainty interval size of 19.

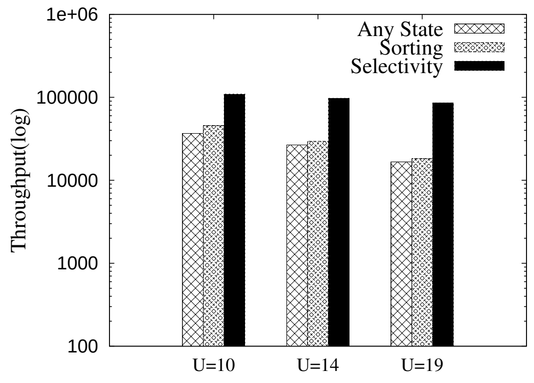


**Figure 3.10.** Performance of event-based algorithms for RFID object tracking.

## 3.8 An Extended Discussion

Handling temporal uncertainty in event streams is related to handling temporal uncertainty in temporal databases. In this section, we provide an extended discussion of the related work in temporal databases [17] by emphasizing the differences from our work in both semantics and complexity.

Semantics: We first introduce the semantics of query processing under temporal uncertainty proposed in [17]. In a traditional database, an SQL query posed to a database has a single interpretation if the database contains complete information. Under temporal uncertainty, existing work [17] offers two interpretations of an SQL query posed to a database that contains incomplete information: (1) One interpretation is that the query selects information that *possibly* matches the retrieval constraints. (2) The other interpretation is that the query selects information that *definitely* matches the retrieval constraints. Using the concept of *completions* of a tuple with an uncertain timestamp, the definite interpretation selects only those tuples that are selected based on the SQL semantics in every completion of the tuple. The possible interpretation differs in that it selects only those tuples that are selected in some completion of the tuple. Finally, the semantics of query processing can be specified by *ordering plausibility*, which means that the percentage of completions of a tuple that could be selected based on the SQL semantics should pass a query specified threshold.

As can be seen, our formal semantics based on the possible words (in Section 3.2) is much easier to follow and does not require a user-specified threshold.

Complexity: After introducing the query semantics under temporal uncertainty in temporal databases, we next use the example in Fig. 3.1 to show the computation complexity if we apply the techniques in temporal databases to our queries. We discuss queries using the two event selection strategies in turn.

*Skip till any match queries.* We can rewrite in SQL the query in Fig. 3.1 when it uses the skip till any match strategy:

```
SELECT  SA, SB, SC
FROM    StreamA SA, StreamB SB, StreamC SC
WHERE   SA.type = A AND SB.type = B AND SC.type = C AND
        SA.ts < SB.ts AND SB.ts < SC.ts
```

According to the SQL semantics, we first apply the filters “SA.type = A and SB.type = B and SC.type = C” and then compute the cross-product of SA, SB, and SC. In general, if there are $\ell$ inputs of the query and each input contains $RW$ tuples matching the specified event type, the complexity of the cross-product is $(RW)^{\ell}$. Then for each result tuple of the cross-product, there are $U^{\ell}$ completions that need to be checked against the temporal predicates, “SA.ts $<$ SB.ts and SB.ts $<$ SC.ts”, in the WHERE clause (here we assume that every tuple has an uncertainty interval of size $U$ for simplicity). So the total complexity is $O((RWU)^{\ell})$, the same as in the worst case of our point-based framework.

Recall from Section 3.5 that in the worst case of the point-based framework, we assume that each possible world would return a match; that is why we have to go over each possible world. In practice, many possible worlds do not contain a match because the events in these worlds do not satisfy the ordering constraint specified in the query. Then the point-based framework can avoid visiting these possible worlds where events are not ordered properly. So the point-based framework performs better than the approach in temporal databases in practice.

*Skip till next match queries.* Given a skip till next match query, we have to rewrite it in SQL using nested queries:

```
SELECT  SA, SB, SC
FROM    StreamA SA, StreamB SB, StreamC SC
WHERE   SA.type = A AND SB.type = B AND SC.type = C AND
        SA.ts < SB.ts AND SB.ts < SC.ts AND
        SB.ts <= ALL (SELECT SB2.ts
                      FROM StreamB SB2
                      WHERE SB2.type = B AND SA.ts < SB2.ts)
                      AND
        SC.ts <= ALL (SELECT SC2.ts
                      FROM StreamC SC2
                      WHERE SC2.type = C AND SB.ts <SC2.ts)
```

Implementing the 'definite' semantics requires that a query answer hold in every completion (possible world). Implementing the 'possible' semantics requires that the query answer hold in at least one completion (possible world). To support 'ordering plausibility', the total probability of all completions (possible worlds) where the query answer holds should be greater than the threshold specified in the query.

However, this query includes two nested queries. Existing work [17] supports only basic SELECT-FROM-WHERE queries and does not specify how to extend to nested queries. In contrast, both of our proposed evaluation frameworks support skip till next match queries by extending the algorithms for skip till any match queries using the NELT mechanism (as detailed in Section 3.3).

## 3.9 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 in case studies of MapReduce cluster monitoring and RFID-based object tracking as well as under a wide range of synthetic workloads.

## 3.10 Notational Convention

Table 3.4 summarizes the notation used in this chapter.

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

**Table 3.4.** Notation used in this chapter

## 3.11 Point-based Evaluation

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

### 3.11.1 Correctness Proofs

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

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

Skip till next match: We next prove the correctness 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 $(e_{m_1}^{t_1}, \ldots, e_{m_j}^{t_j})$ $(j \geq 1)$, with $e_{m_j}$ being the last selected event. We prove that the following statements are true:

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

(2) Every point event, $e_i^t$, that can potentially match the pattern component $E_{j+1}$ and starts **before or at** $e_{m_j}$'s NELT, can actually be used to extend the partial match $(e_{m_1}^{t_1}, \ldots, e_{m_j}^{t_j})$ in a possible world. 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 $e_{m_j}$

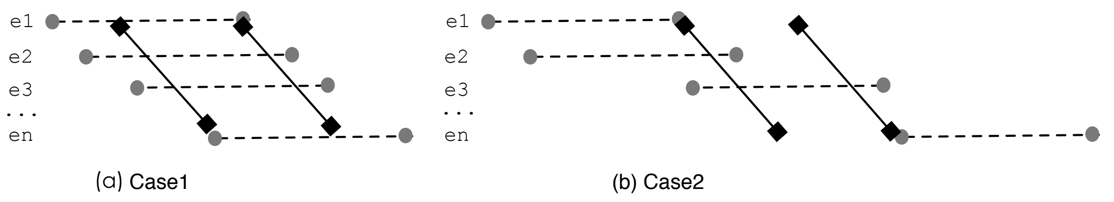


**Figure 3.11.** Two cases of constructing point-matches

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). □

## 3.12 Event-based Evaluation

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

### 3.12.1 Query Order Evaluation

In this part we would discuss the three pass algorithm and the incremental algorithm.

#### 3.12.1.1 Three Pass Algorithm

To prove the correctness of our three pass algorithm (in §3.4.1), 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 $e_{n+1}$.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 $e_{n-1}$.vub $= min(e_n$.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. 3.11(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., $e_i$.vlb $\leq e_1$.vub $+ i - 1 \leq e_i$.vub. We can show this by contradiction: Assume that $e_1$.vub $+ i - 1$ is out of $e_i$'s valid range. Then $e_1$.vub $+ i - 1 >$ $e_i$.vub or $e_1$.vub $+ i - 1 < e_i$.vlb. In the former case, it will contradict with the valid upper bound definition. In the latter case, we can get $e_1$.vub $+ i - 1 < e_i$.vlb $\leq e_n$.vlb $- n + i$, then we can get $e_n$.vlb $- e_1$.vub $\geq n - 1$, which contradicts the case condition.

Case 2: $e_n$.vlb $- e_1$.vub $\geq n - 1$, as shown in Fig. 3.11(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 $t_i = min(e_i$.vub$, e_n$.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 $[e_1$.vub $+ i - 1, e_n$.vlb $- n + i]$. We can show this by contradiction. Assume that there is no overlap. Then $e_i$.vlb $> e_n$.vlb $- n + i$ or $e_i$.vub $< e_1$.vub $+ i - 1$. According to the definition of the valid lower bound and valid upper bound, $e_i$.vlb $\leq e_n$.vlb and $e_i$.vub $\geq e_1$.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 $[e_1$.vub $+ i - 1, e_n$.vlb $- n + i]$. Then we need to show that $t_i > t_{i-1}(1 < i < n)$. If $t_i = e_n$.vlb $- n + i$, then $t_i$ is larger than all points during $[e_{i-1}$.vub $+ i - 1, e_n$.vlb $- n + i - 1]$, and so $t_i > t_{i-1}$. If $t_i = e_i$.vub, then we need to consider two cases: if $t_{i-1} = e_{i-1}$.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. □

#### 3.12.1.2 Incremental Algorithm

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. We would prove the proposition as following:

**Proposition 3.12.1.** The incremental algorithm can obtain the same results as the three-pass algorithm when it evaluates the same query over the same event stream.

The three-pass method is to collect all events for a run, and then compute the valid lower bounds and valid upper bounds. Let us use $3pass(e_j.\text{vlb}/\text{vub})$ to denote the vlb/vub of $e_j$ computed by the three-pass method.

The incremental method is to compute the valid lower bounds and valid upper bounds whenever the run selects a new event. The advantage of this method is that we can end an unqualified run as early as possible. Let us use $Incr(e_j.\text{vlb}/\text{vub})$ to denote the $vl/\text{vub}$ of $e_j$ computed by the incremental method.

In order to prove the equality of the two methods, we would prove that the two methods can get the same valid lower bounds and the same valid upper bounds.

Valid lower bound: At first, we would prove the equality of valid lower bounds. This part is straightforward, because the computation processes for the valid lower bound of the two methods are the same. Start from the first event, and then go over events one by one in the query order using the following definition:

$e_j$.vlb $= max(e_{j-1}.\text{vlb} + 1, e_j.\text{lower})$

Since in both methods we start the computation from the first event, so that means $3pass(e_1.\text{vlb}) = Incrl(e_1.\text{lower})$. In the following computation of the valid lower bound for $e_j(j > 1)$, both methods would use the same definition. So the valid lower bounds of the two methods would be the same.

Valid upper bound: Definition of the valid upper bound: $e_j.\text{rub} = min(e_{j+1}.\text{rub} - 1, e_j.\text{upper})$

The computation processes for the valid upper bounds of the two methods are different. We need to prove the following two statements:

**1).** The temporary(before we select all events for the match) valid upper bounds in the incremental method would not be smaller than the final valid upper bounds in the three-pass method, such that our early pruning of runs would not lead to loss of results.

**2).** The final valid upper bounds of the incremental method would be the same as that of the three-pass method, such that the final results are correct.

Proof of Statement 1): The intuition of this proof is that when we see later events we are always keeping or shrinking the revised upper bounds, so the revised upper bounds should be non-increasing.

*Proof.* Since the valid upper bounds are based on revised upper bounds, we want to first prove the revised upper bounds are the same in two methods. We can use contradiction to prove this statement. We assume $Incr(e_j.\text{rub}) < 3pass(e_j.\text{rub})$. Currently we have $n$ events, and the query length is $\ell$. If it is $Incr(e_j.\text{rub}) < 3pass(e_j.\text{rub})$ , according to the definition of rub, we can get $Incr(e_{j+1}.\text{rub}) < 3pass(e_{j+1}.\text{rub})$, and repeat this, we can get $Incr(e_n.\text{rub}) < 3pass(e_n.\text{rub})$, which does not hold due to the following reasons:

If $n < \ell$, because $Incr(e_n.\text{rub}) = e_n.\text{upper}$, $3pass(e_n.\text{rub}) = min(e_{n+1}.\text{rub} - 1, e_n.\text{upper})$, it means $e_n.\text{upper} < min(e_{n+1}.\text{rub} - 1, e_n.\text{upper})$.

If $n = \ell$, $Incr(e_n.\text{upper}) = 3pass(e_n.\text{upper}) = e_n.\text{upper}$.

So the temporary rub would not be smaller than the final rub. Then, we can apply the third pass. If $e_\ell.\text{rub} \leq T_m$, then we are done because the valid upper bounds are the same as the revised upper bounds. If $e_\ell.\text{rub} > T_m$, we will update $e_\ell.\text{vub} = T_m$, and go back to update the vubb of each event. If $n = \ell$, the computation is the same. If $n < \ell$ , then $Incr(e_n.\text{rub}) = min(e_n.\text{upper}, T_m)$, $3pass(e_n.\text{vub}) = min(e_n.\text{upper}, e_{n+1}.\text{vub})$. According to $e_{n+1}.\text{vub} \leq T_m$, we can get $Incr(e_j.\text{rub}) \geq 3pass(e_j.\text{rub})$. □

Proof of Statement 2): The intuition of the proof is to prove that the last event's revised upper bound is the same by two methods. Because the previous events' revised upper bounds are based on their last event, we can conclude all revised upper bounds are the same. If the revised upper bounds are proved to be the same, then obviously we can derive the same valid upper bounds from the two methods.

*Proof.* As we proved above, the revised upper bounds would not be less than the final revised upper bounds. Here we want to prove all the final revised upper bounds are the same. Obviously the last event's revised upper bounds should be the same, because they are the upper bound of the last event. Next we will also use proof by contradiction to prove the equality of the final rub. We assume $Incr(e_j.\text{rub}) > 3pass(e_j.\text{rub})$for a certain event $e_j(1 \leq j < \ell)$.

Based on the definition, we know:

- $Incr(e_j.\text{rub}) = min(Incr(e_j.currentRub), Incr(e_{j+1}.\text{rub}) - 1)$
- $3pass(e_j.\text{rub}) = min(e_j.\text{upper}, 3pass(e_{j+1}.\text{rub}) - 1)$ .

Next we discuss the relationship between $Incr(e_j.\text{rub})$ and $3pass(e_j.\text{rub})$ by two different cases.

$Incr(e_j.\text{rub}) = e_j.currentRub$. According to our assumption, $Incr(e_j.\text{rub}) > 3pass(e_j.\text{rub})$, and we know that $Incr(e_j.currentRub) \leq e_j.\text{upper}$, so $3pass(e_j.\text{rub})$ has to be equal to $3pass(e_{j+1}.\text{rub}) - 1$. Combining the definition and the assumption, we get $Incr(e_{j+1}.\text{rub}) - 1 > Incr(e_j.\text{rub}) > 3pass(e_{j+1}.\text{rub}) - 1$, thus $Incr(e_{j+1}.\text{rub}) > 3pass(e_{j+1}.\text{rub})$. So we get an induction on the relationship. Repeat this induction, we can finally get $Incr(e_\ell.\text{rub}) > 3pass(e_\ell.\text{rub})$, which contradicts with an easy conclusion we said above.

$Incr(e_j.\text{rub}) = Incr(e_{j+1}.\text{rub}) - 1)$. In this case, we get $Incr(e_{j+1}.\text{rub}) - 1 < Incr(e_j.currentRub) < e_j.\text{upper}$, so again $3pass(e_j.\text{rub})$ has to be equal to $3pass(e_{j+1}.\text{rub}) - 1$ due to our assumption. And we get $Incr(e_{j+1}.\text{rub}) - 1 > 3pass(e_{j+1}.\text{rub}) - 1$, which is the same as the other case. So we can get the same contradiction.

□

#### 3.12.1.3 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. □

### 3.12.2 The Any State Evaluation Method

*Proof. The any state evaluation method can obtain the same results as the query order evalulation method when it evaluates the same query over the same event stream.* □

*Proof.* First we need to show that by using the any state method, we still can capture the order of two events correctly in pattern matching (i.e., in query). Our ext function decides the order of two events according to the query order and their uncertainty intervals, so the arrival order or the order in which a run selects events in the any state method would not affect the results. So we can say that the any state method can capture the correct order between any two events.

Then we would prove the two methods would capture the same matches from two directions.

If we capture a match using the any state evaluation method, we can sort the events by query order, then we can use the query order evaluation method to find the same match. This is one direction for the proof.

In the other direction, we would use induction to show that if the query order evaluation method can find a match, the any state evaluation method also can find it.

Base case: When the sequence length is one, this statement is obvious to hold. When the sequence length is two, the match $(e_1, e_2)$ is found by the query order evaluation method. If the arrival order is $e_1, e_2$, then the any state evaluation method will have run $(e_1)$ after the first event arrives, and have runs $(e_1), (e_1, e_2), (e_2)$ after the second event arrives, then it can return the match $(e_1, e_2)$. If the arrival order is $e_2, e_1$, the run is $(e_2)$ after the first event arrives, and the run will be $(e_2), (e_1, e_2), (e_1)$ when it sees another event, also it can return the match $(e_1, e_2)$.

Induction:Then we can assume when the sequence length is $n$, if the query order evaluation method captures a match $(e_1, e_2, ...e_n)$, the any state evaluation can also capture the same match. When the sequence length is $n + 1$, the query order evaluation method capture the match $(e_1, e_2, ...e_n, e_{n+1})$. We assume $e_{n+1}$ is the $i$th $(1 \leq i \leq n + 1)$ event during the arrival sequence. So right after we select $e_{n+1}$, we already has a partial match $(e_{p_1}, e_{p_2}, ...e_{p_{i-1}})$. Then because $e_{n+1}$ can pass the predicate check, so the partial match can choose to select it, and at the same time clone the partial match. So after selecting $e_{n+1}$, among the active runs, we have $R_a(e_{p_1}, e_{p_2}, ...e_{p_{i-1}})$ and $R_b(e_{p_1}, e_{p_2}, ...e_{p_{i-1}}, e_{n+1})$. Then in the following steps, the two runs can have the same operation on events. According to our assumption, $R_a$ can grow to $(e_1, e_2, ...e_n)$, so $R_b$ can grow to $(e_1, e_2, ...e_n, e_{n+1})$. so the any state evaluation method return the same match. $\square$

For skip till next match, after we get the match, we can use NELT as a postponed operation, then it would be the same as that in query order evaluation, so the results will be no different.

## 3.13 Complexity

In this section, we provide complexity related proofs.

### 3.13.1 Proof for Case 1

We define the arrangement for Case 1 before we prove it, which is shown in Fig. 3.4:

For pattern $(E_1, E_2, \dots, E_l)$, if $e_x \in E_i$ and $e_y \in E_j$, $(1 \leq i \leq j \leq l)$, then $e_x$.vub $<$ $e_y$.vlb.

We will use induction on the query length to show that the largest number of runs occurs if the events are arranged as the above.

*Proof. Base case.* When the query contains only one component, this arrangement can lead to the largest number of runs because all possible arrangements will lead to the same result.

Then we assume when the query length is $k$, this arrangement can lead to the largest number of runs.

*Induction.* When the query length is $k+1$, there are three possible cases on the $k+1$th component's position.

- If the $k+1$th component is before all the existing $k$ components, then events for the $k+1$th components should be arranged before all other events, then each event of the new component is possible to combine a existing k-long match to make a new match.
- If the If the $k+1$th component is after all the existing $k$ components, then events for the $k+1$th components should be arranged after all other events, then each event of the new component is possible to combine a existing k-long match to make a new match.
- If the If the $k+1$th component is between two of the existing $k$ components, after the $i$th component and before the $j$th component, $(1 \leq i < j \leq k)$, then events for the $k+1$th components should be arranged after all events belonging to the $i$th component, and before all events belonging to the $j$th component. Also, each event of the new component is possible to combine a existing k-long match to make a new match.

All the above three cases will lead to the largest number of runs, and they all follow the arrangement as we show in Fig. 3.4. □

### 3.13.2 Proof for Case 3

Here we show the proof for the following claim.

**Claim 3.13.1.** *The maximum number of runs occurs when the events are distributed evenly across the query components.*

If $S_i$ is consisted of two non-consecutive subsets $A$ and $B$. $A$ contains $|A|$ query components and $B$ contains $|B|$ query components. We want to show that if we assign $|A|RW$ events for $A$, $|B|RW$ events for $B$, we get the maximum number of runs.

*Proof.* When we consider how to divide events for $A$ and $B$, we only consider the runs that contain only the events matching $A$ and $B$. This would not affect the result because the number of runs without events matching $A$ and $B$ is fixed. The total number of runs would be the product of the two partial results. Now we use $\#Runs_{A,B}$ to denote the number of runs only containing events from $A$ and $B$. When $A$ has $|A|RW$ events and $B$ has $|B|RW$ events, $\#Runs^1_{A,B} = < |A|RW, |A| > \times < |B|RW, |B| > = \frac{(|A|RW)!}{(|A|RW-|A|)!} \times \frac{(|B|RW)!}{(|B|RW-|B|)!}$.

If we move $k$ events from $B$ to $A$, $(1 \leq k < |B|RW)$, $A$ will have $|A|RW+k$ events while $B$ has $|B|RW-k$ events. And the number of runs is: $\#Runs^2_{A,B} = < |A|RW+k, |A| > \times < |B|RW-k, |B| > = \frac{(|A|RW+k)!}{(|A|RW+k-|A|)!} \times \frac{(|B|RW-k)!}{(|B|RW-k-|B|)!} = \frac{(|A|RW)!}{(|A|RW-|A|)!} \times \prod_{i=1}^{k} \frac{|A|RW+i}{|A|RW+i-|A|} \times \frac{(|B|RW)!}{(|B|RW-|B|)!} \times \prod_{i=1}^{k} \frac{|B|RW-|B|-i+1}{|B|RW-i+1}$

$= \#Runs^1_{A,B} \times \prod_{i=1}^{k} (\frac{|A|RW+i}{|A|RW+i-|A|} \times \frac{|B|RW-|B|-i+1}{|B|RW-i+1})$.

In order to prove $\#Runs^1_{A,B} > \#Runs^2_{A,B}$, now we only need to prove:

$\prod_{i=1}^{k} (\frac{|A|RW+i}{|A|RW+i-|A|} \times \frac{|B|RW-|B|-i+1}{|B|RW-i+1}) < 1$. Next we will prove

$\frac{|A|RW+i}{|A|RW+i-|A|} \times \frac{|B|RW-|B|-i+1}{|B|RW-i+1} < 1$ for each $i$, $(1 \leq i \leq k)$.

$\frac{|A|RW+i}{|A|RW+i-|A|} \times \frac{|B|RW-|B|-i+1}{|B|RW-i+1} < 1$

$\iff (|A|RW+i)(|B|RW-|B|-i+1) < (|A|RW+i-|A|)(|B|RW-i+1)$

$\iff (|A|RW+i)(|B|RW-i+1) - |B|(|A|RW+i) < (|A|RW+i)(|B|RW-i+1) - |A|(|B|RW+1)$

$\iff -|B|(|A|RW+i) < -|A|(|B|RW-i+1)$

$\iff -|B|i < |A|(i-1)$

$\iff 0 < |B|i + |A|(i-1)$

Since $i \geq 1$, the formula is validated. So we can conclude that we get the maximum number of runs by distributing events evenly across the query components. $\square$

# CHAPTER 4

# COMPLEXITY AND OPTIMIZATION OF EXPENSIVE QUERIES IN COMPLEX EVENT PROCESSING

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 [32]. Besides these well-known applications, CEP is gaining importance in a number of emerging applications, which particularly motivated our work in this chapter:

Cluster monitoring: Cluster computing has gained wide-spread adoption in big data analytics. Monitoring a compute cluster, such as a Hadoop cluster, has become crucial for understanding performance issues and managing resources properly [8]. Popular cluster monitoring tools such as Ganglia [33] provide system measurements regarding CPU, memory, and I/O from outside user programs. However, there is an increasing demand to correlate such system measurements with workload-specific logs (e.g., the start, progress, and end of Hadoop tasks) in order to identify unbalanced workloads, task stragglers, queueing of data, etc. Manually writing programs to do so is very tedious and hard to reuse. Hence, the ability to express monitoring needs in declarative pattern queries becomes key to freeing the user from manual programing. In addition, many monitoring queries require the correlation of a series of events (using Kleene closure as defined below), which can be widely dispersed in a trace or multiple traces from different machines. Handling such queries as large amounts of system traces are generated is crucial for real-time cluster monitoring. (For more see §4.3.5.)

Logistics: Logistics management, enabled by sensor and RFID technology advances, is gaining adoption in hospitals [48], supply chains [32], 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 [11, 49].

**Challenges.** Among many challenges in CEP, this chapter 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 [3, 15, 34, 35, 41, 52] 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 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 [3, 34, 35, 52] 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 [55]: (*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 I shall show, techniques for handling imprecise timestamps [55] work only for simple pattern queries and quickly deteriorate for more complex queries.

Contributions: In this chapter, I perform a thorough analysis of pattern queries in CEP, with a focus on the fundamental understanding of which query features make them "expensive". More specifically, our contributions include:

*Runtime Complexity* (§4.1): I begin our study by addressing the question of which features of patten queries make them computationally more expensive. 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

| Symbol | Meaning |
|---|---|
| $l$ | Number of components in a SEQ pattern. |
| $k$ | Number of Kleene closure components in SEQ. |
| $W$ | Size of the time window. |
| $R$ | $R_i$ is the arrival rate of events satisfying the constraints on the $i^{th}$ component of a pattern. A simplifying assumption is: $R_1 = R2 = \ldots = R_l = R$. |
| $U$ | Size of the uncertainty interval for events with imprecise timestamps, assumed to the same for all. |
| $c_r$ | Average cost for a run, including the cost for run creation, event evaluation, etc. |
| $c_m$ | Average cost to compute the probability for a (point-based) match in the imprecise case. |
| $S_1, S_2, S_3$ | Event selection strategy of **Contiguity**, **Skip-till-next-match**, **Skip-till-any-match**, respectively. |

**Table 4.1.** Notation in runtime complexity analysis.

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).

Optimizations: (§4.2): To address bottleneck (1), I break query evaluation into two parts: pattern matching, which can be shared by many matches, and result construction, which constructs individual results. I propose a series of optimizations to reduce shared pattern matching cost from exponential to polynomial time (sometimes close-to-linear). To address bottleneck (2), I provide a dynamic programming algorithm to expedite confidence computation and to improve performance when the user increases the confidence threshold for desired matches.

Evaluation with a case study: (§4.3): I compare our new system with a number of state-of-the-art pattern query systems including SASE [3, 52], ZStream [34], and XSeq [35]. 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, I perform a case study in cluster monitoring using real Hadoop workloads, system traces, and a range of monitoring queries. I show that my system can automate cluster monitoring using declarative pattern queries, return very insightful results, and support real-time processing even for expensive queries.

## 4.1 Runtime Complexity

We begin our study by addressing the question of which features of patten queries make them computationally more expensive. This analysis, which we call "runtime analysis", follows this methodology: it shows how the runtime complexity changes as we add more

| # | **Language Features** | **Selection Strategy** | **Timestamp** | **Complexity Class in W** | **Formula** (using notation in Table 4.1) |
|---|---|---|---|---|---|
| 1 | $\mathcal{L}$ w.o. Kleene+ | S1/S2 | Precise | Linear | $RW \times c_r$ |
| 2 | $\mathcal{L}$ w.o. Kleene+ | S3 | Precise | Polynomial | $(\frac{(RW)^{l+1}-1}{RW-1}) \times c_r$ |
| 3 | $\mathcal{L}$ w. Kleene+ | S3 | Precise | Exponential | $(\frac{(RW)^{l-k+1}-1}{RW-1} \times 2^{kRW}) \times c_r$ |
| 4 | $\mathcal{L}$ w. Kleene+, uncorrelated | S1/S2/S3 | Imprecise | Exponential | $(\frac{(RW)^{l-k+1}-1}{RW-1} \times 2^{kRW}) \times (c_r + U^{l-k} \times c_m)$ |
| 5 | $\mathcal{L}$ w. Kleene+, correlated | S1/S2/S3 | Imprecise | Exponential | $(\frac{(RW)^{l-k+1}-1}{RW-1} \times 2^{kRW}) \times (c_r + U^{l-k+kRW} \times c_m)$ |

**Table 4.2.** Main results of runtime complexity analysis.

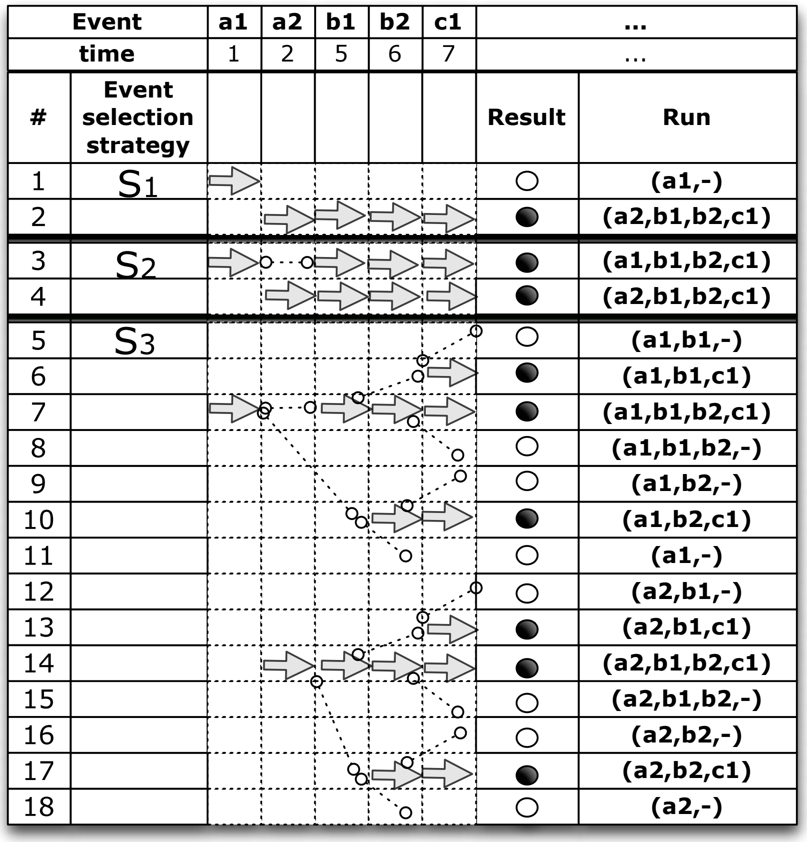
key language features. The runtime analysis will help us find intuitions for optimization later.

Preliminaries: The runtime cost is mainly reflected by the number of simultaneous runs of an NFA$^b$ 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 till 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 4.1.

Figure 4.1 illustrates the possible runs for different selection strategies. Figure 4.1(a) shows a simple pattern with Kleene+. In Figure 4.1(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 4.1 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 $(a1, b1, b2, c1)$ skips $a2$ during pattern matching because it is irrelevant after the run selects $a1$. 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.

```
PATTERN(a, b+, c)
WITHIN 10
```

**(a) A simple query with Kleene+**



| Event | | a1 | a2 | b1 | b2 | c1 | ... | |
|---|---|---|---|---|---|---|---|---|
| time | | 1 | 2 | 5 | 6 | 7 | ... | |
| # | Event selection strategy | | | | | | Result | Run |
| 1 | S1 | | | | | | ○ | (a1,-) |
| 2 | | | | | | | ● | (a2,b1,b2,c1) |
| 3 | S2 | | | | | | ● | (a1,b1,b2,c1) |
| 4 | | | | | | | ● | (a2,b1,b2,c1) |
| 5 | S3 | | | | | | ○ | (a1,b1,-) |
| 6 | | | | | | | ● | (a1,b1,c1) |
| 7 | | | | | | | ● | (a1,b1,b2,c1) |
| 8 | | | | | | | ○ | (a1,b1,b2,-) |
| 9 | | | | | | | ○ | (a1,b2,-) |
| 10 | | | | | | | ● | (a1,b2,c1) |
| 11 | | | | | | | ○ | (a1,-) |
| 12 | | | | | | | ○ | (a2,b1,-) |
| 13 | | | | | | | ● | (a2,b1,c1) |
| 14 | | | | | | | ● | (a2,b1,b2,c1) |
| 15 | | | | | | | ○ | (a2,b1,b2,-) |
| 16 | | | | | | | ○ | (a2,b2,-) |
| 17 | | | | | | | ● | (a2,b2,c1) |
| 18 | | | | | | | ○ | (a2,-) |

**(b) Possible runs for different selection strategies**

**Figure 4.1.** An example to illustrate different selection strategies.

Below we highlight our key results in five cases that cause significant changes of runtime complexity, while leaving out the full results including other cases due to space constraints. The relations of the five cases are summarized in Table 4.2.

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 $(\frac{(RW)^{l+1}-1}{RW-1})$, 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 $(\frac{(RW)^{l-k+1}-1}{RW-1})) \times 2^{kRW})$, exponential in $W$.

Imprecise Timestamps: Finally consider all patterns in $\mathcal{L}$ in the presence of imprecise timestamps. Recent work [55] proposed a solution for simple pattern queries like 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 [55] 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 [55]. More details of the algorithm are shown in Appendix 4.6. 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 [55], 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, requires computing the confidence of this sequence bing 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, $(\frac{(RW)^{l+1}-1}{RW-1})(c_r + U^l \times c_m)$, 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 2.1. 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, $(\frac{(RW)^{l-k+1}-1}{RW-1} \times 2^{kRW})$, times the cost per run, $(c_r + U^{l-k+kRW} \times c_m)$.

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.

## 4.2 Optimizations

Our key insight for optimization is derived from the observed difference between the low-level complexity classes in descriptive complexity analysis[56], 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 (§4.2.1 and §4.2.2).

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 (§4.2.3).

### 4.2.1 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. 4.2(a), and a small event stream in Fig. 4.2(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. 4.2(c). An initial set of operations according to the NFA$^b$ execution model are shown in Fig. 4.2(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:

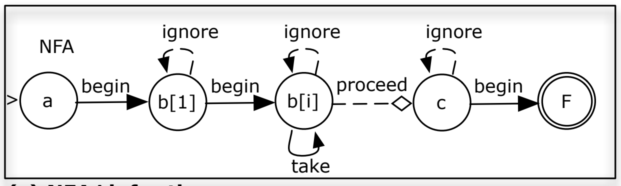
- Edge evaluation evaluates the condition on taking the transition marked by the edge, where the condition is compiled from the event type, time window constraint, and value predicates—this can be broadly considered a "predicate evaluation" step.
- Run initialization is used to start a new run.

```
PATTERN SEQ(A a, B+ b[], C c)
WHERE   [task_id] ^
        b[i].val >= b[i-1].val
WITHIN  100
```

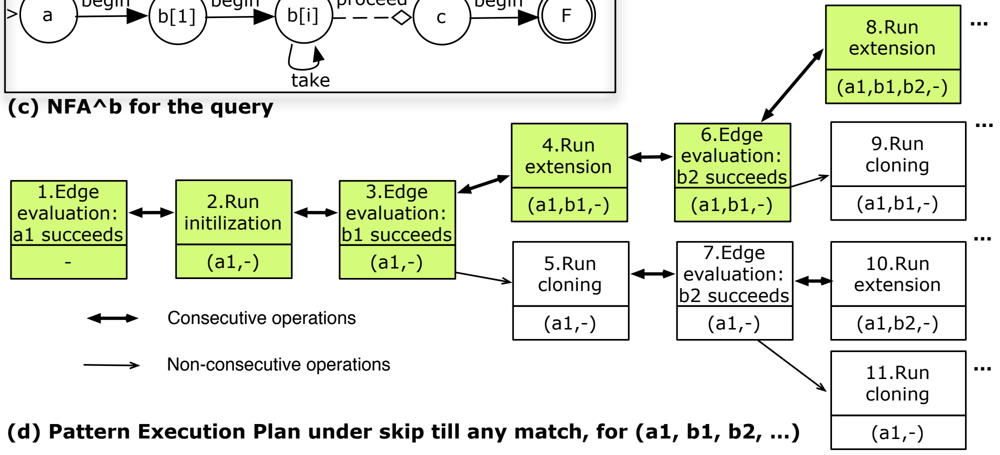
**(a) A simplified version of Q6**

| Event | a1 | b1 | b2 | b3 | c1 |
|---|---|---|---|---|---|
| time | 1 | 4 | 5 | 6 | 7 |
| task_id | 1 | - | - | - | 1 |
| val | - | 6 | 7 | 9 | - |

**(b) An example event sequence**



**(c) NFA^b for the query**



**(d) Pattern Execution Plan under skip till any match, for (a1, b1, b2, ...)**

| | NFA^b | Postponing |
|---|---|---|
| **a1** | (a1,-) | (a1,-) |
| **b1** | (a1,b1,-)<br>(a1,-) | (a1,<b1>,-) |
| **b2** | (a1,b1,b2,-)<br>(a1,b1,-)<br>(a1,b2,-)<br>(a1,-) | (a1,<b1,b2>,-) |
| **b3** | (a1,b1,b2,b3,-)<br>(a1,b1,b2,-)<br>(a1,b1,b3,-)<br>(a1,b1,-)<br>(a1,b2,b3,-)<br>(a1,b2,-)<br>(a1,b3,-)<br>(a1,-) | (a1,<b1,b2,b3>,-) |
| **c1** | **(a1,b1,b2,b3,c1)**<br>(a1,b2,b3,-)<br>**(a1,b1,b2,c1)**<br>(a1,b1,b2,-)<br>**(a1,b1,b3,c1)**<br>(a1,b1,b3,-)<br>**(a1,b1,c1)**<br>(a1,b1,-)<br>**(a1,b2,b3,c1)**<br>(a1,b1,b2,b3,-)<br>**(a1,b2,c1)**<br>(a1,b2,-)<br>**(a1,b3,c1)**<br>(a1,b3,-)<br>(a1,-) | (a1,<b1,b2,b3>,c1)<br><br>Then enumerate,<br>**(a1,b1,b2,b3,c1)**<br>**(a1,b1,b2,c1)**<br>**(a1,b1,b3,c1)**<br>**(a1,b1,c1)**<br>**(a1,b2,b3,c1)**<br>**(a1,b2,c1)**<br>**(a1,b3,c1)** |

**(e) Runs w. and w.o. postponing**

**Figure 4.2.** A running example for the postponing algorithm.

- Run extension adds a new event to an existing run.
- Run cloning duplicates an existing run to enable non-deterministic 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 'take' edge in the NFA$^b$ in Fig. 4.2(c), where Kleene+ is trying to select more 'b' events. Let $e$ denote the current event. The predicates in this edge evaluation are: $e.type = B \wedge e.time < W \wedge e.val \geq b[i-1].val$. 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. 4.2(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. 4.2(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. 4.2(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. 4.2(d)). We denote the main path as $\rho_1$, and each suffix path as $\rho_j = \rho_i + \Delta\rho$, with some $1 \leq i < j$. The observations above lead to two propositions key to our optimization.

**Proposition 4.2.1.** *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, $\mathcal{M} = (e_{i_1}, e_{i_2}, \ldots, e_{i_m})$, then $\mathcal{M}$ 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 $\mathcal{M}$. □

**Proposition 4.2.2.** *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 4.2.1, $\mathcal{M}$ 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 $S_3$. □

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 4.2.1: 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 4.2.2: We take the match $\mathcal{M}$ produced by the main run $\rho_1$ with value predicates postponed. Then we enumerate all subsets of $\mathcal{M}$ 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. 4.2(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 $\mathcal{M} = (a_1, < b_1, b_2, b_3 >, c_1)$, 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.

### 4.2.2 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 $\mathcal{M}$ 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 $\mathcal{M} =$ (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 $b[i].val > max(b[..i-1].val)$ 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 selected events. $c.val < max(b[..i].val)$ 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", 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 $b[i].val > avg(b[1..i-1].val)$ 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.

### 4.2.3 Optimization on Confidence Computation

As mentioned in §4.1, 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 [55] 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.

**Algorithm 7** Dynamic programming optimization

**Input: A run $r$, whose buffers has been all filled, events for its Kleene closure component are $e[0..n]$; the confidence threshold $c$**

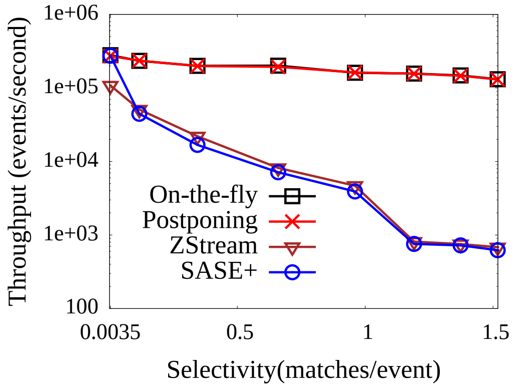
**Output: All enumerations that will make a match with confidence higher than $c$**

```
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 $S$
```

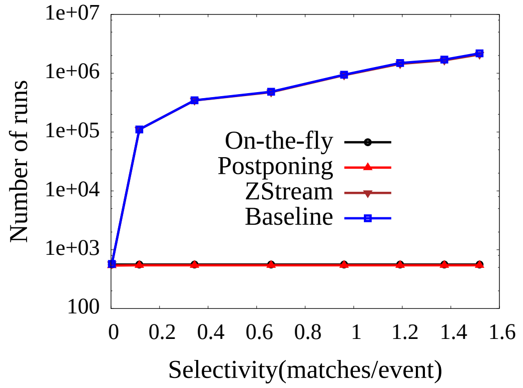
**Dynamic programming optimization**. Based on the above intuition, a dynamic programming method is designed to optimize the performance of confidence computation. The pseudocode is in Algorithm 7. The input is $r$ which already collects a set of events $e[0..n]$ for a Kleene+ component. We first initialize 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 1-3. 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, $S$ would be the whole qualified matches with confidence higher than $c$.

## 4.3 Evaluation

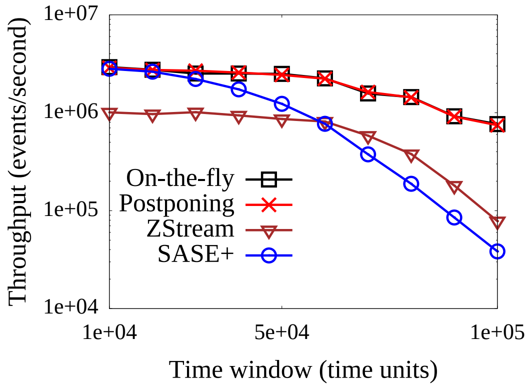
In this section, we evaluate our new system, called $SASE^{++}$, with the proposed optimizations, and compare it with several state-of-the-art pattern evaluation systems. Our



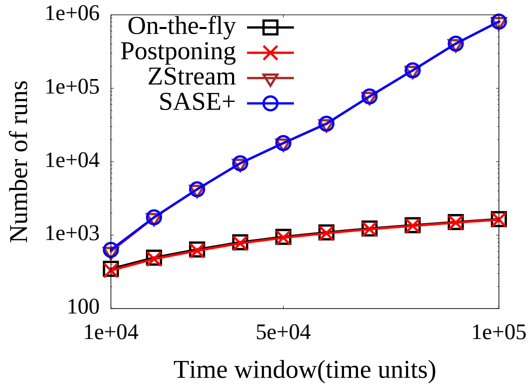
(a) Vary $\theta$ (T-value consistent predicates)



(b) Number of runs with varied $\theta$ (T-value consistent predicates)



(c) Vary $W$ (T-value consistent predicates)



(d) Number of runs with varied $W$ (T-value consistent predicates)

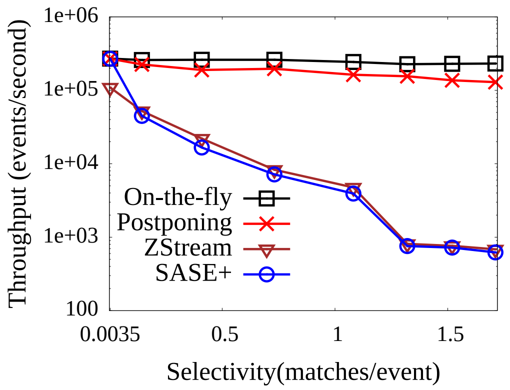
**Figure 4.3.** Microbenchmarks results for T-value consistent predicates

evaluation uses both microbenchmarks with controlled properties and a detailed case study in Hadoop cluster monitoring.

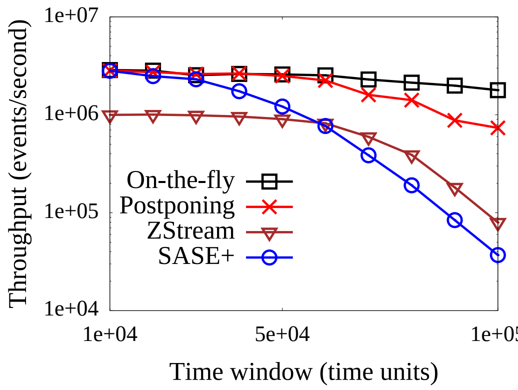
### 4.3.1 Microbenchmarks

Queries in the microbenchmarks use the template, SEQ(A $a$, B+ $b$[ ], C $c$), unless stated otherwise, and $S_3$ . We vary two parameters: The **selectivity** ($\theta$) defined as, $\frac{\#Matches}{\#Events}$, is controlled by changing the value predicates in the pattern. 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(§4.2.1) only; (2) **On-the-fly,** which applies early filters (§4.2.2) based on postponing; (3) **Collapsed,**



(a) Vary $\theta$ (F-value consistent predicates)



(b) Vary $W$ (F-value consistent predicates)

**Figure 4.4.** Microbenchmarks results for F-value consistent predicates

which returns results in collapsed format based on on-the-fly; (4) **DP $x$:** it applies dynamic programming (§4.2.3) 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** [34], 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+** [3, 52], which strictly follows the execution of the NFA$^b$ model, and (7) **XSeq** [35], which we describe in detail shortly. Table 4.3 lists all the algorithms and systems compared in our study.

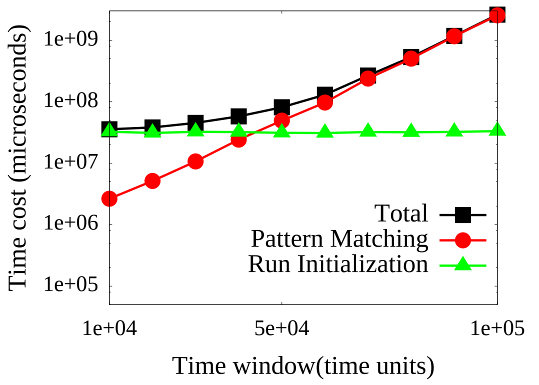
| # | System/Algorithm in Comparison | Shorthand |
|---|---|---|
| (1) | $SASE^{++}$ with optimization of postponed operations | Postponing |
| (2) | Postponing + predicate evaluation on the fly | On-the-fly |
| (3) | On-the-fly + results in a collapsed format | Collapsed |
| (4) | On-the-fly + dynamic programming for confidence computation (threshold = x%) | DP x |
| (5) | $SASE^{++}$ with the ZStream optimization | ZStream |
| (6) | The SASE+ system | SASE+ |
| (7) | The full XSeq system | XSeq |

**Table 4.3.** Algorithms and systems compared in our study.

All experiment results were obtained on a server with an Intel Xeon Quad-core 2.83GHz CPU and 8GB memory. System $SASE^{++}$ runs on Java HotSpot 64-Bit Server VM.

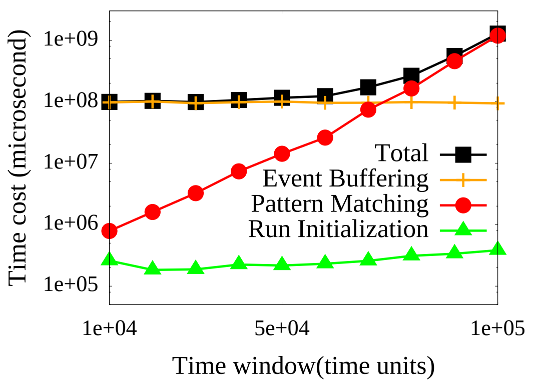
### 4.3.2 Evaluation with Precise Timestamps

We first evaluate the two optimizations, postponing (§4.2.1) and on-the-fly (§4.2.2), using streams with precise timestamps.

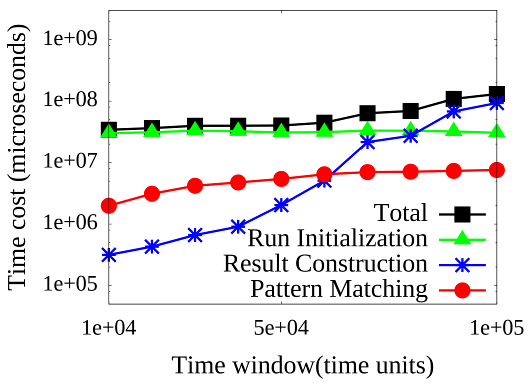


(a) SASE+ cost breakdown with varied $\theta$ (T-value consistent predicates)

8



(b) ZStream cost breakdown with varied $\theta$ (T-value consistent predicates)



(c) On-the-fly cost breakdown with varied $\theta$ (T-value consistent predicates)

**Figure 4.5.** Microbenchmarks results with precise timestamps

Throughput: Figure 4.3(a) and 4.3(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 4.3(b) and 4.3(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 4.4(a) and 4.4(b). 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 4.5(a). 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 4.5(b). The additional cost compared to SASE+ is the 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 4.5(c). 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 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.

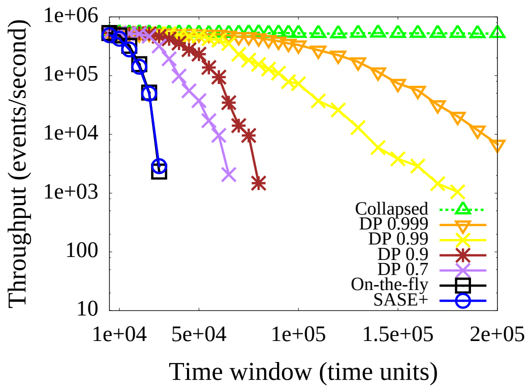
### 4.3.3 Evaluation with Imprecise Timestamps

In this set of experiments, we generate streams where each event has an uncertainty interval size of 10. Fig. 4.6(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. 4.6(d). Note that the DP method is based on the postponing algorithm; without the intermediate matches, such optimization on confidence computation is not feasible.

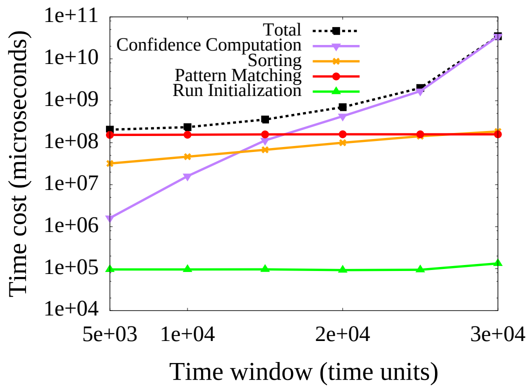
Figure 4.6(b) shows the cost breakdown of the SASE+ algorithm, while Figure 4.6(c) show the cost breakdown of the dynamic programming algorithm with 0.999 as the confidence threshold. We could see confidence computation cost is dominant in both figures. Please note that the value range of X-axis in Figure 4.6(c) is much wider than that in Figure 4.6(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.

### 4.3.4 Comparison with XSeq

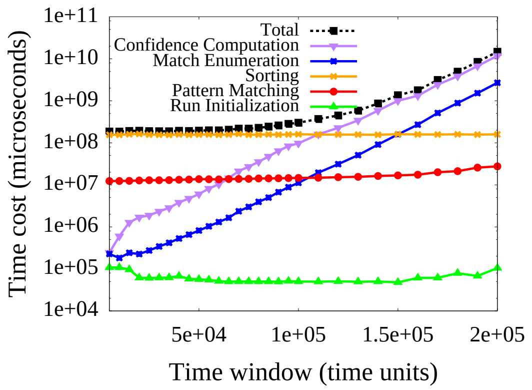
We further compare the performance of our system with XSeq, an engine for high-performance complex event processing over hierarchical data like XML, JSON etc, (which won the best paper award at SIGMOD 2012). For comparison, the same synthetic stream is used, and it is converted to the SAX format required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



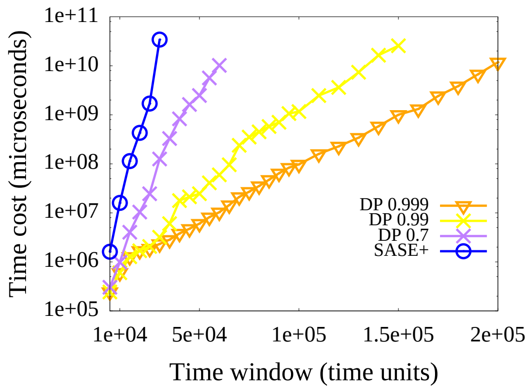
(a) Vary $W$, TF-value consistent predicates



(b) Cost breakdown for SASE+



(c) Cost breakdown for DP



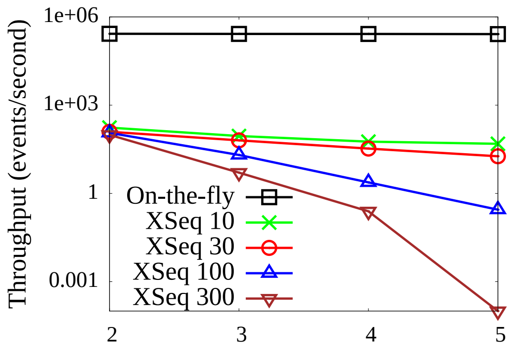
(d) Cost on confidence computation

**Figure 4.6.** Microbenchmarks results for imprecise timestamps

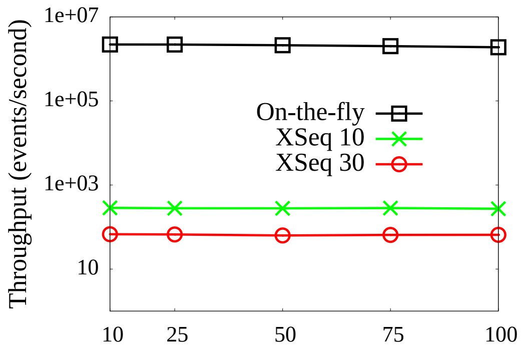
to the All_Match_Skip_One mode, which finds all possible matches for each starting point. The optimization method of XSeq is set to VP_OPS_ OPTIMIZATION, which gives the best performance.

We first vary the **query length** $l$ for SEQ$(A_1, \ldots, A_l)$. The result is in Fig. 4.7(a). 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. 4.7(b). 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



(a) XSeq for a pattern($A_1$,. . .,$A_l$), $W$=25

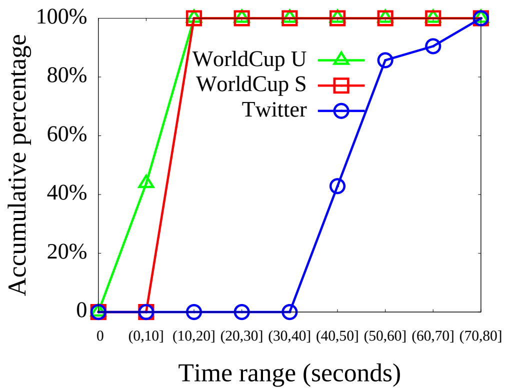


(b) XSeq for a Pattern(A,B+,C), varied $W$

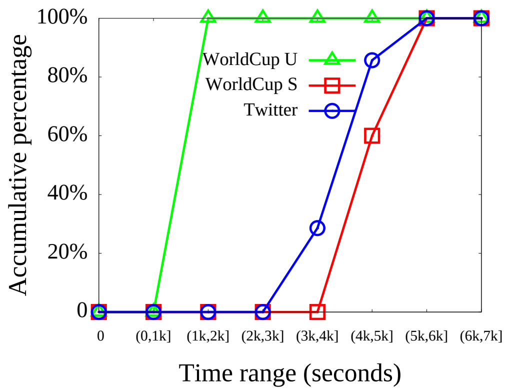
**Figure 4.7.** Microbenchmarks results for comparison with XSeq

the paper [35], 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.



(a) Mapper average running time



(b) Reducer average running time

**Figure 4.8.** Results for Q1 and Q2 in Hadoop use case study

### 4.3.5 Case study: Hadoop Cluster Monitoring

As stated in recent work [39], 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.

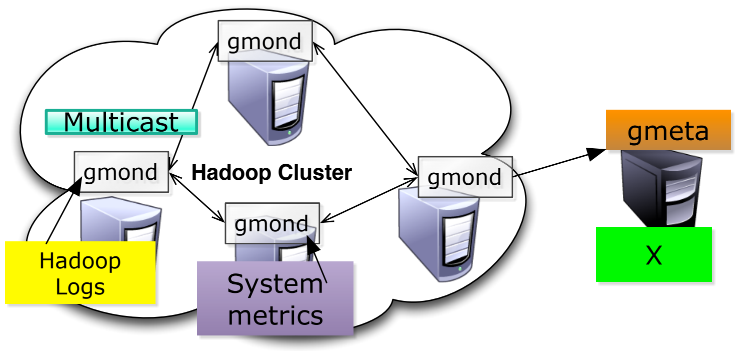


**Figure 4.9.** The architecture for real-time monitoring

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 [33], a popular distributed system monitor tool, is used as the core part of our real-time data collection system as shown in Fig. 4.9. 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.

| **Workload** | **Raw data (GB)** | **Map output (GB)** |
|---|---|---|
| Twitter | 53.5 | 565 |
| Worldcup U | 252.9 | 32 |
| Worldcup S | 252.9 | 263.5 |

**Table 4.4.** Hadoop workload statistics

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 2.1, we discuss other queries below, which are listed in Table 4.5.

Q1 computes the statistics of lifetime of mappers in Hadoop. Similarly, Q2 does it for reducers. Fig. 4.8(a) and Fig. 4.8(b) show the average lifetime of mappers and reducers for three different workloads in Table 4.4: 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. 4.8(a), the Twitter job has much longer running time than the other two workloads because the output size of its mappers is larger than the other two. In Fig. 4.8(b), the reducers for the WorldCup U

| Q | Pattern Query |
|---|---|
| Q1 | Pattern SEQ(JobStart $a$, Mapper+ $b$[ ], JobEnd $c$)<br>Where $a$.job_id = $b[i]$.job_id $\wedge$ $a$.job_id=$c$.job_id<br>Within 1 day<br>Return $avg$($b$[ ].period), $max$($b$[ ].period) |
| Q2 | Pattern SEQ(JobStart $a$, Reducer+ $b$[ ], JobEnd $c$)<br>Where [job_id]<br>Within 1 day<br>Return $avg$($b$[ ].period), $max$($b$[ ].period) |
| Q3 | Pattern SEQ(ReducerStart $a$, DataPull+ $b$[ ])<br>Where [task_id] $\wedge$ ($b$[$b$.LEN].period$>2\times avg$($b$[ ].period))<br>Within 10 minutes<br>Return $a$.(task_id, period) |
| Q4 | Pattern SEQ(JobStart $a$, DataIO+ $b$[ ])<br>Where [job_id]<br>Within 1 day<br>Return $b$[$b$.LEN].timestamp,$a$.job_id, $a$.timestamp, $sum$($b$[].size) |
| Q5 | Pattern SEQ(Balance $a$, ReducerStart $b$, Imbalance+ $c$[],<br>ReducerEnd $d$, Balance $e$)<br>Where [task_id]<br>Within 10 minutes<br>Return $a$.task_id |
| Q6 | Pattern SEQ(ReducerStart $a$, LoadStd+ $b$[ ], ReducerEnd $c$)<br>Where [task_id] $\wedge$ ($b[i]$.val $\geq$ $b[i$-1].val //option 1)<br>($b[i]$.val $\geq$ $max$($b$[1..$i$-1].val //option 2)<br>Within 10 minutes<br>Return $a$.task_id |

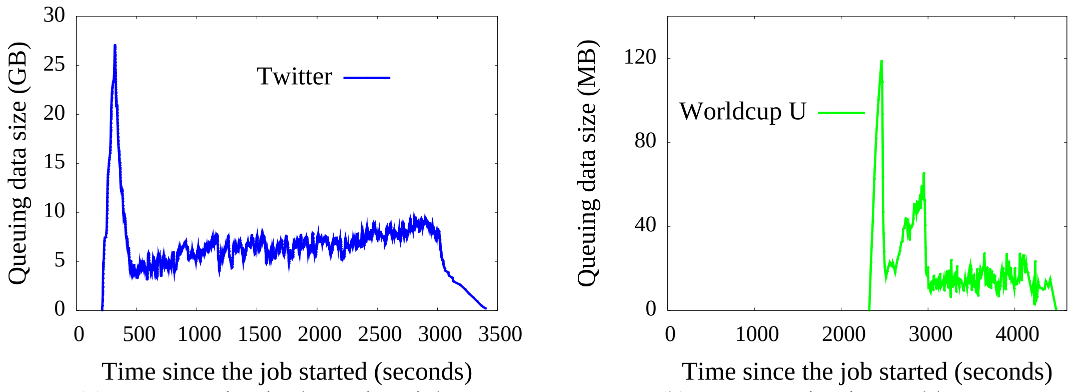
**Table 4.5.** Other pattern queries for Hadoop monitoring.

workload have longer running time because the job of sessionization is more complex than the other two jobs.

Q3 is used to find the data pull stragglers. A reducer is considered as a straggler when its runtime is two times the average of other reducers [39]. Given the task id returned by the query, user can then check logs and locate the specific information to know what was wrong with that task.

Q4 offers real-time monitoring for the queuing data size. 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. 4.10(a). 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. 4.10(b) 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.

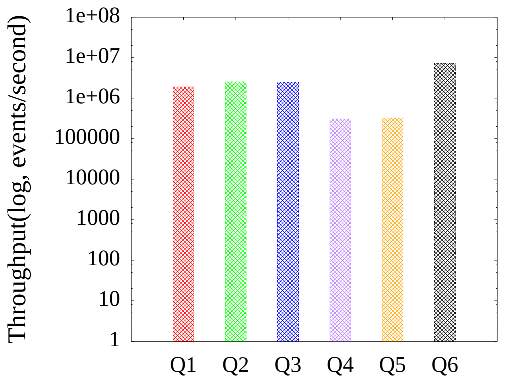
Q5 and Q6 are used to find tasks that cause cluster imbalance. As Q6 is described above, we simply note that they both use uncertainty intervals for timestamps due to granularity mismatch of Ganglia logs and Hadoop logs, and differ only in the ways of defining imbalanced load.



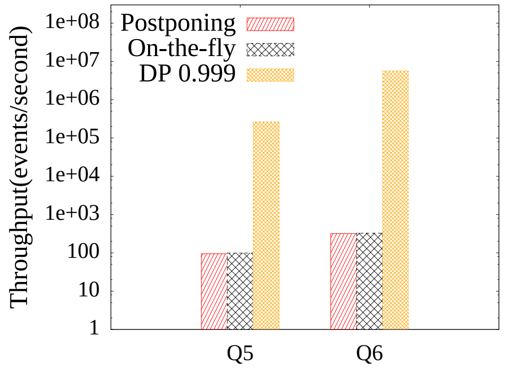
(a) Data queuing in the Twitter job (b) Data queuing in Worldcup U

**Figure 4.10.** Results for Q4 in Hadoop use case study

Performance: Fig. 4.11(a) shows throughput of all 6 queries, ranging from 300,000 to over 7 million events per second. The data rate in our experiment is 13.62 event/second/node. This means that a single server running system $SASE^{++}$ can monitor up to 22,000 nodes in real-time for these queries. For post analysis, it only takes 0.00454% of the actual running time of the monitoring process. Authors of [39] provide some public datasets, where the data rate is 0.758 event/second/node in the busiest month, and even lower in other months. Fig. 4.11(b) compares the optimization algorithms for Q5 and Q6. It shows the effectiveness of the optimizations.



(a) Throughput for all queries



(b) Optimizations

**Figure 4.11.** Performance Results for Hadoop use case study

## 4.4 Related work

Temporal models: The discussion for CEP over streams with imprecise timestamps is based on the uncertain temporal model in [55]. Other temporal models [5, 4, 6, 15] 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 [26, 54]; optimize for performance given out of order streams [24]; optimize the performance of nested pattern queries by pushing negation into inner subexpressions [30]; and rewrite queries in a more efficient form before translating them into automata [42]. In distributed systems, the work [4] applies plan-based techniques to minimize event transmission costs and efficiently perform CEP queries across distributed event sources.

## 4.5 Conclusions

This paper presented theoretical results on 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.

## 4.6 Algorithm for Imprecise timestamps

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[55]. 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 [55], always using $S_3$.

# CHAPTER 5

# EXPLAINING ANOMALIES IN CEP-BASED MONITORING AND PROACTIVE MONITORING

## 5.1 Introduction

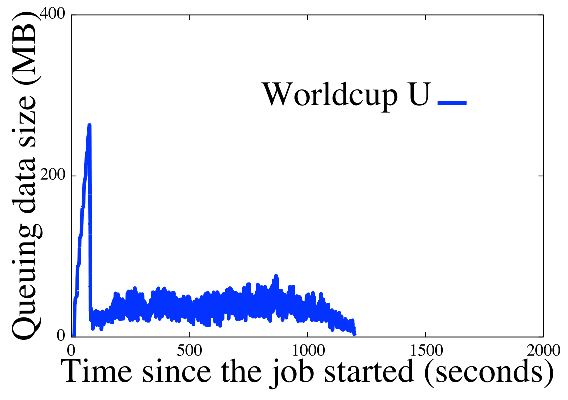
Complex Event Processing (CEP) extracts useful information from large-volume event streams in real-time. Users define interesting patterns in a CEP query language (e.g,. [3, 6]). With expressive query languages and high performance processing power, CEP technology is now at the core of real-time monitoring in a variety of areas, including the Internet of Things [32], financial market analysis [32], and cluster monitoring [56].

However, today's CEP technology supports only *passive monitoring* by requesting the monitoring application (or user) to explicitly define patterns of interest. There is a recent realization that many real-world applications demand a new service beyond passive monitoring, that is, the ability of the monitoring system to identify interesting patterns (including anomalous behaviors), produce a concrete explanation from the raw data, and based on the explanation enable a user action to prevent or remedy the effect of an anomaly. We broadly refer to this new service as *proactive monitoring*. We present two motivating applications as follows.

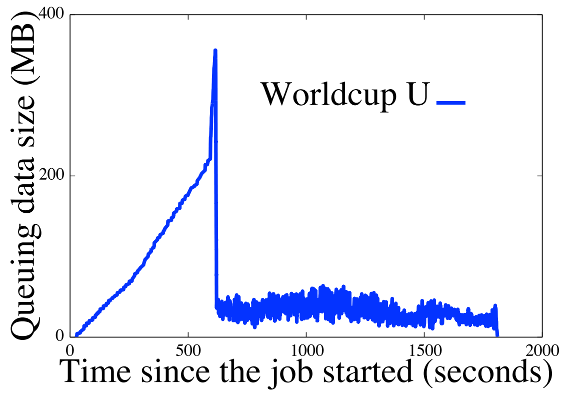
### 5.1.1 Motivating Applications

Production Cluster Monitoring: Cluster monitoring is crucial to many enterprise businesses. For a concrete example, consider a production Hadoop cluster that executes a mix of Extract-Transform-Load (ETL) workloads, SQL queries, and data stream tasks. The programming model of Hadoop is MapReduce, where a MapReduce job is composed of a `map` function that performs data transformation and filtering, and a `reduce` function that performs aggregation or more complex analytics for all the data sharing the same key. During job execution, the map tasks (called mappers) read raw data and generate intermediate results, and the reduce tasks (reducers) read the output of mappers and generate final output. Many of the Hadoop jobs have deadlines because any delay in these jobs will affect the entire daily operations of the enterprise business. As a result, monitoring of the progress of these Hadoop jobs has become a crucial component of the business operations.

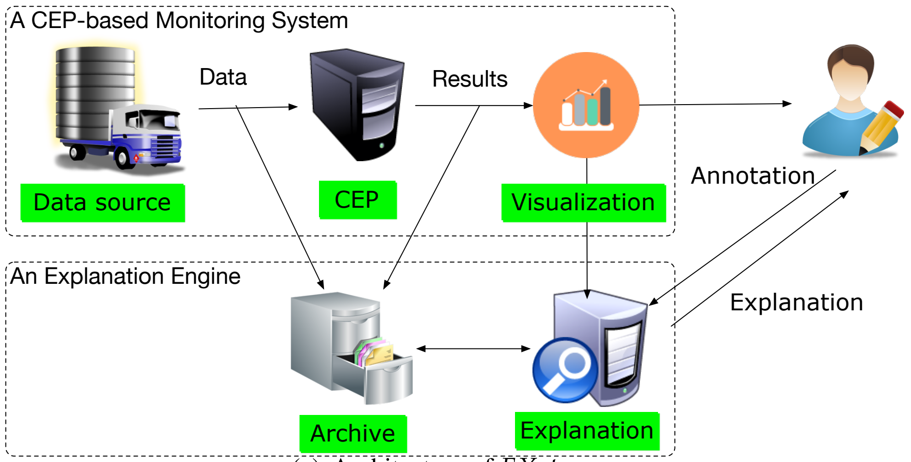
However, the Hadoop system does not provide sufficient monitoring functionality by itself. CEP technology has been shown to be efficient and effective for monitoring a variety of measures [56]. By utilizing the event logs generated by Hadoop and system metrics collected by Ganglia[21], CEP queries can be used to monitor Hadoop job progress; to find tasks that cause cluster imbalance; to find data pull stragglers; and to compute the statistics



(a) Data queuing size of a normal Hadoop job



(b) Data queuing size of an abnormal Hadoop job



(c) Architecture of *EXstream*

**Figure 5.1.** Hadoop cluster monitoring: examples and system architecture.

of lifetime of mappers and reducers. Consider a concrete example below, where the CEP query monitors the size of intermediate results that have been queued between mappers and reducers.

**Example 5.1.1** (Data Queuing Monitoring)**.** *Collect all the events capturing intermediate data generation/consumption for each Hadoop job. Return the accumulative intermediate data size calculated from those events (Q1).*

Figure 5.1(a) shows the data queuing size of a monitored Hadoop job. The X-axis stands for the time elapsed since the beginning of the job, while the Y-axis represents the size of queued data. In this case, the job progress turns out to be normal: the intermediate results output by the mappers start to queue at the beginning and reach a peak after a short period of time. This is because a number of mappers have completed in this period while the reducers have not been scheduled to consume the map output. Afterwards, the queued data size decreases and then stabilizes for a long period of time, meaning that the mappers and reducers are producing and consuming data at constant rates, until the queued data reduces to zero at the end of the job.

Suppose that a Hadoop user sees a different progress plot, as shown Figure 5.1(b), for the same job on another day: there is a long initial period where the data queuing size increases gradually but continually, and this phase causes the job completion time to be delayed by more than 500 seconds. When the user sees the job with an odd shape in Figure 5.1(b), he may start considering the following questions:

- What is happening with the submitted job?
- Should I wait for the job to complete or re-submit it?
- Is the phenomenon caused by the bugs in the code or some system anomalies?
- What should I do to bring the job progress back to normal?

Today's CEP technology, unfortunately, does not provide any additional information that helps answer the above questions. The best practice is manual exploration by the Hadoop user: he can dig into the complex Hadoop logs and manually correlate the Hadoop events with the system metrics such as CPU and memory usage returned by a cluster monitoring tool like Ganglia [21]. If he is lucky to get help from the cluster administrator, he may collect additional information such as the number of jobs executed concurrently with his job and the resources consumed by those jobs.

For our example query, the odd shape in Figure 5.1(b) is due to high memory usage of other programs in the Hadoop cluster. However, this fact is not obvious from the visualization of the user's monitoring query, Q1. It requires analyzing additional data beyond what is used to compute Q1 (which used data relevant only to the user's Hadoop job, but not all the jobs in the system). Furthermore, the discovery of the fact requires new tools that can automatically generate explanations for the anomalies in monitoring results such that these explanations can be understood by the human and lead to corrective / preventive actions in the future.

Supply Chain Management: The second use case is derived from an aerospace company with a global supply chain. By talking with the experts in supply chain management, we identified an initial set of issues in the company's complex production process which may lead to imperfect or faulty products. For instance, in the manufacturing process of a certain product the environmental features must to be strictly controlled because they affect the quality of production. For example, the temperature and humidity need to be controlled in a certain range, and they are recorded by the sensors deployed in the environment. However, if some sensors stop working, the environmental features may not be controlled properly and hence the products manufactured during that period can have quality issues. When such anomalies arise, it is a huge amount of work to investigate the claims from customers given the complexity of manufacturing process and to analyze a large set of historical data to find explanations that are meaningful and actionable.

### 5.1.2 Problem Statement and Contributions

The overall goal of *EXstream* is to provide good explanations for anomalous behaviors that users annotate on CEP monitoring results. We assume that an enterprise information system has CEP monitoring functionality: a CEP monitoring system offers a dashboard to illustrate high-level metrics computed by a CEP query, such as job progress, network traffic, and data queuing. When a user observes an abnormal value in the monitoring results, he annotates the value in the dashboard and requests *EXstream* to search for an explanation from the archived raw data streams. *EXstream* generates an optimal explanation(formalized in Section 5.2.2) by quickly replaying a fraction of the archived data streams. Then the explanation can be encoded into the system for proactive monitoring for similar anomalies in the future.

| **Event type** | **Meaning** | **Schema** |
|---|---|---|
| JobStart | Recording a Hadoop job starts | (timestamp, eventType, eventId, jobId, clusterNodeNumber) |
| JobEnd | Recording a Hadoop job finishes | (timestamp, eventType, eventId, jobId, clusterNodeNumber) |
| DataIO | Recording the activities of generation (positive values) / consumption (negative values) of intermediate data | (timestamp, eventType, eventId, jobId, taskId, attemptId, clusterNodeNumber, dataSize) |
| CPUUsage | Recording the CPU usage for a node in the cluster | (timestamp, eventType, eventId, clusterNodeNumber, CPUUsage) |
| MemUsage | Recording the memory usage for a node in the cluster | (timestamp, eventType, eventId, clusterNodeNumber, memUsage) |

**Figure 5.2.** Example event types in Hadoop cluster monitoring. Event types can be specific to the Hadoop job (e.g., JobStart, DataIO, JobEnd), or they may report system metrics (e.g., CPUUsage, FreeMemory).

Challenges: The challenges in the design of XStream arise from the requirements for such explanations. Informed by the two real-world applications mentioned above, we consider three requirements in this work: (a) Conciseness: The system should favor smaller explanations, which are easier for humans to understand. (b) Consistency: The system should produce explanations that are consistent with human interpretation. In practice, this means that explanations should match the true reasons for an anomaly (*ground truth*). (c) Prediction power: We prefer explanations that have predictive value for future anomalies.

It is difficult for existing techniques to meet all three requirements. In particular, prediction techniques such as logistic regression and decision trees [2] suffer severely in conciseness or consistency as shown in our evaluation results. This is because these techniques were designed for prediction, but not for explanations with conciseness and consistency requirements. Recent database research [53, 40] seeks to explain outliners in SQL query answers. This line of work assumes that explanations can be found by searching through various subsets of the tuples that were used to compute the query answers. This assumption does not suit real-world stream systems for two reasons: As shown for our example, Q1, the explanation of memory usage contention among different jobs cannot be generated from only those events that produced the monitoring results of Q1. Furthermore,

| | |
|---|---|
| $Q$ | Pattern SEQ($Component_1$, $Component_2$ , . . . )<br>Where [$partitionAttribute$] $\wedge$ $Pred_1$ $\wedge$ $Pred_2$ $\wedge$ . . .<br>Return ($timestamp$, $partitionAttribute$, $derivedA_1$, $derivedA_2$, . . .)[] |

| | |
|---|---|
| $Q_1$ | Pattern SEQ(JobStart $a$, DataIO+ $b$[], JobEnd $c$)<br>Where [jobId]<br>Return ($b[i]$.timestamp, $a$.jobId, $sum$($b$[1$\cdots$ i].dataSize))[] |

**Figure 5.3.** Syntax of a query in SASE (on the left), and an example query for monitoring data activity (on the right).

the stream execution model does not allow us to repeat query execution over different subsets of events or perform any precomputation in a given database [40].

Contributions: In this work, we take an important step towards discovering high-quality explanations for anomalies observed in monitoring results. Toward this goal, we make the following contributions:

1) Formalizing explanations (Section 5.2): We provide a formal definition of optimally explaining anomalies in CEP monitoring as a problem that maximizes the information reward provided by the explanation.

2) Sufficient feature space (Section 5.3): A key insight in our work is that discovering explanations first requires a sufficient feature space that includes all necessary features for explaining observed anomalies. *EXstream* includes a new module that automatically transforms raw data streams into a richer feature space, **F**, to enable explanations.

3) Entropy-based, single-feature reward (Section 5.4): As a basis for building the information reward of an explanation, we model the reward that each feature, $f \in \mathbf{F}$, may contribute using a new entropy-based distance function.

4) Optimal explanations via submodular optimization (Section 5.5): We next model the problem of finding an optimal explanation from the feature space, **F**, as a submodular maximization problem. Since submodular optimization is NP-hard, we design a heuristic algorithm that ranks and filters features efficiently and effectively.

5) Evaluation (Section 5.7): We have implemented *EXstream* on top of the SASE stream engine [3, 56]. Experiments using two real-world use cases show promising results: (1) Our entropy distance function outperforms state-of-the-art distance functions on time series by reducing the features considered by 94.6%. (2) *EXstream* significantly outperforms logistic regression [2], decision tree [2], majority voting [28] and data fusion [37] in consistency and conciseness of explanations while achieving comparable, high predication accuracy. Specifically, it outperforms others by improving consistency from 10.7% to 87.5% on average, and reduces 90.5% of features on average to ensure conciseness. (3) Our implementation is also efficient: with 2000 concurrent monitoring queries, the triggered explanation analysis returns explanations within half a minute and affects the performance only slightly, delaying events processing by 0.4 second on average.

## 5.2 Explaining CEP anomalies

The goal of *EXstream*is to provide good explanations for anomalous behaviors that users annotate on CEP-based monitoring results. We first describe the system setup, and give examples of monitoring queries and anomalous observations that a user may annotate. We then discuss the requirements for providing explanations for such anomalies, and examine whether some existing approaches can derive explanations that fit these requirements. Finally, we define the problem of optimally explaining anomalies in our setting.

### 5.2.1 CEP Monitoring System and Queries

In this section, we describe the system setup for our problem setting. The overall architecture of *EXstream*is shown in Figure 5.1(c). Within the top dashed rectangle in Figure 5.1(c) is a CEP-based monitoring system. We consider a data source $S$, generating events of $n$ types, $\mathbf{E} = \{E_1, E_2, \ldots, E_n\}$. Events of these types are received by the CEP-based monitoring system continuously. Each event type follows a schema, comprised of a set of attributes; all event schemas share a common timestamp attribute. The timestamp attribute records the occurrence time of each event. Figure 5.2 shows some example event types in the Hadoop cluster monitoring use case [56].

We consider a CEP engine that monitors these events using user-defined queries. For the purposes of this paper, monitoring queries are defined in the SASE query language [3], but this is not a restriction of our framework, and our results extend to other CEP query languages. Figure 5.3 shows the general syntax of CEP queries in SASE, and an example query, $Q_1$, from the Hadoop cluster monitoring use case. $Q_1$ collects all data-relevant events during the lifetime of a Hadoop job. We now explain the main components of a SASE query.

Sequence: A query $Q$ may specify a sequence using the `SEQ` operator, which requires components in the sequence to occur in the specified order. One component is either a single event or the Kleene closure of events. For example, $Q_1$ specifies three components: the first component is a single event of the type $JobStart$; the second component is a Kleene closure of a set of events of the type $DataIO$; and the third component is a single event of type $JobEnd$.

Predicates: $Q$ can also specify a set of predicates in its `Where` clause. One special predicate among these is the bracketed $partitionAttribute$. The brackets apply an equivalence test on the attribute inside, which requires all selected events to have the same value for this attribute. The $partitionAttribute$ tells the CEP engine which attribute to partition by. In $Q_1$, $jobId$ is the partition attribute.

Return matches: $Q$ specifies the matches to return in the `Return` clause. Matches comprise a series of events with raw or derived attributes; we assume $timestamp$ and the $partitionAttribute$ are included in the returned events. We denote with $m$ a match on one partition and with $M_Q$ the set of all matches. $Q_1$ returns a series of events based on selected $DataIO$ events, and the returned attributes include $timestamp$, $jobId$, and a derived attribute— the total size for all selected $DataIO$ events. In order to visualize results in real time, matches will be sent to the visualization module as events are collected.

Visualizations and feedback: Our system visualizes matches from monitoring queries on a dashboard that users can interact with. The visualizations typically display the (relative)

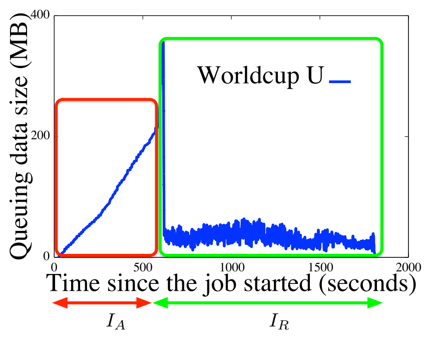


**Figure 5.4.** Abnormal ($I_A$) and reference ($I_R$) intervals.

occurrence time on the X-axis. The Y-axis represents one of the derived attributes in returned events. Users can specify simple filters to focus on particular partitions. All returned events of $M_Q$ are stored in a relational table $T_{M_Q}$, and the data to be visualized for a particular partition is specified as $\pi_{t,attr\_i}(\sigma_{partitionAttribute=v}(M))$. Figure 5.1(a) shows the visualization of a partition, which corresponds to a Hadoop job for this query. In this visualization, the X-axis displays the time elapsed since the job started, and the Y-axis shows the derived sum over the "DataSize" attribute.

Users can interact with the visualizations by annotating anomalies. For example, the visualization of Figure 5.1(b) demonstrates an unexpected behavior, with the queueing data size growing slowly. A user can drag and draw rectangles on the visualization, to annotate the abnormal component, as well as reference intervals that demonstrate normal behavior. We show an example of these annotations in Figure 5.4. A user may also annotate an entire period as abnormal, and choose a reference interval in a different partition. The annotations will be sent to the explanation engine of $EXstream$, which is shown in the bottom dashed rectangle of Figure 5.1(c). The explanation engine will be introduced in detail in following sections. We use $I_A$ to denote the annotated abnormal interval in a partition $P_A$: $I_A = (Q, [lower, upper], P_A)$. We use $I_R$ to denote the reference interval, which can be explicitly annotated by the user, or inferred by $EXstream$as the non-annotated parts of the partition. We write $I_R = (Q, [lower, upper], P_R)$, where $P_R$ and $P_A$ might be the same or different partitions.

### 5.2.2 Explaining Anomalies

Monitoring visualizations allow users to observe the evolution of various performance metrics in the system. While the visualizations help indicate that something may be unusual (when an anomaly is observed), they do not offer clues that point to the reasons for the unexpected behavior. In our example from Figure 5.4, there are two underlying reasons for the abnormal behavior: (1) the free memory is lower than normal, and (2) the free swap space is lower than normal. However, these reasons are not obvious from the visualization; rather, a Hadoop expert had to manually check a large volume of logs to derive this explanation. Our goal is to automate this process, by designing a system that seamlessly integrates with CEP monitoring visualizations, and which can produce explanations for surprising observations.

We define three desirable criteria for producing explanations in $EXstream$:

| No. | Feature | Weight |
|---|---|---|
| 1 | DataIOFrequency | -0.01376 |
| 2 | CPUIdleMean | 0.0089 |
| 3 | PullFinishFrequency | -0.00708 |
| 4 | ProcTotalMean | 0.00085 |
| ... | ... | ... |
| 23 | SwapFreeMean | -4.79E-07 |
| 24 | MemFreeMean | -3.28E-07 |
| ... | ... | ... |
| 30 | BoottimeMean | 2.61E-10 |

**Figure 5.5.** Model generated by logistic regression for the annotated anomaly of Figure 5.4.

1. **Conciseness**: The system should favor smaller, and thus simpler explanations. Conciseness follows the Occam's razor principle,and produces explanations that are easier for humans to understand.

2. **Consistency**: The system should produce explanations that are consistent with human interpretation. In practice, this means that explanations should match the true reasons for an anomaly (*ground truth*).

3. **Prediction power**: We prefer explanations that have predictive value for future anomalies. Such explanations can be used to perform *proactive monitoring*.

**Explanations through predictive models** The first step of our study explored the viability of existing prediction techniques for the task of producing explanations for CEP monitoring anomalies. Prediction techniques typically learn a model from training data; by using the anomaly and reference annotations as the training data, the produced model can be perceived as an explanation. For now, we will assume that a sufficient set of features is provided for training (we discuss how to construct the feature space in Section 5.3), and evaluate the explanations produced by two standard prediction techniques for the example of Figure 5.4.

**Logistic regression** [2] produces models as weights over a set of features. The algorithm processes events from the two annotated intervals as training data, and the trained prediction model — a classifier between abnormal and reference classes — can be considered an explanation to the anomaly. The resulting logistic regression model for this example is shown in Figure 5.5. While the model has good predictive power, it is too complex, and cannot facilitate human understanding of the reported anomaly. The model assigns non-zero weights to 30 out of 345 input features, and while the two ground truth explanations identified by the human expert are among these features (23 and 24), their weights in this model are low. This model is too noisy to be of use, and it is not helpful as an explanation.

**Decision tree** [2] builds a tree for prediction. Each non-leaf node of the tree is a predicate while leaf nodes are prediction decisions. Figure 5.6 shows the resulting tree for our example. The decision tree algorithm selects three features for the non-leaf nodes, and only one of them is part of the ground truth determined by our expert. The other two features happen to

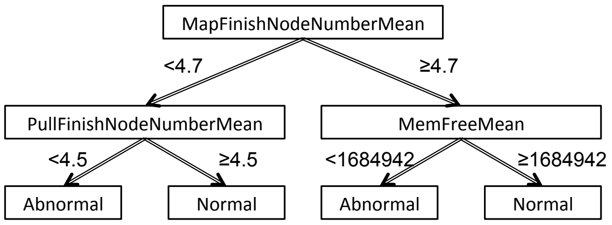


**Figure 5.6.** Model Generated by Decision Tree

| Algorithm | Conciseness | Consistency | Prediction quality |
|---|---|---|---|
| Logistic regression | Bad | Bad | Good |
| Decision tree | Ok | Bad | Good |
| Goal | Good | Good | Good |

**Figure 5.7.** Performance of prediction methods on our three criteria for explanations.

be *coincidentally* correlated with the two intervals, as revealed in our profiling. This model is more concise than the result of logistic regression, but it is not consistent with the ground truth.

The above analyses showed that prediction techniques are not suitable for producing explanations in our setting. While the produced models have good predictive power (as this is what the techniques are designed for), they make poor explanations, as they suffer in consistency and conciseness. Our goal is to design a method for deriving explanations that satisfies all three criteria (Figure 5.7).

### 5.2.3 Formalizing Explanations

Explanations need to be understandable to human users, and thus need to have a simple format. $EXstream$builds explanations as a conjunction of predicates. In their general format, explanations are defined as follows.

**Definition 1** (Explanation)**.** *An explanation is a boolean expression in Conjunctive Normal Form (CNF). It contains a conjunction of clauses, each clause is a disjunction of predicates, and each predicate is of the form* $\{v\ o\ c\}$*, where* $v$ *is a variable value,* $c$ *is a constant, and* $o$ *is one of five operators:* $o \in \{>, \geq, =, \leq, <\}$.

**Example 5.2.1.** *The formal form of the true explanations for the anomaly annotated in Figure 5.4 is (*$MemFreeMean$ *¡ 1978482* $\wedge$ $SwapFreeMean$ *¡ 361462), which is a conjunction of two predicates. It means that the average available memory is less than 1.9GB and free swap space is less than 360MB. The two predicates indicate that the memory usage is high in the system (due to resource contention), thus the job runs slower than normal.*

Arriving at the explanation of Example 5.2.1 requires two non-trivial components. First, we need to identify important features for the annotated intervals (e.g., $MemFreeMean$, $SwapFreeMean$); these features will be the basis of forming meaningful predicates for the explanations. Second, we have to derive the best explanation given a metric of optimality.

For example, the explanation ($MemFreeMean$ ¡ 1978482) is worse than ($MemFreeMean$ ¡ 1978482 $\wedge$ $SwapFreeMean$ ¡ 361462), because, while it is smaller, it does not cover all issues that contribute to the anomaly, and is thus less consistent with the ground truth.

Ultimately, explanations need to balance two somewhat conflicting goals: simplicity, which pushes explanations to smaller sizes, and informativeness, which pushes explanations to larger sizes to increase the information content. We model these goals through a reward function that models the information that an explanation carries, and we define the problem of deriving optimal explanations as the problem of maximizing this reward function.

**Definition 2** (Optimal Explanation)**.** *Given an archive of data streams $D$ for CEP, a user-annotated abnormal interval $I_A$ and a user-annotated reference interval $I_R$, an optimal explanation $e$ is one that maximizes a non-monotone, submodular information reward $R$ over the annotated intervals:* $\text{argmax}_e R_{I_A,I_R}(e)$

The reward function in Definition 2 is governed by an important property: rewards are not additive, but *submodular*. This means that the sum of the reward of two explanations is greater than or equal to the reward of their union: $R_{I_A,I_R}(e_1) + R_{I_A,I_R}(e_2) \geq R_{I_A,I_R}(e_1 \cup e_2)$. The intuition for the submodularity property is based on the observation that adding predicates to a conjunctive explanation offers diminishing returns: the more features an explanation already has, the lower the reward of adding a new predicate tends to be. Moreover, $R$ is non-monotone. This means that adding predicates to an explanation *could decrease the reward*. This is due to the conciseness requirement that penalizes big explanations. The optimal explanation problem (Definition 2) is therefore a submodular maximization problem, which is known to be NP-hard [20].

### 5.2.4 Existing Approximation Methods

Submodular optimization problems are commonly addressed with greedy approximation techniques. We next investigate the viability of these methods for our problem setting.

For this analysis, we assume a reward function for explanations based on mutual information. Mutual information is a measure of mutual dependence between features. This is important in our problem setting, as features are often correlated. For example, $PullStartFrequency$ and $PullFinishFrequency$ are highly correlated, because they always appear together for every pull operation. For this precise reason, Definition 2 demands a submodular reward function. Mutual information satisfies the submodularity property. Greedy algorithms are often used in mutual information maximization problems. The way they would work in this setting is the following: given an explanation $e$, which is initially empty, at each greedy step, we select the feature $f$ that maximizes the mutual information of $e \cup f$.

Figure 5.8 shows the performance of the greedy algorithm for maximization of mutual information, with a strawman alternative. The random algorithm selects a random feature at each step. The greedy strategy clearly outperforms the alternative by reaching higher mutual information gains with fewer features, but it still selects a large number of features (around 20-30 features before it levels off). This means that this method produces explanations that are too large, and unlikely to be useful for human understanding.

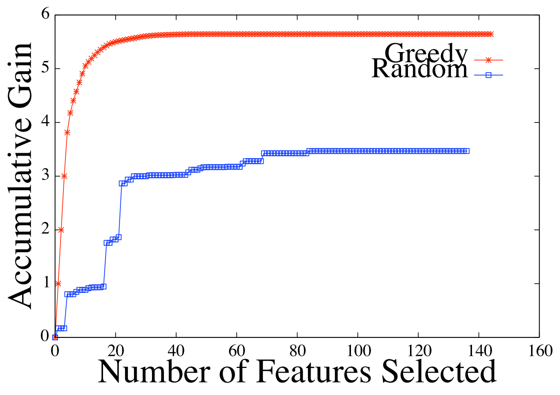


**Figure 5.8.** Accumulative mutual information gain under greedy and random strategies.

### 5.2.5 Overview of the *EXstream* Approach

Since standard approaches for solving the optimal explanation problem are insufficient for our problem setting, we develop a new heuristic method based on good intuitions to address the problem. We next provide a high-level overview of our approach in building *EXstream*.

**1. Sufficient feature space** (Section 5.3): A key insight in our work is that discovering optimal explanations first requires a sufficient feature space that includes all necessary features for explaining observed anomalies. Our work differs fundamentally from existing work on discovering explanations from databases [53, 40]: First, *EXstream* operates on raw data streams, as opposed to the data carefully curated and stored in a relational database. Second, *EXstream* does not assume that the raw data streams carry all necessary features for explaining anomalous behaviors. In our above example, the feature, *SwapFreeMean*, captures average free swap space and it does not exist in Hadoop event logs or Ganglia output. Our system includes a module that automatically transforms raw data streams into a richer feature space, **F**, to enable the discovery of optimal explanations.

**2. Entropy-based, single-feature reward** (Section 5.4): As a basis for building the information reward defined in Definition 2, we consider the reward that each feature, $f \in \mathbf{F}$, may contribute. To capture the reward in such a base case, we propose a new, entropy-based distance function that is defined on a single feature across the abnormal interval, $I_A$, and the reference interval, $I_R$. The larger the distance, the more differentiating power over the two intervals that the feature contributes, and hence more reward produced.

**3. Optimal explanations via submodular optimization** (Section 5.5): The next task is to find an optimal explanation from the feature space, **F**, that maximizes the information reward provided by the explanation. The reward function in Definition 2 is non-monotone and submodular, resulting in a submodular maximization problem. Since submodular optimization is NP-hard, our goal is to design a heuristic to solve this problem. Our heuristic algorithm first uses the entropy-based, single-feature reward to rank features, subsequently identifies a cut-off to reject features with low reward, and finally uses correlation-based filtering to

| timestamp | node | usagePercent |
|---|---|---|
| 4 | 2 | 35 |
| 5 | 5 | 49 |
| 6 | 8 | 99 |
| 7 | 1 | 86 |
| 8 | 2 | 61 |
| 9 | 6 | 43 |

**Figure 5.9.** Sample events in the type of $CPUUsage$.

eliminate features with information overlap (emulating the submodularity property). Our evaluation shows that our heuristic method is extremely effective in practice.

## 5.3 Discovering useful features

Explanations comprise of predicates on measurable properties of the CEP system. We call such properties *features*. Some features for our running example are $DiskFreeMean$, $MemFreeMean$, $DataIOFrequency$, etc. In most existing work on explanations, features are typically determined by the query or the schema of the data (e.g., the query predicates in Scorpion [53]). In CEP monitoring, using as features the query predicates or data attributes is not sufficient, because many factors that impact the observed performance are due to other events and changes in the system. This poses an additional challenge in our problem setting, as the set of relevant features is non-trivial. In this section, we discuss how $EXstream$derives the space of features as a first step to producing explanations.

In an explanation problem, we are given an anomaly interval $I_A$ and a reference interval $I_R$; the relevant features for this explanation problem are built from events that occurred during these two intervals. To support the functionality of providing explanations, the CEP system has to maintain an archive of the streaming data. The system has the ability to purge archived data after the relevant monitoring queries terminate, but maintaining the data for longer can be useful, as the reference interval can be specified on any past data.

Formally, the events arriving in a CEP system in input streams and the generated matches compose the input to the feature space construction problem. We assume that the CEP system maintains a table for each event type, such as the one depicted in Figure 5.9. That is, for each event type $E_i$, logically there is a relational table $R(E_i)$ to store all events of this type in temporal order. There is also a table $R(M)$ to archive all match events, denoted as type $M$. Let $D$ denote the database for $EXstream$, which is composed of those tables. So, $D$ is defined as $D = \{R(E_i)|1 \leq i \leq n\} \cup R(M)$.

Each attribute in event type $E_i$, except the timestamp, forms a time series in a given interval (which can be an anomaly interval $I_A$ or a reference interval $I_R$). Such time series as features are called *raw features*.

**Example 5.3.1.** *The table of Figure 5.9 records events of type $CPUUsage$ in a given time interval [4, 9], and forms two raw features, from two time series. The first one is*

*CPUUsage.Node, and its values are ((4, 2), (5,5), (6,8), (7,1), (8,2), (9,6)); the other is CPUUsage.UsagePercent with values ((4,35), (5,49), (6,99), (7,86), (8,61), (9,43)).*

We found that the raw feature space is not good for deriving explanations due to noise. Instead, we need higher-level features, which we construct by applying aggregation functions to features at different granularities. We apply sliding windows over the time series features and over each window, aggregate functions including *count* and *avg* to generate new time serious features. The *EXstream* system has an open architecture that allows any window size and any new aggregate functions to be used in the feature generation process. Features produced this way are "smoothed" time series; they demonstrate more general trends than raw features, and outliers are smoothed. Example high-level features that we produce by applying aggregations over windows on the raw features are *DataIOFrequency* and *MemFreeMean*.

In our experiment, we use 30 seconds for the sliding window size because it smoothes the raw data, most of which are collected every 15 seconds, and also it is not too large that details will be lost. For aggregate functions, we use Average and Frequency. In practice, users should try to use as many aggregate functions as possible because the more features, the higher possibility that ground truth is included.

## 5.4 Single-feature reward

In this section, we present the core of our technique: an entropy-based distance function that models the reward of a single feature. We first discuss the intuition and requirements for this function, we then discuss existing, state-of-the-art distance functions and explain why they are not effective in this setting, and, finally, we present our new entropy-based distance metric.

### 5.4.1 Motivation and Insights

In seeking explanations for CEP monitoring anomalies, users contrast an anomaly interval with a reference interval. An intuitive way to think about the different behaviors in the two intervals is to consider the differences in the events that occur within each interval. We can measure this difference per feature: how different is each feature between the reference and the anomaly. Each feature is a vector of values, a time series, and our goal is to measure the distance between the time series of a feature during the abnormal interval and the time series of the same feature during the normal interval.

To explain one of the desirable properties of the distance function, we visualize a feature as follows: We order the values of a feature in increasing order and assign a color to each value; red for values that appear in the abnormal interval only, yellow for values that appear in the normal interval only, and blue for values that appear in both normal and abnormal intervals. Figure 5.10 shows this visualization for 4 different features. In this figure, we note that the first 2 features show a clear separation of values between the normal and abnormal periods. The third feature has less clear separation, but still shows the trend that lower values are more likely to be abnormal. Finally, the fourth feature is mixed for a significant portion of values.

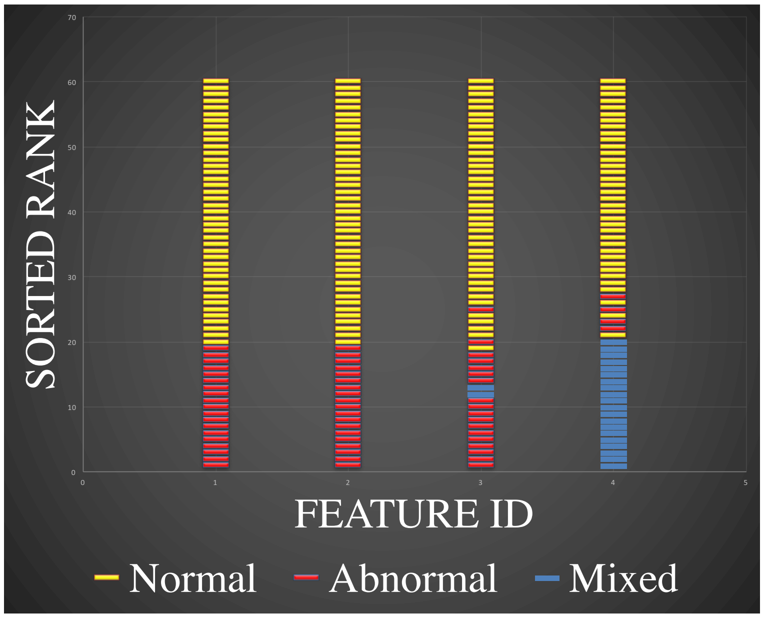


**Figure 5.10.** Visualization of the separating power of four features: (1) free memory size, (2) idle CPU percentage, (3) CPU percentage used by IO, and (4) system load. This visualization is not part of $EXstream$, but we show it here for exposition purposes.

Intuitively, the first two features in Figure 5.10 are better explanations for the anomaly, and thus have higher reward. The first feature means when the anomalies occur, the free memory size is relatively low, while during the reference interval the free memory size is relatively high. The second feature means that during the abnormal interval, idle CPU percentage is low while it is high during the reference interval. The unclear separation of the other two features, in particular the blue segments, indicate randomness between the two intervals, making them less suitable to explain the annotated anomalies.

This example provides insights on the properties that we need from the distance function: it should favor clear separation of normal and abnormal values, and it should penalize features with mixed segments (values that appear in both normal and abnormal periods). Therefore, the reward of a feature is high if the feature has good separating power, and it is lower with more segmentation in its values.

### 5.4.2 Existing State of the Art

Distance functions measuring similarities of time series have been well studied [50], and there is over a dozen distance functions in the literature. However, these metrics were designed with different goals in mind, and they do not fit our explanation problem well. We discuss this issue for the two major categories of distance functions [50].

Lock-step measure: In the comparison of two time series, lock-step measures compare the $i$th point in one time series to exactly the $i$th point in another. Such measures include the Manhattan distance ($L_1$), Euclidean distance ($L_2$) [18], other $L_p$-norms distances and approximation based $DISSIM$ distance. Those distance functions treat each pair of points independently, but in our case, we need to compare the time series holistically. For example, assume four simple time series: $TS_1 = (1,1,1)$, $TS_2 = (0,0,0)$, $TS_3 = (1,0,1)$ and $TS_4 = (0,1,0)$. Based on our separating power criterion, $D(TS_1, TS_2)$ should be larger than $D(TS_3, TS_4)$ because there is a clear separation between the values of $TS_1$ and $TS_2$, while the values of $TS_3$ and $TS_4$ are conflicting. However, applying any of the lock-step measures produces $D(TS_1, TS_2) = D(TS_3, TS_4)$.

Elastic measure: Elastic measures allow comparison of one-to-many points to find the minimum difference between two time series. These measures try to compare time series on overall patterns. For example, Dynamic Time Warping (DTW) tries to stretch or compress one time series to better match another time series; while Longest Common SubSequence(LCSS) is based on the longest common subsequence model. Although these measures also take value difference into account, the additional emphasis on pattern matching makes them ill-suited for our problem.

Both lock-step and elastic measures fall in the category of sequence-based metrics. This means that they consider the order of values. Lock-step functions perform strict step-by-step, or event-by-event comparisons; such rigid measures cannot find similarities in the flexible event series of our problem setting. Elastic measures allow more flexibility, but the emphasis on matching the microstructure of sequences introduces too much randomness in the metric.

In our case, temporal ordering is not important, because we assume the sample points in time series are independent. This makes set-based functions a better fit (as opposed to sequence-based). Set-based functions measure the macro trend while smoothing low-level details.

### 5.4.3 Entropy-Based Single-Feature Reward

Since existing distance functions are not suitable to model single-feature rewards, we design a new distance function that emphasizes the separation of feature values between normal and abnormal intervals (Section 5.4.1). Our distance function is inspired by an entropy-based discretization technique [19], which cuts continuous values into value intervals by minimizing the class information entropy. The segmentation visualized in Figure 5.10, shows an intuitive connection with entropy: The more mixed the color segments are, the higher the entropy (i.e., more bits are needed to describe the distribution). We continue with some background definitions, and then define our entropy-based distance function, which we will use to model single-feature rewards.

**Definition 3** (Class Entropy)**.** *Class entropy is the information needed to describe the class distributions between two time series. Given a pair of time series, $TS_A$ and $TS_R$, belonging to the abnormal and reference classes, respectively. Let $|TS_A|$ and $|TS_R|$ denote the number of points in the two time series, let $p_A = \frac{|TS_A|}{|TS_A|+|TS_R|}$, and let $p_R = \frac{|TS_R|}{|TS_A|+|TS_R|}$. Then, the entropy of the class distribution is:*

$$H_{Class}(f) = p_A * log(\frac{1}{p_A}) + p_R * log(\frac{1}{p_R}) \tag{5.1}$$

**Definition 4** (Segmentation Entropy)**.** *Segmentation entropy is the information needed to describe how merged points are segmented by class labels. If there are $n$ segmentations,* *and $p_i$ represents the ratio of data points included in the ith segmentation, the segmentation entropy is:*

$$H_{Segmentation} = \sum_{i=1}^{n} p_i * log(\frac{1}{p_i}) \tag{5.2}$$

Complicated segmentations in a feature result in more entropy. When there is a clear separation of the two classes, as in the first two features of Figure 5.10, the segmentation entropy is the same as the class entropy. Otherwise, the segmentation entropy is more than the class entropy.

Penalizing for mixed segments: Segmentation entropy captures the segmentation of the normal and abnormal classes, but does not penalize mixed segments with values that appear in both classes (blue segments in the visualization). Take an extreme case, where all values appear in both classes (single mixed segment). This is the scenario with the worst separation power, but its segmentation entropy is 0, because it is treated as a single segment. This indicates that we need special treatment for mixed (blue) segments.

We assume the worst case distribution of normal and abnormal data points within the segment. This is the uniform distribution, which leads to most segmentation and highest entropy. For example, if a mixed segment $c$ consists of 5 data points, 3 contributed from the normal class (N) and 2 contributed from the abnormal class (A), distributing them uniformly leads to 5 segments: (N,A,N,A,N). We denote this worst-case ordering of segment $c$ as $c^*$. We assign a penalty term for each segment $c$, which is equal to the segmentation entropy of its worst-case ordering, $c^*$: $H_{Segmentation}(c^*)$. We thus define the regularized segmentation entropy:

$$H^+_{Segmentation} = H_{Segmentation} + \sum_{j=1}^{m} H_{Segmentation}(c^*_j) \tag{5.3}$$

The first term in this formula is the segmentation entropy of the feature, and the second term sums the regularization penalties of all mixed segments ($m$).

Accounting for feature size: Features may be of different sizes, as different event types may occur more frequently than others. The segmentation entropy is only comparable between two features $f_1$, $f_2$, if $|f_1.TS_A| = |f_2.TS_A|$ and $|f_1.TS_R| = |f_2.TS_R|$. However this does not hold for most features. To make these metrics comparable, we normalize segmentation entropy using class entropy and get the following definition for our entropy-based feature distance:

$$D(f) = \frac{H_{Class}(f)}{H^+_{Segmentation}(f)} \tag{5.4}$$

We use this distance function as a measure of single-feature reward. Features with perfect separation, such as the first two features of Figure 5.10, have reward equal to 1. Features with more complex segmentation have lower rewards. For the 4 features displayed in Figure 5.10, the rewards are 1, 1, 0.31, and 0.18, respectively.

## 5.5 Constructing explanations

The entropy-based single-feature reward identifies the features that best distinguish the normal and abnormal periods. However, ranking the features based on this distance metric is not sufficient to generate explanations. We need to address three additional challenges. First, it is not clear how to select a set of features from the ranked list. There is no specific constant $k$ for selecting a set of top-$k$ features, and moreover, such a set would likely not be meaningful as a top-$k$ set is likely to contain highly-correlated features with redundant information. Second, there are cases where large distances are coincidental, and not associated with anomalies. Third, the rewards are computed for each feature individually, and due to submodularity, they are not additive. Determining how to combine features into an explanation requires eliminating redundancies due to feature correlations.

We proceed to describe the $EXstream$approach to constructing explanations by addressing these challenges in three steps. Each step filters the feature set to eliminate features based on intuitive criteria, until we are left with a high-quality explanation.

### 5.5.1 Step 1: reward leap filtering

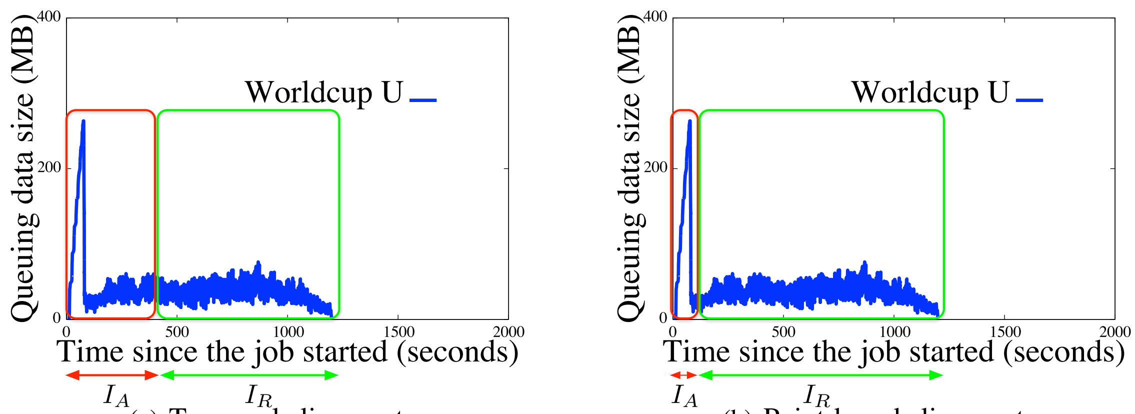
The single-feature distance function produces a ranking of all features based on their individual rewards. Sharp changes in the reward between successive features in the ranking indicate a semantic change: Features that rank below a sharp drop in the reward are unlikely to contribute to an explanation. Therefore, features whose distance is low, relatively to other features, can be safely discarded.

### 5.5.2 Step 2: false positive filtering

It is possible for features to have high rewards due to reasons unrelated to the investigated anomaly. For example, a feature that measures system uptime can have strong separating power between the annotated anomaly and reference regions (e.g., the anomaly is before the reference), but this is simply due to the nature of the particular feature, and it is not related to the anomaly. We call these features false positives. Our method for identifying and purging such false positives leverages other partitions (e.g., other Hadoop jobs in our running example). The intuition is that if a feature is a false positive, the feature will demonstrate similar behavior in other partitions without an indication of anomaly.

Identifying related partitions: We search the archived streams to identify similar partitions. Intuitively, such partitions should be results generated by the same query, monitoring the same Hadoop program, on the same dataset. $EXstream$maintains a record of partitions in a partition table to facilitate fast retrieval. The partition table contains dimension attributes that record categorical information (e.g., $CEP - QueryID$, $HadoopJobName$, $Dataset$) about the partition, and measure attributes that record partition statistics (e.g., monitoring duration, number of points). The system identifies related partitions, as those that match the dimension attributes.

Partition alignment: Once it discovers related partitions, $EXstream$needs to map the annotated regions to each related partition. This alignment can be temporal-based or point-based. In temporal-based alignment, an annotation is mapped to a partition based on its temporal length. For example, in Figure 5.4, the abnormal period occupies 31% of temporal length;



(a) Temporal alignment (b) Point-based alignment

**Figure 5.11.** Two ways of alignment

this annotation will align with the the first 31% of the temporal length in a related partition (Figure 5.11(a)). In point-based alignment an annotation is mapped to a partition based on the ratio of data points that it occupies in the monitoring graph. For example, the annotated high-memory usage partition of Figure 5.4 includes 113,070 points, with 2116 points falling in the abnormal annotation; this annotation will align with the first equal fraction of points in a related partition (Figure 5.11(b)). *EXstream*selects the alignment for which the two partitions have the smallest relative difference. For example, if a related partition has 10% more points, but is 50% longer in time compared to the annotated partition, point-based alignment is preferred.

Interval labeling: Alignment maps the annotations to all related partitions. Now, these new annotations need to be labeled as normal or abnormal. *EXstream*assigns labels through hierarchical clustering: a period that is placed in the same cluster as the annotated anomaly is labeled as abnormal. The clustering uses two distance functions: entropy-based, and normalized difference of frequencies. Periods whose cluster is far from the anomaly cluster are labeled as normal (reference). Finally, periods that cannot be assigned with certainty are discarded and not used later for validation.

In Figure 5.11(b), both intervals are assigned a “Reference” label. The left one is “Reference” because its frequency is significantly different from the annotated one (3.7 vs. 50.1); while the right one is “Reference” because both its frequency and value difference are quite small, meaning it is similar to the annotated “Reference” interval.

Feature validation: The process of partition discovering and automatic labeling generates a lot more labeled data that helps *EXstream*filter out false positives, and improve the current set of features. Features that have high entropy reward on the annotated partition will be reevaluated on the large dataset. If the high reward is validated in the larger dataset as well, the feature is maintained; otherwise, it is discarded. In our running example, after the validation step, only 6 out of 670 features remain. Figure 5.12 shows the reward for each of these 6 features for the annotated partition and the augmented partition set.

| Feature | Reward (annotated) | Reward (all) |
|---|---|---|
| Free memory size | 1 | 0.77 |
| Hadoop DataIO size | 1 | 0.64 |
| Num. of processes | 1 | 0.64 |
| Free swap size | 1 | 1 |
| Cached memory size | 0.81 | 0.77 |
| Buffer memory size | 0.65 | 0.72 |

**Figure 5.12.** The six validated features after the removal of false positives.

### 5.5.3 Step 3: filtering by correlation clustering

After the validation step, we are usually left with a small set of features, which have high individual rewards, and the high rewards are likely related to the investigated anomaly. However, it is still possible that several of these features have information overlap. For example, two identical features, are good individually, but putting them together in an explanation does not increase the information content. We identify and remove correlated features using clustering.

We use pairwise correlation to identify similar features. We represent a feature as a node; two nodes are connected, if the pairwise correlation of the two features exceeds a threshold. In our experiment, we manually choose this threshold. Users can adjust the threshold to control the conciseness of results: lower thresholds correlate more features, and the results contain fewer features, while higher thresholds lead to more features. We treat each connected component in this graph as a cluster, and select only one representative feature from each cluster. In our running example, the final six features are clustered into two clusters, one cluster with a single node, and another cluster with five nodes. Based on this result, the final explanation has two features.

### 5.5.4 Building final explanations

Once we make the final selection of features, the construction of an explanation is straightforward. For each selected feature, we can build a partial explanation in the format defined in Section 5.2.3. The feature name becomes the variable name. The value boundaries for the abnormal intervals become the constants. If a feature offers perfect separation during segmentation (Section 5.4), there is one boundary and only one predicate is built: e.g., the abnormal value range of feature $f_1$ is $(-\infty, 10]$, then the predicate is $f_1 \leq 10$. If a feature has more than one abnormal intervals, then multiple predicates are built to compose the explanation: e.g., the abnormal value ranges of feature $f_2$ are $(-\infty, 20]$,$[30, 50]$, and then the explanations are $f_2 \leq 20 \vee (f_2 \geq 30 \wedge f_2 \leq 50)$. Then we simply connect the partial explanations constructed from different features using conjunction and write the final formula into the conjunctive normal form.

## 5.6 System design

This section shows the system design of *EXstream*. We first present our design goals, then the overall system architecture, and finally the details of the two modules developed for finding explanations.

### 5.6.1 Design goals

Our design goals include both the functionality and performance of the system. Regarding the functionality, the explanation module should be highly integrated with existing CEP-based monitoring system, because it is triggered only when users observe anomalies from results of the monitoring system. For performance, it should meet two requirements: (1)after users trigger the explanation request, it should return answers as soon as possible, and the design goal is to be within 1 minute, which is a reasonable delay for such kind of tasks; (2)the explanation functionality should not hurt the monitoring performance seriously because the monitoring queries are always running and monitoring is the main purpose of the system, while explanation is an additional service which is supposed to run infrequently.

### 5.6.2 Architecture

With these design goals mentioned above, we design a CEP-based monitoring system with explanation functionality, and the new system is named *EXstream*. The architecture of *EXstream* is shown in Figure 5.13.

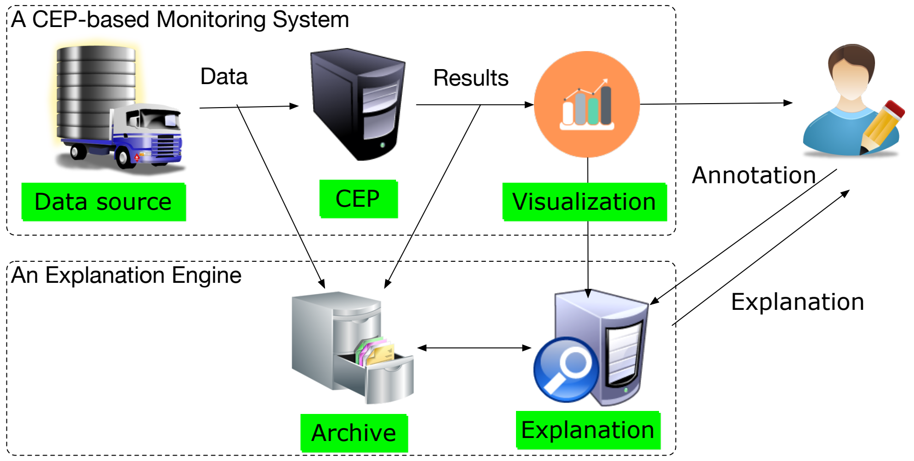
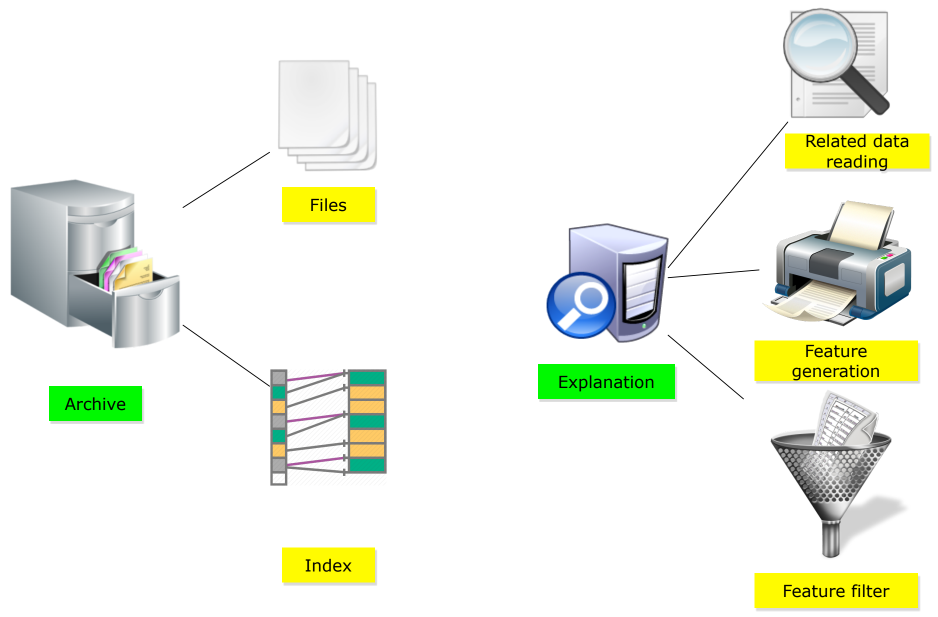


**Figure 5.13.** Architecture of *EXstream*

Within the top dashed rectangle is a CEP-based monitoring system without explanation functionality. The monitoring system is composed of three components. The data source represents the input data stream, which works as a gateway to collect all types of events from outside the monitoring system. For example, in our Hadoop cluster monitoring use case, the data source module collects system metrics and Hadoop logs from all nodes across the whole cluster. The CEP module in the architecture represents a CEP engine, which is the workhorse of the monitoring system. Users submit queries to the CEP engine and the engine will match incoming events against all queries. The visualization module visualizes the results from the user-specified queries.

We add two new modules for generating explanations, which are illustrated in the bottom dashed rectangle and discussed in detail below.

The archive module stores all events from the input stream by timestamp, as shown in Figure 5.14(a). Events are written into disk by event types. Events of the same type will be written sequentially to disk. The sequential writing lowers the cost on I/O in both writing and reading. In order to avoid reading unnecessary events when an explanation is triggered, events of the same type are chopped into smaller chunk files on disk. An index of the time range for each chunk is built. When events of a certain period are requested, the archive module first looks up the index, and then reads the needed events. The choice of the chunk size explores the trade off between the I/O cost and index storage cost. A larger chunk size potentially increases the cost on reading unnecessary events, but it reduces the size of index: the extreme case is all events of the same type are written into one big file, and every time the explanation is triggered the module has to read from the beginning of the file, while the index has only one entry for each event type. A smaller chunk size is better at avoiding unnecessary events, but the index needs more space: the extreme case is each file only stores events for one timestamp, and every event read is needed but the index will hold a large number of entries for each event type.



(a) Archive Module (b) Explanation Module

**Figure 5.14.** Main modules of *EXstream*

The explanation module accepts a request from users and analyzes related events to generate explanations, and its components are shown in Figure 5.14(b). Given user annotated abnormal intervals, the explanation engine requests events in the specific time period according to the index built by the archive module. Then all relevant events are converted to features in different granularities. After computing the entropy distance of every feature, and filtering noisy features, the module will return selected features as an explanation to users.

The architecture is demonstrated to be effective: it returns explanations quickly and only slightly affects the performance of the monitoring engine. Secion **??** shows the detailed results on performance.

## 5.7 Evaluation

We have implemented *EXstream* on top of the SASE stream engine [3, 56]. The implementation details are in Appendix 5.6. In this section, we evaluate *EXstream* on the conciseness, consistency, and prediction power of its returned explanations, and compare its performance with a range of alternative techniques in the literature. We further evaluate the efficiency of *EXstream* when the explanation module is run concurrently with monitoring queries in an event stream system.

### 5.7.1 Experimental Setup

In our first use case, we monitored a Hadoop cluster of 30 nodes which was used intensively for experiments at the University of Massachusetts Amherst. To evaluate *EXstream* for explaining anomalous observations, we used three Hadoop jobs: (A) Twitter Trigram: count trigrams in a twitter stream; (B) WC-Frequent users: find frequent users in a click stream; (C) WC-session: sessionization over a click stream.

The running example throughout this paper, which starts to show in Figure 5.1(a) and 5.1(b), is a real use case. A Hadoop expert found out the root causes by manually checking a large volume of logs. The expert also confirmed that the results generated by *EXstream* match the ground truth perfectly.

To enable the ground truth for evaluation further, we manually created four types of anomalies by running additional programs to interfere with resource consumption: (1) High memory usage: the additional programs use up memory. (2) High CPU: the additional programs keep CPU busy. (3) Busy disk: the programs keep writing to disk. (4) Busy network: the programs keep transmitting data between nodes. By combining the anomaly types and Hadoop jobs, we create 8 workloads listed in Figure 5.15. The ground truth features are verified by a Hadoop expert.

Our second use case is supply chain management of an aerospace company. Due to confidentiality issues we were unable to get real data. Instead, we consulted an expert and built a simulator to generate manufacturing data and anomalies such as faulty sensors and subpar material.

All of our experiments were run on a server with two Intel Xeon 2.67GHz, 6-core CPUs and 16GB memory. *EXstream* is implemented in Java and runs on Java HotSpot 64-bit server VM 1.7 with the maximum heap size set to 8GB.

| **No.** | Anomaly | **Hadoop workload** |
|---|---|---|
| 1 | High memory | WC-frequent users |
| 2 | High memory | WC-sessions |
| 3 | Busy Disk | WC-frequent users |
| 4 | High High CPU | WC-frequent users |
| 5 | High High CPU | WC-sessions |
| 6 | Busy High CPU | Twitter trigram |
| 7 | High Busy Network | WC-sessions |
| 8 | High Busy Network | Twitter trigram |

**Figure 5.15.** Workloads for evaluating the explanations returned by *EXstream*.

### 5.7.2 Effectiveness of Explanations by *EXstream*

We compare *EXstream* with a range of alternative techniques. We use **decision trees** to build explanations based on the latest version of weka, and **logistic regression** based on a popular R package. We consider two additional techniques, **majority voting** [28] and **data fusion** [37]. Both techniques make full use of every feature, and make prediction based on all features. Majority voting treats features equally and uses the label which counts the most as the prediction result. The fusion method fuses the prediction result from each feature based on their precision, recall and correlations. We compare these techniques on three measures: (1) consistency: selected features as compared against ground truth; (2) conciseness: the number of selected features; (3) prediction accuracy when the explanation is used as a prediction model on new test data.

Consistency: First we compare the selected features of each algorithm with the ground truth features. The results are shown in Figure 5.16. X-axis represents different workloads (1 - 8), while Y-axis is the F-measure, namely, the harmonic mean of precision and recall regarding the inclusion of ground truth features in the returned explanations. *EXstream* represents our results before applying clustering on selected features, while *EXstream*-cluster represents results clustered by correlations (Section 5.5). We can see that *EXstream*-cluster works better than *EXstream*without clustering for most of workloads, and *EXstream*-cluster provides much better quality than the alternative techniques. Majority voting and fusion do not select features, and hence their F-measures are low. Logistic regression and decision tree generate models with selected features, with sightly increased F-measures but still significantly below those of *EXstream*-cluster.

Conciseness: Figure 5.17 shows the sizes of explanations from each solution. Here the Y-axis (in logarithmic scale) is the number of features selected by each solution, where the total number of available features is 345. “Ground truth” represents the number of features in ground truth, while “Ground truth cluster” represents the number of clusters after we apply clustering on the contained features. Again, majority voting and fusion do not select features, so the size is the same as the size of feature space. The models of logistic regression includes 20 - 30 features, which is roughly 10 times of the ground truth. Decision trees are more concise with less than 10 features selected. Overall, *EXstream* outperforms other algorithms, and is quite close to the number of features in ground truth cluster.

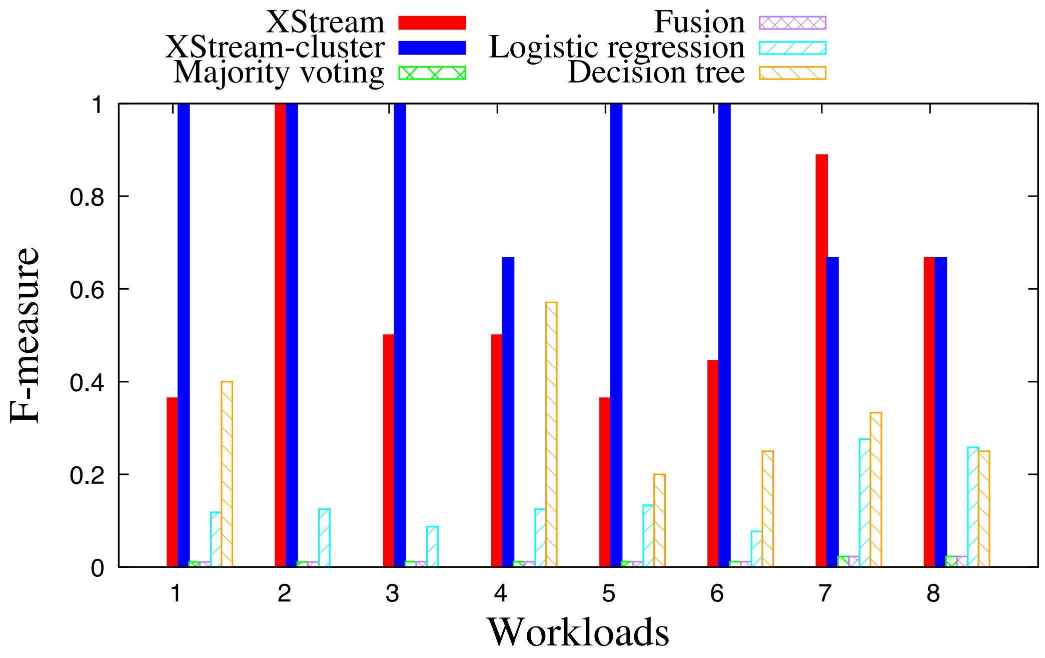


**Figure 5.16.** Consistency comparison

Predication accuracy: In Figure 5.18 we compare the prediction accuracy of each method. The Y-axis represents F-measure for prediction over new test data. The F-measures of $EXstream$, logistic regression and decision tree are quite stable, most of time above 0.95. Data fusion and majority voting fluctuate more. Overall, our method can provide consistent high-quality prediction power.

### 5.7.3 Evaluation for Supply chain management use case

#### 5.7.3.1 Detailed design of simulation

In order to study the problems in supply chain management, it is better to get some real traces from manufacturers. However, due to the confidentiality issues, we were unable to get real data. Thus a simulator is designed to generate manufacturing data to study this use case.

The two use cases shown above involves two categories of events: monitoring and materials. In practice, there must be other types of events, which are ignored here because we focus on studying the two types of use cases. So in simulation only those two categories of events will be generated.

In the monitoring category, we assume there are a number of different sensors reporting different measurements for the same place. **Rate:** each monitoring event series is reporting a specific measurement at a fixed rate, like one report per 10 seconds. **Value range:** there is a valid value range for each monitoring series, and any value outside the valid range is some abnormal value.

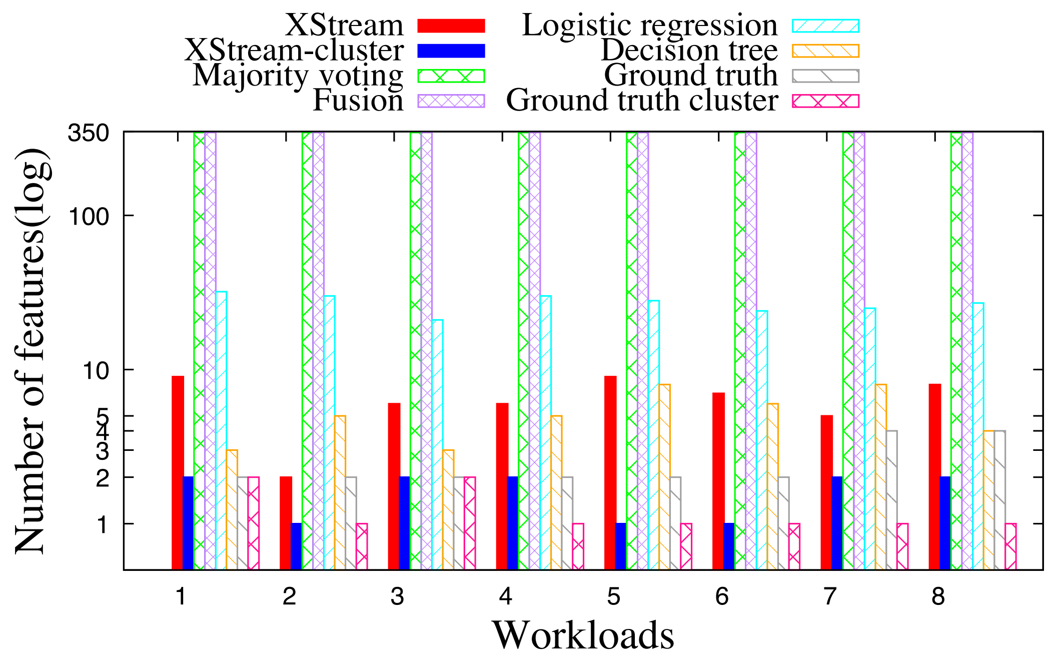


**Figure 5.17.** Conciseness comparison

In the materials category, we assume there are a number of machines consuming materials at the same time. These machines are producing different parts for one product. **Rate:** while monitoring event series have fixed rate, the interval between material recording events are not fixed because manufacturing step might have different length of durations. **Value range:** we use numeric value to denote the quality of materials, and there is a valid bar for the quality value: any value equal to or larger than the bar is satisfying the standard, otherwise it is sub-par.These configurations are summarized in Table 5.1.

| **Category** | Event Rate | Num. of types | **Schema** | Valid values |
|---|---|---|---|---|
| Monitoring | Fixed rate | 100 | $\{(Monitoringname, timestamp, monitoringvalue)\}$ | Valid range |
| Material | Non-Fixed rate | 100 | $\{(Machinename, timestamp, materialqualityvalue)\}$ | Valid bar |

**Table 5.1.** Simulation configurations.

### 5.7.3.2 Anomalies

With the above settings, we can generate all events with different types occurred during the manufacturing period for a specific product. For normal products, all events are generated

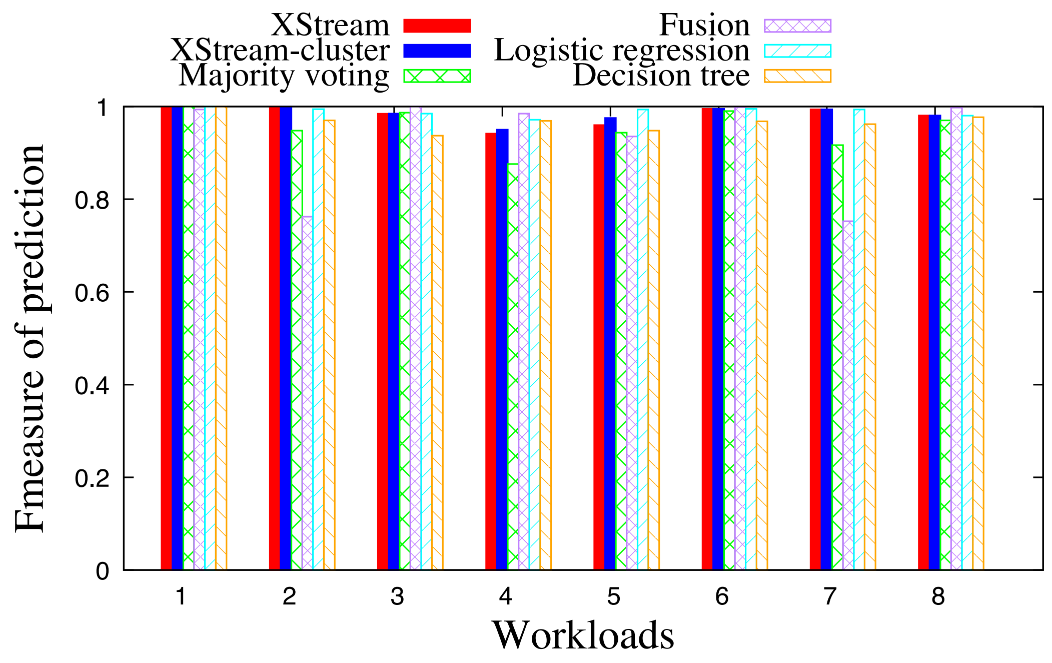


**Figure 5.18.** Prediction power comparison

strictly following the event rates, and all values for monitoring category fall into the valid range and all material values are equal to or above the quality bar.

For anomalies, we simulated two different types of products corresponding to the use cases discussed above. A product with **missing monitoring** issue lacks some monitoring measurements during its manufacturing period. If a product has sub-par material issue, in the simulation of its manufacturing events, some values of its material recording events are under the quality bar.

In this usecase, CEP queries will be used to track the progress of each product. The historical query results will be archived. The analysis will be triggered when customers report quality problems. After a product is claimed to be problematic, the query result for this product will be compared against a few products which are guaranteed to be of good quality. Products have no claims during until expiration date are automatically labeled as normal. And the claimed product will be labeled as abnormal.

#### 5.7.3.3 Evaluation

For the supply chain management use case, we created six anomalies: the first three use cases are about missing monitoring, and the last three use cases are about sub-par materials.

Explanation quality results are shown in Fig 5.19. Our techniques are providing much better explanations for every use case. Fig 5.20 compares the conciseness of results, and again our algorithm beats other techniques and the size is always the same as that of the ground truth. The prediction results are listed in Fig 5.21, and our techniques provide results as good as state-of-the-art techniques.

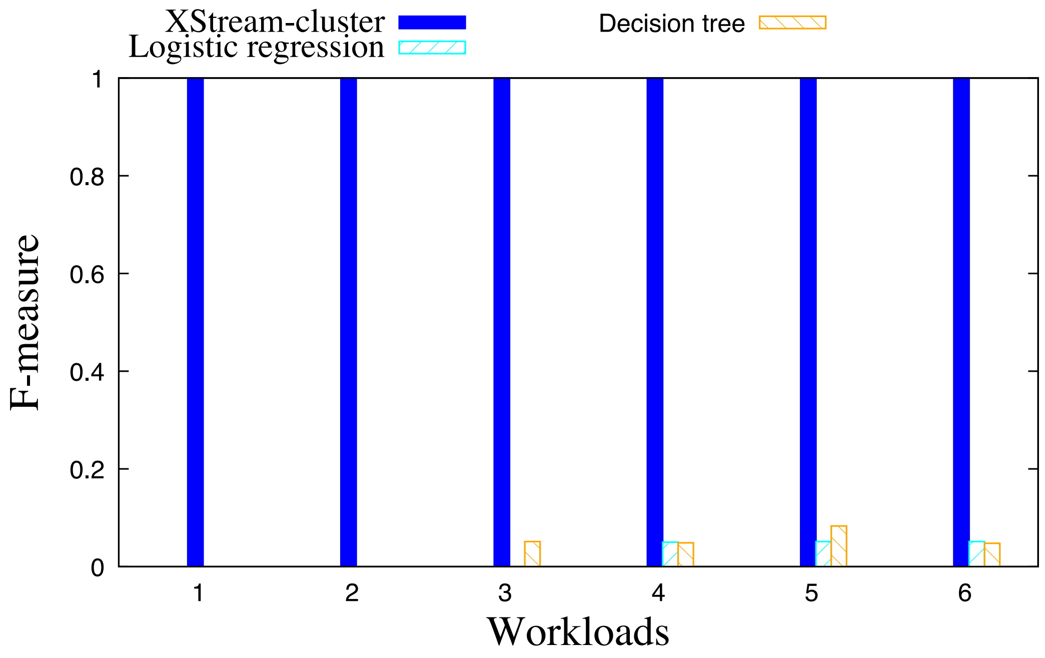


**Figure 5.19.** Supply chain management: consistency comparison

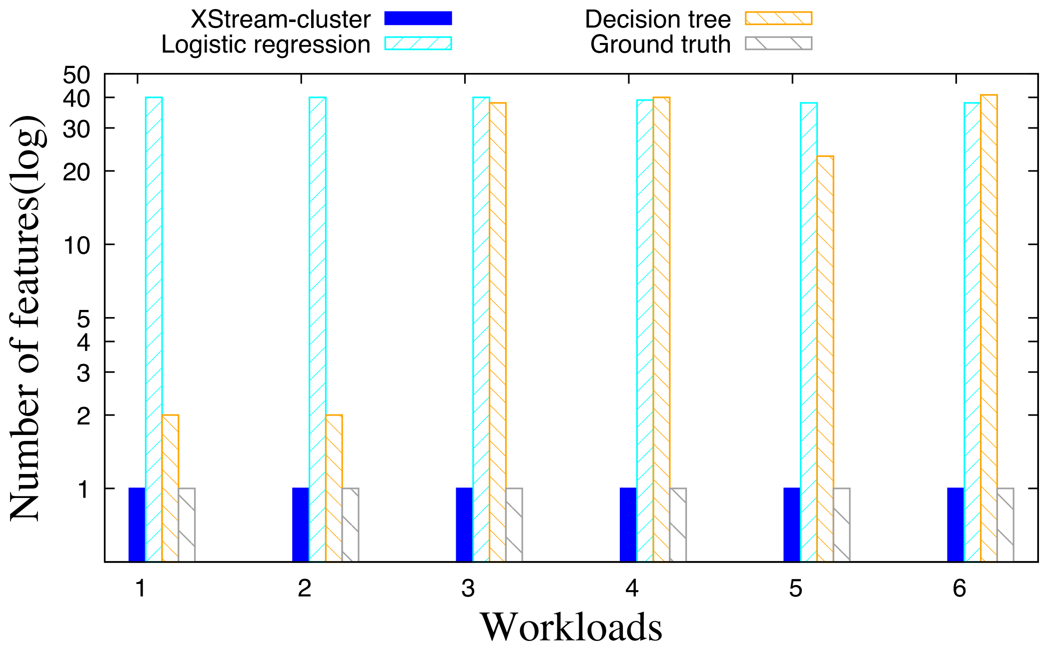


**Figure 5.20.** Supply chain management: conciseness comparison

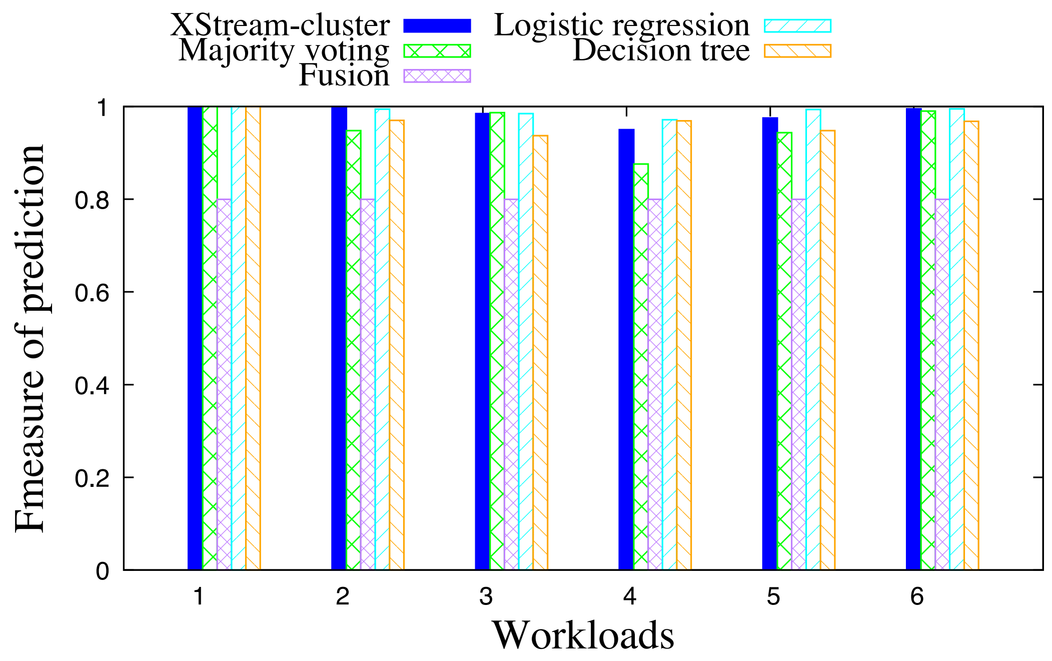


**Figure 5.21.** Supply chain management: prediction comparison

Effectiveness of the distance function: We finally demonstrate the effectiveness of our entropy-based distance function by comparing it with a set of existing distance functions [50] for time series: (1) Manhattan distance, (2) Euclidean distance, (3) DTW, (4) EDR, (5) ERP and (6) LCSS.

The results are shown in Figure 5.22. In each method, all available features are sorted by the distance function of choice in decreasing order. We measure the number of features retrieved from each sorted list in decreasing order in order to cover all the features in the ground truth, shown as the Y-axis. We see that our entropy distance is always the one using the minimum number of features to cover the ground truth. LCSS works well in the first two workloads, but it works poorly for workloads 3, 4, 5, and 6. This is because the ground truth features for the first two workloads have perfect separating power based on LCSS distance, while in other workloads they contain some noisy signals. So LCSS is not as robust as our distance function. Other distance functions always use large number of features.

Summary: Our explanation algorithm outperforms other techniques in consistency and conciseness while achieving comparable, high predication accuracy. Specifically, $EXstream$ improves consistency to other methods from 10.7% to 87.5% on average, and up to 100% in some cases. $EXstream$ is also more concise, reducing the number of features in an explanation 90.5% on average, up to 99.5% in some cases. $EXstream$ is as good as other techniques on prediction quality: its F-measure on prediction is only slightly worse than logistic regression by 0.4%, while it is 3.3% higher than majority voting, 6.1% percent higher than fusion, and 1.9% higher than decision tree.

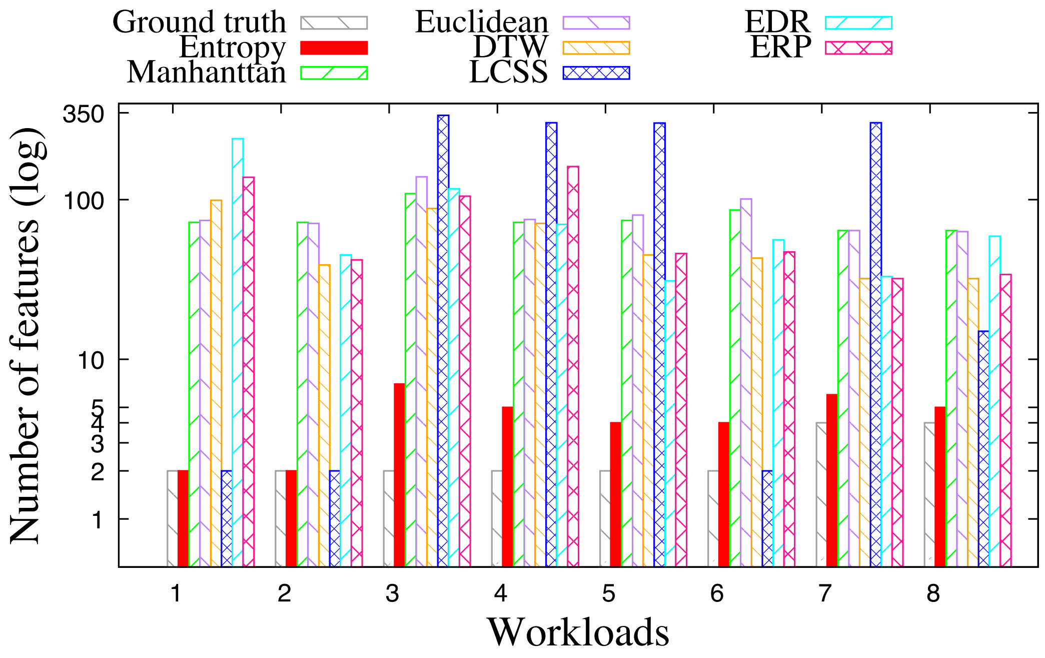


**Figure 5.22.** Distance function comparison

Our entropy distance function works better than existing distance functions on time series. It reduces the size of explanations by 94.6% on average, up to 97.2%, compared to other functions.

### 5.7.4 Efficiency of *EXstream*

We further evaluate the efficiency of *EXstream*. Our main result shows that our implementation is highly efficient: with 2000 concurrent monitoring queries, triggered explanation analysis returns explanations within half a minute and affects the performance only slightly, delaying events processing by only 0.4 second on average.

In the experiments, we first run 2000 monitoring queries for a few minutes, then trigger the explanation request every a few minutes. Each time we only trigger one explanation request, so there are no concurrent explanation requests.

In Figure 5.23, we show the number of delayed threads caused by the explanation function. Affected threads mean the monitoring thread having a delay more than 0.01 second in processing incoming events. We choose 0.01 second as the threshold because most events are processed within this range when no explanation analysis is triggered. In the figure we can see that, most use cases, only less than 25% of threads are affected. Only in Use case 3 , 26% of all threads are affected. In Use case 6, only 4.7% are affected. In Use case 8, no threads are affected. In summary, only a small portion of all monitoring queries are affected by the explanation function.

The blue bars in Figure 5.24 show how long the explanation engine runs to generate an explanation. The explanation engine runs fast, and all of them can complete in half a minute.

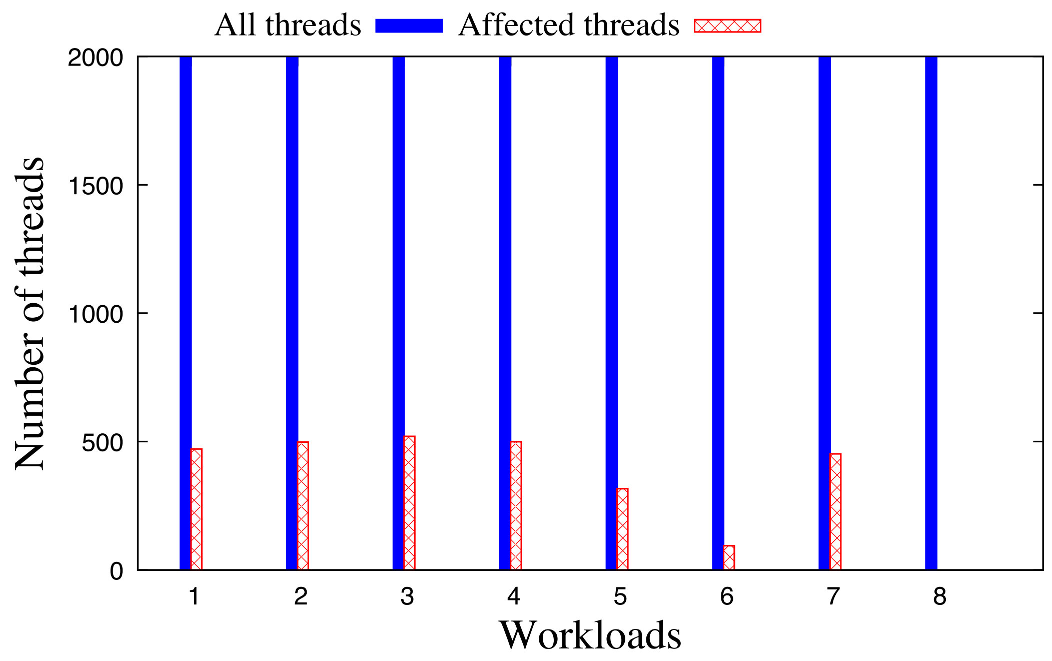


**Figure 5.23.** Total running threads vs. delayed threads

For our running example, it took the system admin more than a day to figure out the root causes. *EXstream*is quite efficient and helpful compared to the human needed time.

The red bars in Figure 5.24 show the number of seconds when the monitoring threads are delayed. We count the number of time units(seconds) when there are threads showing delay. It turns out the delays are not frequent, most of which are less than 10 seconds or so. As observed, these affected threads can catch up very quickly after the explanation triggered for some time.

The green bars in Figure 5.24 show how much time is delayedfor the delayed threads. The average delayed time among all the affected threads is around 0.4 second. In Use case 5, the average delay time is 0.24 second. For Use case 8, it is 0 second because no threads are affected. It means that, although some threads are affected, but they are still going ahead, and only slightly behind the latest event. For monitoring purposes, such delay should be acceptable.

In summary, after the explanation is triggered, a small portion of threads are affected slightly in a short period.

## 5.8 Related Work

In the previous section, we compared our entropy distance with a set of state-of-the-art distance functions [50] and compared our techniques with prediction techniques including decision trees and logistic regression [2]. In this section we survey broadly related work.

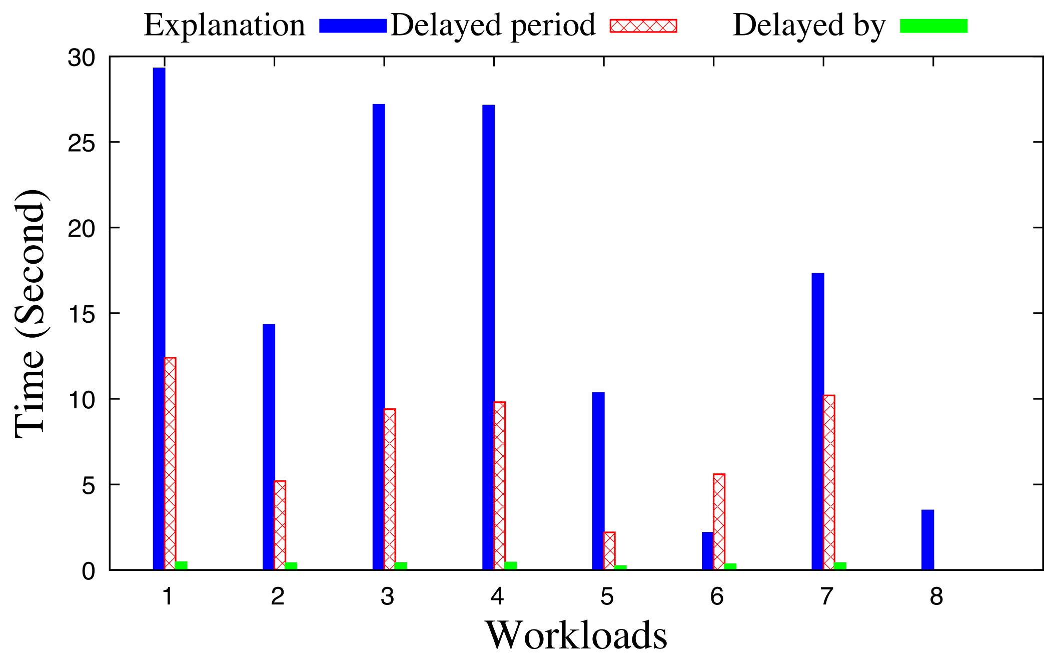


**Figure 5.24.** Explanation duration vs. affected duration vs. delayed distance

CEP systems: There are a number of CEP systems in the research community [15, 34, 1, 44, 48]. These systems focus on passive monitoring using CEP queries by providing either more powerful query languages or better evaluation performance. Existing CEP techniques do not produce explanations for anomalous observations.

Explaining outliers in SQL query results Scorpion [53] explains outliers in group-by aggregate queries. Users annotate outliers on the results of group-by queries, and then scorpion searches for predicates that remove these outliers while minimally affect the normal answers. It does not suit our problem because it works only for group-by aggregation queries and it searches through various subsets of the tuples that were used to compute the query answers. As shown for our example, Q1, the explanation of memory usage contention among different jobs cannot be generated from only those events that produced the monitoring results of Q1. Recent work [40] extends Scropion by supporting richer and insightful explanations by pre-computation and thus enables interactive explanation discovery. This work assumes a set of explanation templates given by the user and requires precomputation in a given database. Neither of the assumptions fits our problem setting.

Explaining outputs in iterative analytics: Recent work [13] focuses on tracking, maintaining, and querying lineage and "how" provenance in the context of arbitrary iterative data flows. It aims to create a set of recursively defined rules that determine which records in a data-parallel computation inputs, intermediate records, and outputs require explanation. It allows one to identify when (i.e., the points in the computation) and how a data collection changes, and provides explanations for only these few changes.

**Set-based distance function for time series.** Besides the lock-step and elastic distance functions we compared with, time series are also transformed into sets [36] for measurement.

However, the goal of the set-based function is to speed up the computation of existing elastic distance, so it is different from our entropy based distance function.

Anomaly detection: Common anomaly detection techniques [10, 12, 23, 22, 45] do not fit our problem setting. There are two main approaches. One is using a prediction model, which is learned on labeled or unlabeled data. Then incoming data is compared against with expected value by the model. If the difference is significant, the point or time series will be reported as outlier. The other approach is using distance functions, and outliers are those points or time series far from normal values. Both approaches report only outliers, but not the reasons (explanations) why they occur.

## 5.9 Conclusions

In this chapter, we present *EXstream*, a system that provides high-quality explanations for anomalous behaviors that users annotate on CEP-based monitoring results. Formulated as a submodular optimization problem, which is hard to solve, we provide a new approach that integrates a new entropy-based distance function and effective feature ranking and filtering methods. Evaluation results show that *EXstream* outperforms existing techniques significantly in conciseness and consistency, while achieving comparable high prediction power and retaining a highly efficient implementation of a data stream system.

## 5.10 Other ways of finding minimum explanations

### 5.10.1 Optimization with penalty

Let $\theta^i$ denote a selection vectors, where
$\theta^i = (x_1, x_2, \ldots, x_p), \exists i, 1 \le i \le p, \forall j \ne i \wedge 1 \le j \le p, x_i = 1, x_j = 0$

$$\Theta^q = \begin{bmatrix} \theta^{m_1} \\ \theta^{m_2} \\ \vdots \\ \theta^{m_q} \end{bmatrix}$$

So $\Theta^q$ is a $q \times p$ matrix, and $F_j = \Theta^q \times V_{C_j}$, which is a vector with $q$ coordinates.

$\mathbb{T} = \{\Theta^q = \begin{bmatrix} \theta^{m_1} \\ \theta^{m_2} \\ \vdots \\ \theta^{m_q} \end{bmatrix} | \theta^i = (x_1, x_2, \ldots, x_p), 1 \le i, j \le p, j \ne i, x_i = 1, x_j = 0, 1 \le m_1 < m_2 < \ldots < m_q \le p\}$

The objective function is used to find a subset of features as $F$ from all the features of $V_{C_j}$, such that the distance between $F_0$ and $F_1$ is maximum while $|F|$ is minimal. We tried quite a few functions, and none of them works.

$$\underset{\Theta^q \in \mathbb{T}}{\operatorname{argmax}} ||\Theta^q \cdot (V_{C_0} - V_{C_1})||_2^2 - \lambda ||\Theta^q||_1 \tag{5.5}$$

We choose **euclidean distance** for the object because it tends to select the most distinguished features. For one instance, there is no difference for the two distances. With two annotated instances, if one instance selects (a=0.8, b=0.5) while another instance selects(a=0.9,c=0.4), which set of feature should we consider? The Mahanttan distance would be the same. While the Euclidean distance prefers the latter. Or even a different case, (a=0.8, b=0.5) vs. (a=0.9,c=0.3).

While for the penalty term, we choose $L_1$ norm because it tends to return sparser results.

Function 5.5 is convex instead of concave according to Jesen's inequality[27]. Being convex means a local minimum value is a global minimum value, while being concave means a local maximum value is a global minimum value.

Based on Jesen's inequality, we can prove the convexity of Function 5.5 by proving Function 5.6 for any given a,

$$0 \le a \le 1$$

$$af(\Theta_1) + (1-a)f(\Theta_2) \ge f(a\Theta_1 + (1-a)\Theta_2) \tag{5.6}$$

**Proof Sketch:**

Left side:

$$\begin{aligned} & af(\Theta_1) + (1-a)f(\Theta_2) \\ &= a||\Theta_1 \cdot (V_{C_0} - V_{C_1})||_2^2 - a\lambda||\Theta_1||_1 \\ &+(1-a)(||\Theta_2 \cdot (V_{C_0} - V_{C_1})||_2^2 - \lambda||\Theta_2||_1) \\ &= a||\Theta_1 \cdot (V_{C_0} - V_{C_1})||_2^2 + (1-a)(||\Theta_2 \cdot (V_{C_0} - V_{C_1})||_2^2) \\ &-\lambda(||\Theta_1||_1 + (1-a)\lambda||\Theta_2||_1) \end{aligned}$$

Right side:

$$\begin{aligned} & f(a\Theta_1 + (1-a)\Theta_2) \\ &= ||(a\Theta_1 + (1-a)\Theta_2) \cdot (V_{C_0} - V_{C_1})||_2^2 - \lambda||a\Theta_1 + (1-a)\Theta_2||_1 \end{aligned}$$

Let us remove the

$$-\lambda||a\Theta_1 + (1-a)\Theta_2||_1$$

on both sides.

Then we are trying to prove

$$\begin{aligned} & ||a\Theta_1 \cdot (V_{C_0} - V_{C_1})||_2^2 + ||(1-a)\Theta_2 \cdot (V_{C_0} - V_{C_1})||_2^2 \\ & \ge ||(a\Theta_1 + (1-a)\Theta_2) \cdot (V_{C_0} - V_{C_1})||_2^2 \end{aligned}$$

Then if we can prove for any dimension $d$ of $V_{C_0} - V_{C_1}$ this inequation holds, this proof is done. Let $V$ to denote $V_{C_0} - V_{C_1}$ for short.

$$a(\Theta_1 \cdot V)_d^2 + (1-a)(\Theta_2 \cdot V)_d^2$$
$$\geq ((a\Theta_1 + (1-a)\Theta_2) \cdot V)_d^2$$

$$a(\Theta_1 \cdot V)_d^2 + (1-a)(\Theta_2 \cdot V)_d^2$$
$$\geq a^2(\Theta_1 \cdot V)_d^2 + (1-a)^2(\Theta_2 \cdot V)_d^2 + 2a(1-a)(\Theta_1 \cdot V)_d(\Theta_2 \cdot V)$$

$$a(\Theta_1 \cdot V)_d^2 + (1-a)(\Theta_2 \cdot V)_d^2$$
$$-a^2(\Theta_1 \cdot V)_d^2 - (1-a)^2(\Theta_2 \cdot V)_d^2 - 2a(1-a)(\Theta_1 \cdot V)_d(\Theta_2 \cdot V) \geq 0$$

$$(a-a^2)(\Theta_1 \cdot V)_d^2 + (a-a^2)(\Theta_2 \cdot V)_d^2$$
$$-2(a-a^2)(\Theta_1 \cdot V)_d(\Theta_2 \cdot V) \geq 0$$

$$(a-a^2)((a\Theta_1 - (1-a)\Theta_2) \cdot V)_d^2 \geq 0$$

Given $0 \leq a \leq 1$, so $a - a^2 \geq 0$. And $((a\Theta_1 - (1-a)\Theta_2) \cdot V)_d^2$ is obviously larger than or euqal to 0. So the proof is done.

□

Then we designed the concave version shown in Function 5.7 The problem of Function 5.7 is the maximum is reached when $\Theta^q$ is 0.

$$\underset{\Theta^q \in \mathbb{T}}{\text{argmax}} ||\Theta^q \cdot (V_{C_0} - V_{C_1})||_2^2 - \lambda||\Theta^q||_2^2 (\lambda \geq 1) \tag{5.7}$$

So we adjusted it as Function 5.8. It is required that $(\lambda_1 \geq 1, \lambda_1 > \lambda_2 > 0)$. In practice, in the constraint set, $||\Theta^q||_2^2 = ||\Theta^q||_1$, so the meaning of Function 5.8 is: it only selects features with distance larger than $|\lambda_1 - \lambda_2|$. It can be proved. So the maximum value will be reached when all features with larger distance are selected while all features with smaller distance are filtered.

$$\underset{\Theta^q \in \mathbb{T}}{\text{argmax}} ||\Theta^q \cdot (V_{C_0} - V_{C_1})||_2^2 - \lambda_1||\Theta^q||_2^2 + \lambda_2||\Theta^q||_1 \tag{5.8}$$

In summary, those optimizations either cannot find optimal solution or the results are equal to uninteresting thresholds.

# CHAPTER 6
# CONCLUSION AND FUTURE WORK

This thesis makes contributions to CEP technology on two types of tasks: passive monitoring and proactive monitoring. And there is space to improve CEP along the two directions.

## 6.1 Conclusion

In Chapter 3, to apply CEP technology over event streams with imprecise timestamps, I present a temporal model that assigns a time interval to each event to represent all of its possible occurrence times and formal semantics of pattern matching under the new model. Under the uncertain temporal model, I further present two evaluation frameworks, a point-based framework which processes converted events with point timestamp, and an event-based framework which processes patterns over events with intervals directly. Optimizations are also designed for these frameworks to improve evaluation performance. My solution achieves high efficiency for workloads tested using both both real traces and synthetic datasets. While existing systems do not support this type of streams, the throughput of my system is up to tens of thousands of events per second for MapReduce case study. This contribution makes CEP techniques applicable for streams with imprecise timestamps.

In Chapter 4, another contribution for the passive monitoring is presented: I analyze expensive queries in CEP, find performance bottlenecks by analyzing their runtime complexity, and propose a set of optimizations to improve the evaluation performance significantly. The factors resulting in expensive queries include Kleene closure patterns, flexible event selection strategies, and imprecise timestamps. The runtime complexity of each language component is analyzed and two performance bottlenecks are found: Kleene closure under the most flexible event selection strategy and confidence computation in the case of imprecise timestamps. I break query evaluation into two parts to solve the first bottleneck: pattern matching, which can be shared by many matches and result construction. With optimizations for the shared pattern matching, exponential cost is cut to polynomial time and even close-to-linear. A dynamic programming algorithm is designed to improve performance for the second bottleneck. State-of-the-art systems suffer poor performance in microbenchmark results, while my system can provide 2 to 10 orders of magnitude improvement. The throughput is over 1 million events per second for a Hadoop cluster monitoring case study.

In Chapter 5, I present contribution to proactive monitoring: *EXstream*, a system that provides high-quality explanations for anomalous behaviors that users annotate on CEP-based monitoring results. Formulated as a submodular optimization problem, which is hard to solve, we provide a new approach that integrates a new entropy-based distance function and effective feature ranking and filtering methods. Evaluation results show that $EXstream$ outperforms existing techniques significantly in conciseness and consistency, while achieving comparable high prediction power and retaining a highly efficient implementation of a data stream system. Our entropy distance function outperforms state-of-the-art distance functions on time series by reducing the features considered by 94.6%. $EXstream$ significantly outperforms existing technologies in consistency and conciseness of explanations while achieving comparable, high predication accuracy. Specifically, it outperforms others by improving consistency from 10.7% to 87.5% on average, and reduces 90.5% of features on average to ensure conciseness. Our implementation is also efficient: with 2000 concurrent monitoring queries, the triggered explanation analysis returns explanations within half a minute and affects the performance only slightly, delaying events processing by 0.4 second on average.

## 6.2 Future work

CEP systems have been deployed in variety of areas such as financial services, logistics, monitoring and so on because of its high performance in matching events against complex queries. There are possibilities to extend and enhance CEP in different directions in both passive monitoring and proactive monitoring. The follows are a few interesting ones.

### 6.2.1 Passive monitoring

CEP in passive monitoring has been studies for years, performance and functionalities are two key directions.

Distributed processing will be the key to improve the performance to next level. While existing systems mainly focus on speeding up the throughput on a single server, there is not much study about CEP in a distributed architecture. Load balancing for stream events between nodes, synchronization of query instances and scheduling of concurrent queries need to be considered in the distributed environment.

User friendly language will make CEP more applicable to general consumers. Currently CEP aims to server for professional IT staff. With the development of new technologies such as Internet of things, smart home and so on, general users have the needs to process more complicated information. If the query language is more user friendly, like using drag and drop on a touch screen, CEP has the potential to be the engine for personal information processing.

### 6.2.2 Proactive monitoring

Explaining anomalies is only the beginning of proactive monitoring. There are many directions to go and technical challenges to be solved.

Data collection is fundamental for later processing. How does users know what data stream they need to collect to ensure that the features that can provide an explanation are present? How does users realize they do not collect enough data to provide an explanation?

These problems need to be solved to make data collection for proactive monitoring both efficient and effective.

Feature space generation is another key step to generate high quality explanations. How should the system derive features based on raw data? What kind of aggregate functions should be applied? How should user to set the size of sliding window? There are many possibilities to construct better features.

User study is a good way to validate the quality of generated explanations. It will provide useful feedback to improve the techniques on both conciseness and consistencies. With those improvements, explanations could be more user friendly.

# BIBLIOGRAPHY

[1] Abadi, Daniel J, Carney, Don, Çetintemel, Ugur, Cherniack, Mitch, Convey, Christian, Lee, Sangdon, Stonebraker, Michael, Tatbul, Nesime, and Zdonik, Stan. Aurora: a new model and architecture for data stream management. *The VLDB JournalThe International Journal on Very Large Data Bases 12*, 2 (2003), 120–139.

[2] Aggarwal, Charu C. *Data Mining: The Textbook*. Springer Publishing Company, Incorporated, 2015.

[3] Agrawal, Jagrati, Diao, Yanlei, Gyllstrom, Daniel, and Immerman, Neil. Efficient pattern matching over event streams. In *SIGMOD* (New York, NY, USA, 2008), ACM, pp. 147–160.

[4] Akdere, Mert, Çetintemel, Ugur, and Tatbul, Nesime. Plan-based complex event detection across distributed sources. *PVLDB 1*, 1 (2008), 66–77.

[5] Ali, Mohamed H., Gerea, Ciprian, Raman, Balan, Sezgin, Beysim, Tarnavski, Tiho, Verona, Tomer, Wang, Ping, Zabback, Peter, Kirilov, Anton, Ananthanarayan, Asvin, Lu, Ming, Raizman, Alex, Krishnan, Ramkumar, Schindlauer, Roman, Grabs, Torsten, Bjeletich, Sharon, Chandramouli, Badrish, Goldstein, Jonathan, Bhat, Sudin, Li, Ying, Nicola, Vincenzo Di, Wang, Xianfang, Maier, David, Santos, Ivo, Nano, Olivier, and Grell, Stephan. Microsoft cep server and online behavioral targeting. *PVLDB 2*, 2 (2009), 1558–1561.

[6] Barga, Roger S, Goldstein, Jonathan, Ali, Mohamed, and Hong, Mingsheng. Consistent streaming through time: A vision for event stream processing. *arXiv preprint cs/0612115* (2006).

[7] Barham, Paul, Donnelly, Austin, Isaacs, Rebecca, and Mortier, Richard. Using magpie for request extraction and workload modelling. In *OSDI* (2004), pp. 259–272.

[8] Boulon, Jerome, Konwinski, Andy, Qi, Runping, Rabkin, Ariel, Yang, Eric, and Yang, Mac. Chukwa, a large-scale monitoring system. In *Proceedings of Conference on Cloud Computing and Its Applications (CCA)* (2008), vol. 8.

[9] Bhlen, Michael H., and Jensen, Christian S. Temporal data model and query language concepts. *Encyclopedia of Information Systems 4* (2003), 437–453.

[10] Cao, Lei, Wang, Jiayuan, and Rundensteiner, Elke A. Sharing-aware outlier analytics over high-volume data streams. In *Proceedings of the 2016 International Conference on Management of Data* (2016), ACM, pp. 527–540.

[11] Cao, Zhao, Sutton, Charles, Diao, Yanlei, and Shenoy, Prashant J. Distributed inference and query processing for rfid tracking and monitoring. *PVLDB 4*, 5 (2011), 326–337.

[12] Chandola, Varun, Banerjee, Arindam, and Kumar, Vipin. Anomaly detection: A survey. *ACM Computing Surveys (CSUR) 41*, 3 (2009), 15.

[13] Chothia, Zaheer, Liagouris, John, McSherry, Frank, and Roscoe, Timothy. Explaining outputs in modern data analytics. *Proceedings of the VLDB Endowment 9*, 4 (2015).

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

[15] Demers, Alan J., Gehrke, Johannes, Panda, Biswanath, Riedewald, Mirek, Sharma, Varun, and White, Walker M. Cayuga: A general purpose event monitoring system. In *CIDR* (2007), pp. 412–422.

[16] Ding, Luping, Chen, Songting, Rundensteiner, Elke A., Tatemura, Jun'ichi, Hsiung, Wang-Pin, and Candan, K. Selçuk. Runtime semantic query optimization for event stream processing. In *ICDE* (2008), pp. 676–685.

[17] Dyreson, Curtis E., and Snodgrass, Richard T. Supporting valid-time indeterminacy. *ACM Trans. Database Syst. 23*, 1 (1998), 1–57.

[18] Faloutsos, Christos, Ranganathan, Mudumbai, and Manolopoulos, Yannis. *Fast subsequence matching in time-series databases*, vol. 23. ACM, 1994.

[19] Fayyad, Usama, and Irani, Keki B. Multi-interval discretization of continuous-valued attributes for classification learning.

[20] Feige, Uriel, Mirrokni, Vahab S, and Vondrak, Jan. Maximizing non-monotone submodular functions. *SIAM Journal on Computing 40*, 4 (2011), 1133–1153.

[21] Ganglia monitoring system. `http://ganglia.sourceforge.net/`.

[22] Gupta, Manish, Gao, Jing, Aggarwal, Charu, and Han, Jiawei. Outlier detection for temporal data. *Synthesis Lectures on Data Mining and Knowledge Discovery 5*, 1 (2014), 1–129.

[23] Huang, Hao, and Kasiviswanathan, Shiva Prasad. Streaming anomaly detection using randomized matrix sketching. *Proceedings of the VLDB Endowment 9*, 3 (2015), 192–203.

[24] Johnson, Theodore, Muthukrishnan, S., and Rozenbaum, Irina. Monitoring regular expressions on out-of-order streams. In *ICDE* (2007), pp. 1315–1319.

[25] Koskinen, Eric, and Jannotti, John. Borderpatrol: isolating events for black-box tracing. In *EuroSys* (2008), pp. 191–203.

[26] Krishnamurthy, Sailesh, Wu, Chung, and Franklin, Michael J. On-the-fly sharing for streamed aggregation. In *SIGMOD Conference* (2006), pp. 623–634.

[27] Kuczma, Marek. *An introduction to the theory of functional equations and inequalities: Cauchy's equation and Jensen's inequality*. Springer Science & Business Media, 2009.

[28] Lam, Louisa, and Suen, SY. Application of majority voting to pattern recognition: an analysis of its behavior and performance. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans 27*, 5 (1997), 553–568.

[29] Liu, Mo, Li, Ming, Golovnya, Denis, Rundensteiner, Elke A., and Claypool, Kajal T. Sequence pattern query processing over out-of-order event streams. In *ICDE* (2009), pp. 784–795.

[30] Liu, Mo, Rundensteiner, Elke, Dougherty, Dan, Gupta, Chetan, Wang, Song, Ari, Ismail, and Mehta, Abhay. High-performance nested cep query processing over event streams. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on* (2011), IEEE, pp. 123–134.

[31] Lo, Eric, Kao, Ben, Ho, Wai-Shing, Lee, Sau Dan, Chui, Chun Kit, and Cheung, David W. Olap on sequence data. In *SIGMOD Conference* (2008), pp. 649–660.

[32] Luckham, D.C. *Event Processing for Business: Organizing the Real-Time Enterprise*. Wiley, 2011.

[33] Massie, Matt, Li, Bernard, Nicholes, Brad, Vuksan, Vladimir, Alexander, Robert, Buchbinder, Jeff, Costa, Frederiko, Dean, Alex, Josephsen, Dave, Phaal, Peter, and Pocock, Daniel. Monitoring with ganglia, 11 2012.

[34] Mei, Yuan, and Madden, Samuel. Zstream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD Conference* (2009), pp. 193–206.

[35] Mozafari, Barzan, Zeng, Kai, and Zaniolo, Carlo. High-performance complex event processing over xml streams. In *SIGMOD Conference* (2012), pp. 253–264.

[36] Peng, Jinglin, Wang, Hongzhi, Li, Jianzhong, and Gao, Hong. Set-based similarity search for time series. In *Proceedings of the 2016 International Conference on Management of Data* (2016), ACM, pp. 2039–2052.

[37] Pochampally, Ravali, Das Sarma, Anish, Dong, Xin Luna, Meliou, Alexandra, and Srivastava, Divesh. Fusing data with correlations. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (2014), ACM, pp. 433–444.

[38] Ré, Christopher, Letchner, Julie, Balazinska, Magdalena, and Suciu, Dan. Event queries on correlated probabilistic streams. In *SIGMOD* (2008), pp. 715–728.

[39] Ren, Kai, Kwon, YongChul, Balazinska, Magdalena, and Howe, Bill. Hadoop's adolescence. *PVLDB 6*, 10 (2013), 853–864.

[40] Roy, Sudeepa, Orr, Laurel, and Suciu, Dan. Explaining query answers with explanation-ready databases. *Proceedings of the VLDB Endowment 9*, 4 (2015), 348–359.

[41] Sadri, Reza, Zaniolo, Carlo, Zarkesh, Amir, and Adibi, Jafar. Expressing and optimizing sequence queries in database systems. *ACM Transactions on Database Systems (TODS) 29*, 2 (2004), 282–318.

[42] Schultz-Møller, Nicholas Poul, Migliavacca, Matteo, and Pietzuch, Peter. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems* (2009), ACM, p. 4.

[43] Srivastava, Utkarsh, and Widom, Jennifer. Flexible time management in data stream systems. In *PODS* (2004), pp. 263–274.

[44] StreamSQL Team. StreamSQL: a data stream language extending SQL. `http://blogs.streamsql.org/`.

[45] Tran, Luan, Fan, Liyue, and Shahabi, Cyrus. Distance based outlier detection for data streams. *Proceedings of the VLDB Endowment 9*, 4 (2015), 1089–1100.

[46] Tran, Thanh, Sutton, Charles, Cocci, Richard, Nie, Yanming, Diao, Yanlei, and Shenoy, Prashant. Probabilistic inference over rfid streams in mobile environments. In *ICDE* (2009).

[47] Tucker, Peter A., Maier, David, Sheard, Tim, and Stephens, Paul. Using punctuation schemes to characterize strategies for querying over data streams. *IEEE Trans. Knowl. Data Eng. 19*, 9 (2007), 1227–1240.

[48] Wang, Di, Rundensteiner, Elke A., and Ellison, Richard T. Active complex event processing over event streams. *PVLDB 4*, 10 (2011), 634–645.

[49] Wang, Di, Rundensteiner, Elke A., Ellison, Richard T., and Wang, Han. Probabilistic inference of object identifications for event stream analytics. In *EDBT* (2013), pp. 513–524.

[50] Wang, Xiaoyue, Mueen, Abdullah, Ding, Hui, Trajcevski, Goce, Scheuermann, Peter, and Keogh, Eamonn. Experimental comparison of representation methods and distance measures for time series data. *Data Mining and Knowledge Discovery 26*, 2 (2013), 275–309.

[51] White, Walker M., Riedewald, Mirek, Gehrke, Johannes, and Demers, Alan J. What is "next" in event processing? In *PODS* (2007), pp. 263–272.

[52] Wu, Eugene, Diao, Yanlei, and Rizvi, Shariq. High-performance complex event processing over streams. In *SIGMOD* (2006), pp. 407–418.

[53] Wu, Eugene, and Madden, Samuel. Scorpion: explaining away outliers in aggregate queries. *Proceedings of the VLDB Endowment 6*, 8 (2013), 553–564.

[54] Yang, Di, Rundensteiner, Elke A, and Ward, Matthew O. A shared execution strategy for multiple pattern mining requests over streaming data. *Proceedings of the VLDB Endowment 2*, 1 (2009), 874–885.

[55] Zhang, Haopeng, Diao, Yanlei, and Immerman, Neil. Recognizing patterns in streams with imprecise timestamps. *PVLDB 3*, 1 (2010), 244–255.

[56] Zhang, Haopeng, Diao, Yanlei, and Immerman, Neil. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD* (2014), ACM.