

# Lazy Evaluation Methods for Detecting Complex Events

Ilya Kolchinsky  
Technion, Israel Institute of  
Technology

Izchak Sharfman  
Technion, Israel Institute of  
Technology

Assaf Schuster  
Technion, Israel Institute of  
Technology

## ABSTRACT

The goal of Complex Event Processing (CEP) systems is to efficiently detect complex patterns over a stream of primitive events. A pattern of particular significance is a sequence, where we are interested in identifying that a number of primitive events have arrived on the stream in a predefined order. Many popular CEP systems employ Non-deterministic Finite Automata (NFA) arranged in a chain topology to detect such sequences. Existing NFA-based mechanisms incrementally extend previously observed *prefixes* of a sequence until a match is found. Consequently, each newly arriving event needs to be processed to determine whether a new prefix is to be initiated or an existing one extended. This approach may be very inefficient when events at the beginning of the sequence are very frequent.

We address the problem by introducing a lazy evaluation mechanism that is able to process events in descending order of selectivity. We employ this mechanism in a chain topology NFA, which waits until the most selective event in the sequence arrives and then adds events to partial matches according to a predetermined order of selectivity. In addition, we propose a tree topology NFA that does not require the selectivity order to be defined in advance. Finally, we experimentally evaluate our mechanism on real-world stock trading data, demonstrating a performance gain of two orders of magnitude, with significantly reduced memory resource requirements.

## Categories and Subject Descriptors

H.2.4 [Database Management]: Systems

## General Terms

Algorithms, Design, Performance

## Keywords

Stream Processing, Complex Event Processing, Lazy Evaluation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [Permissions@acm.org](mailto:Permissions@acm.org).

DEBS '15, June 29 - July 3, 2015, Oslo, Norway.

©2015 ACM. ISBN 978-1-4503-3286-6/15/06\$15.00.

DOI: <http://dx.doi.org/10.1145/2675743.2771832>

## 1. INTRODUCTION

Complex Event Processing (CEP) is an emerging field with important applications for real-time systems. The goal of CEP systems is to detect predefined patterns over a stream of primitive events. Examples of applications of CEP systems include financial services [13], RFID-based inventory management [29], and click stream analysis [26]. A pattern of particular interest is a sequence, where we are interested in detecting that a number of primitive events have arrived on the stream in a given order.

As an example of a sequence pattern, consider the following:

*EXAMPLE 1. A securities trading firm would like to analyze a real-time stream of stock price data in order to identify trading opportunities. The primitive events arriving on the stream are price quotes for the various stocks. An event of the form  $x_{p=y}^n$  denotes that the price of stock  $x$  has changed to  $y$ , where  $n$  is a running counter of the events for stock  $x$  (an event also includes a timestamp, omitted from the notation for brevity's sake). The trading firm would like to detect a sequence consisting of the events  $a_{p=p_1}$ ,  $b_{p=p_2}$ , and  $c_{p=p_3}$  occurring within an hour, where  $p_1 < p_2 < p_3$ .*

Modern CEP systems are required to process growing rates of incoming events. In addition, as this technology becomes more prevalent, languages for defining complex event patterns are becoming more expressive. A popular approach is to compile patterns expressed in a declarative language into Non-deterministic Finite state Automata (NFAs), which are in turn used by the event processing engine. Wu et al. [30] proposed the SASE system, which is based on a language that supports logic operators, sequences and time windows. The authors describe how a complex pattern formulated using this language is translated into an NFA consisting of a finite set of states and conditional transitions between them. Transitions between states are triggered by the arrival of an appropriate event on the stream. At each point in time, an instance of the state machine is maintained for every prefix of the pattern detected in the stream up to that point. In addition, a data structure referred to as the *match buffer* holds the primitive events constituting the match prefix. Gyllstrom et al. [22] propose additional operators for SASE, such as iterations and aggregates. Demers et al. [13, 14] describe Cayuga, a general purpose event monitoring system, based on a CEL language. It employs non-deterministic automata for event evaluation, supporting typical SQL operators and constructs. Tesla [11] extends previous works by offering fully customizable policies for event detection and

consumption. NextCEP [27] enables distributed evaluation using NFAs in clustered environments.

An NFA detects sequences by maintaining at every point in time all the observed prefixes of the sequence until a match is detected. As an example, consider the following stream of events:  $a_{p=3}^1, a_{p=5}^2, a_{p=8}^3, b_{p=7}^1, b_{p=13}^2, c_{p=9}^1$ . In this case, after the first three events have arrived,  $\{a^1\}$ ,  $\{a^2\}$  and  $\{a^3\}$  are match prefixes for the pattern described in Example 1. All these prefixes must be maintained by the NFA at this point in time, since all of them may eventually result in a match. After the first five events have arrived, the NFA must maintain five match prefixes (all combinations of  $a$  events and  $b$  events except for  $\{a^3b^1\}$ ). Finally, after the last event is received, the NFA detects two sequences matching the pattern,  $\{a_1b_1c_1\}$  and  $\{a_2b_1c_1\}$ .<sup>1</sup>

NFA based matching mechanisms are most commonly implemented by constructing partial matches according to the order of events in the sequence (i.e., every partial match is a prefix of a match). We refer to this prefix detection strategy as an “eager” strategy, since every incoming event is processed upon arrival in order to determine whether it starts a new prefix or extends an existing one. When the first events in a sequence pattern are very frequent, the NFA must maintain a large number of match prefixes that may not lead to any matches. Since the number of match prefixes to be kept can grow exponentially with the length of the sequence, such an approach may be very inefficient in terms of memory and computational resources.

In this paper we propose a new NFA based matching mechanism that overcomes this drawback. The proposed mechanism constructs partial matches starting from the most selective (i.e., least frequent) event, rather than from the first event in the sequence. In addition, partial matches are extended by adding events in descending order of selectivity (rather than according to their order in the sequence). This not only minimizes the number of partial matches held in memory, but also reduces computation time, since there are fewer partial matches to extend when processing a given event.

Our proposed solution relies on a lazy evaluation mechanism that can either process an event upon arrival or store it in a buffer, referred to as the *input buffer*, to be processed at a later time if necessary. To enable efficient search and retrieval of events from the input buffer, a new edge property called *scoping parameters* is introduced. In addition, we present two new types of NFA that make use of the input buffer and scoping parameters to detect sequence patterns; we call these types a chain NFA and a tree NFA.

A chain NFA requires specifying the selectivity order of the events in the sequence. For example, to construct an automaton for detecting the sequence  $a, b, c$ , it is necessary to specify that  $b$  is expected to be the most frequent event, followed by  $a$ , which is expected to be less frequent, followed by  $c$ , which is expected to be the least frequent.

<sup>1</sup>The consumption policy is important for the semantics of an event definition language. It specifies how to handle a particular event once it is included in a match, i.e., whether it can be reused for other matches, or should be discarded. For the purpose of our discussion in this work, we assume a reuse consumption policy, which means that an event instance can be included in an unlimited number of matches [16].

A tree NFA also employs lazy evaluation, but it *does not* require specifying the selectivity order of the events in the sequence. Instead, it computes the selectivity order at each step in an ad hoc manner.

We experimentally evaluate our mechanism on real-world stock trading data. The results demonstrate that the tree NFA matching mechanism improves run-time performance by two orders of magnitude in comparison to existing solutions, while significantly reducing memory requirements. It is also shown that for every stream of events, a tree NFA is at least as efficient as the best performing chain NFA.

The remainder of the paper is organized as follows. Section 2 describes related work. Section 3 briefly describes the eager NFA evaluation framework. It also provides the terminology and notations used throughout the paper. In Section 4 we introduce the concepts and ideas of lazy evaluation, accompanied by intuitive explanations and examples. Formal definitions presented there prepare the ground for the rest of the paper. In Section 5 we proceed to describe how a lazy *chain NFA* can be constructed using given frequencies of the participating events. We present a lazy *tree NFA* in Section 6. Section 7 contains the experimental evaluation. Section 8 summarizes the paper.

## 2. RELATED WORK

The detection of complex events over streams has become a very active research field in recent years [12]. The earliest systems designed for solving this problem fall under the category of Data Stream Management Systems. Those systems are based on SQL-like specification languages and focus on processing data coming from continuous, usually multiple input streams. Examples include NiagaraCQ [10], TelegraphCQ [9] and STREAM [21]. Later, the need to analyze event notifications of interesting situations – as opposed to generic data – was identified. Then, *complex event processing systems* were introduced. One example of an advanced CEP system is Amit [2], based on a strongly expressive detection language and capable of processing notifications received from different sources in order to detect patterns of interest. SPADE [17] is a declarative stream processing engine of System S. System S is a large-scale, distributed data stream processing middleware developed by IBM. It provides a computing infrastructure for applications that need to handle large scale data streams. Cayuga [7, 13, 14] is a general purpose, high performance, single server CEP system developed at Cornell University. Its implementation focuses on multi-query optimization methods.

Apart from the SASE language, on which our mechanism is based, many other event specification languages were proposed. SASE+ [22] is an expressive event processing language from the authors of SASE. This language extends the expressiveness of SASE by including iterations and aggregates. CQL [5] is an expressive SQL-based declarative language for registering continuous queries against streams and updatable relations. It allows creating transformation rules with a unified syntax for processing both information flows and stored relations. CEL (Cayuga Event Language) [7, 13, 14] is a declarative language used by the Cayuga system, supporting patterns with Kleene closure and event selection strategies, including partition contiguity and skip till next match. TESLA [11] is a newer declarative language, attempting to combine high expressiveness with a relatively small set of operators, achieving compactness and simplic-

ity. Even though our work focuses exclusively on sequence patterns, extensions to other operators are possible, including those added by the aforementioned languages.

Unlike most recently proposed CEP systems, which use non-deterministic finite automata (NFAs) to detect patterns, ZStream [24] uses tree-based query plans for the representation of query patterns. The careful design of the underlying infrastructure and algorithms makes it possible for ZStream to unify the representation of sequence, conjunction, disjunction, negation, and Kleene closure as variants of the join operator. While some of the ideas discussed in this work are close to ours, it is not based on state automata and employs matching trees instead.

Several works mention the concept of lazy evaluation in the context of event processing. In [4], the authors describe “plan-based evaluation,” where, similarly to our work, temporal properties of primitive events can be exploited to reduce network communication costs. The focus of their paper is on communication efficiency, whereas our goal is to reduce computational and memory requirements. [15] discusses a mechanism similar to ours, including the concept of buffering incoming events into an intermediate storage. However, the authors only consider a setting in which the frequencies of primitive events are known in advance and do not change. An optimization method based on postponing redundant operations was proposed by [31]. This work focuses on optimizing Reuse Consumption Policy queries by dividing evaluation into a shared part (pattern construction) and a per-instance part (result construction). The main goal of the authors is to improve the performance of Kleene closure patterns and solve the problem of imprecise timestamps. In comparison, our work focuses solely on sequence pattern matching.

The concept of lazy evaluation has also been proposed in the related research field of online processing of XML streams. [8] describes an XPath-based mechanism for filtering XML documents in stream environments. This mechanism postpones costly operations as long as possible. However, the goal in this setting is only to detect the presence or absence of a match, whereas our focus is on finding all possible matches between primitive events. In [20], a technique for lazy construction of a DFA (Deterministic Finite Automaton) on-the-fly is discussed. This work is motivated by the problem of exponential growth of automata for XPath pattern matching. Our work solves a different problem of minimizing the number of runtime NFA instances rather than the size of the automaton itself. In addition, while there is some overlap in the semantics of CEP and XPath queries, they were designed for different purposes and allow different types of patterns to be defined.

### 3. EAGER EVALUATION

In this section we present a subset of the SASE language for defining sequence patterns. SASE itself is thoroughly discussed in [3]. We formally describe the eager NFA matching mechanism, how a given sequence is compiled into an NFA, and how this NFA is used at runtime to detect the pattern. Here we also introduce the notations and terminology to be used in later sections.

#### 3.1 Specification Language

Most CEP systems enable users to define patterns using a declarative language. Common patterns supported by such

languages include sequences, conjunctions, disjunctions, and negation of events. As described in Section 3.2, patterns expressed in these languages will be compiled into a state machine for use by the detection mechanism.

The SASE language combines a simple, SQL-like syntax with a high degree of expressiveness, making it possible to define a wide variety of patterns. The semantics and expressive power of the language are precisely described in a formal model. In its most basic form, SASE event definition is composed of three building blocks: PATTERN, WHERE and WITHIN.

Each primitive event in SASE has an arrival timestamp, a type, and a set of attributes associated with the type. An attribute is a data item related to a given event type, represented by a name and a value. Attributes can be of various data types, including, but not limited to, numeric and categorical.

The PATTERN clause defines the pattern of simple events we would like to detect. Each event in this pattern is represented by a unique name and a type. The only information it provides is with regard to the types of participating events and the relations between them. In this work we limit the discussion to sequence patterns. A sequence is defined using the operator  $SEQ(A\ a, B\ b, \dots)$ , which provides an ordered list of event types and gives a name to each event in the sequence.

The WHERE clause specifies constraints on the values of data attributes of the primitive events participating in the pattern. These constraints may be combined using Boolean expressions. We assume, without loss of generality, that this clause is in the form of a CNF formula.

Finally, the WITHIN clause defines a time window over the entire pattern, specifying the maximal allowed time interval (in some predefined time units) between the arrival timestamps of the first primitive event and the last one. This time interval is denoted by  $W$ .

As an example, consider the pattern presented in Example 1. There is a single event type, which we will denote by  $E$ . This event type has two data attributes: a categorical attribute called “ticker,” which represents the stock for which the event has occurred, and a numerical attribute called “price,” which is the price of the stock. Assuming the stocks  $a$ ,  $b$ , and  $c$  are MSFT, GOOG and AAPL respectively, this pattern can be declared in SASE, as depicted in Figure 1.

```
PATTERN SEQ(E a, E b, E c)
WHERE (a.ticker = MSFT) AND (b.ticker=GOOG)
AND (c.ticker = AAPL) AND (a.price < b.price) AND
(b.price<c.price)
WITHIN 4 hours
```

Figure 1: SASE specification of a pattern from Example 1

#### 3.2 The Eager Evaluation Mechanism

In this subsection we formally describe the structure of the eager NFA and how it is used to detect patterns. Formally, an NFA is defined as follows:

$$A = (Q, E, q_1, F),$$

where:

- $Q$  is a set of states;
- $E$  is a set of directed edges, which can be of several types, as described below;
- $q_1$  is an initial state;
- $F$  is a final accepting state.

An edge is defined by the following tuple:

$$e = (q_s, q_d, action, name, condition),$$

where  $q_s$  is the source state of an edge,  $q_d$  is the destination state,  $action$  is always one of those described below,  $name$  may be any of the event names specified in the PATTERN block, and  $condition$  is a Boolean predicate that has to be satisfied by an incoming event in order for the transition to occur.

Evaluation starts at the initial state. Transitions between edges are triggered by event arrivals from the input stream. The runtime engine runs multiple instances of an NFA in parallel, one for each partial match detected up to that point. Each NFA instance is associated with a *match buffer*. As we proceed through an automaton towards the final state, we use the match buffer to store the primitive events constituting a partial match. It is always empty at  $q_1$ , and events are gradually added to it during the evaluation. This is done by executing an appropriate edge action.

The *action* associated with an edge is performed when the edge is traversed. It can be one of the following (the actions listed below are simplified versions of the ones defined for SASE [3]):

- *take* – consumes the event from the input stream and adds it to the match buffer.
- *ignore* – skips the event (consumes an event from an input stream and discards it instead of storing it in any kind of buffer).

A *condition* on an edge reflects the conditions in the WHERE part of the input pattern. It may reference the currently accepted event *name*, as well as events in the match buffer.

If during the traversal of an NFA instance the final state is reached, the content of the associated match buffer is returned as a successful match for the pattern. If during evaluation the time constraint specified in the WITHIN block is violated, the NFA instance and the match buffer are discarded.

Figure 2 illustrates the NFA compiled for the pattern in Figure 1. Note that the final state can only be reached by executing three *take* actions; hence, successful evaluation will produce a match buffer containing three primitive events comprising the detected match.

The match buffer should be thought of as a logical construct. As discussed by Agrawal et al. [3], there is no need to allocate dedicated memory for each match buffer, since multiple match buffers can be stored in a compact manner that takes into account that certain events may be included in many buffers.

Note that there may be several edges leading from the same state and specifying the same event type, whose conditions are not mutually exclusive (i.e., an event can satisfy several conditions). In this case, an event will cause more than one traversal from a given state. If an event triggered the traversal of  $n$  edges, the instance will be replicated  $n - 1$

times. On each of the resulting  $n$  instances a different edge will be traversed. As an example, consider the situation described in Figure 3. In 3a, there is some instance of an NFA from Figure 2 with an event  $m$  in its match buffer, currently in state  $q_2$  (we mark the current state of an instance with bold border). In 3b, an event  $g$ ,  $g.ticker = GOOG$  has arrived. This event triggers the traversal of two edges, namely the outgoing *take* edge and the outgoing *ignore* edge. As a result, one new instance will be created to allow both traversals to occur.

### 3.2.1 Eager Sequence NFA Structure

This section describes the structure and construction of an NFA that detects a sequence pattern of  $n$  primitive events.

A sequence pattern will be compiled into a chain of  $n + 1$  states, with each of the first  $n$  states corresponding to each primitive event in the sequence, followed by a final state  $F$ . Each state, except for the last one, has an edge leading to itself for every event name (referred to as *self-loops*) and an edge leading to the next state (referred to as *connecting edges*).

The self-loops for all event names have an *ignore* action. The edge leading from the  $k^{th}$  state to the next one has a *take* action with the event name of the  $k^{th}$  event in the sequence. The purpose of the self-loops is to allow detection of all possible combinations of events. This is achieved by exploiting non-deterministic behavior as illustrated by Figure 3.

To describe the conditions on the edges, we define an auxiliary predicate, known as the *timing predicate*, and denoted by  $p_t$ . Let  $t_{min}$  denote the timestamp of the earliest event in the match buffer, and  $now()$  denote the current time. If the match buffer is empty,  $t_{min}$  holds the current time. The timing predicate checks whether the match buffer still adheres to the timing constraint, i.e., all primitive events are located within the allowed time window  $W$ . More formally,  $p_t = (t_{min} > now() - W)$ . The condition on self-loops is  $p_t$ . The conditions in the WHERE part are translated to the conditions on the connecting edges as follows:

1. For each clause of the CNF, let  $i$  denote the index of the latest primitive event it contains (in the specified order of appearance in the pattern).
2. The condition on the edge connecting the  $i^{th}$  state with the following state is a conjunction of all the CNF clauses with the index  $i$  and the timing predicate.

For example, consider constructing a sequence NFA for the pattern in Figure 1. The edge from  $q_1$  to  $q_2$  will only contain a part of the global condition on  $a$ , the next edge will specify the constraint on  $b$  and the mutual constraint on  $a$  and  $b$ , and, finally, the final edge towards the accepting state will validate the constraint on  $c$  and the mutual constraint on  $c$  and  $b$ .

Figure 2 demonstrates the result of applying the construction process described above on the pattern in Figure 1.

### 3.2.2 Runtime Behavior

As described above, the pattern detection mechanism consists of multiple NFA instances running simultaneously, where each instance represents a partial match. Each NFA instance contains the current state and a match buffer. Upon startup, the system creates a single instance with an empty match

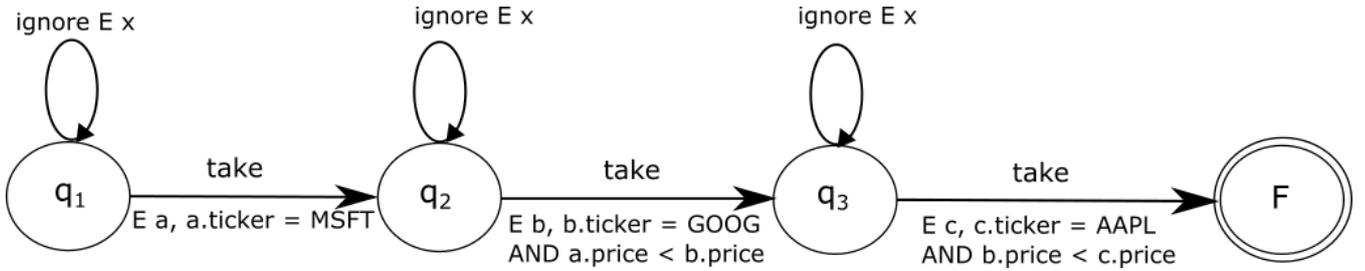


Figure 2: NFA for Example 1

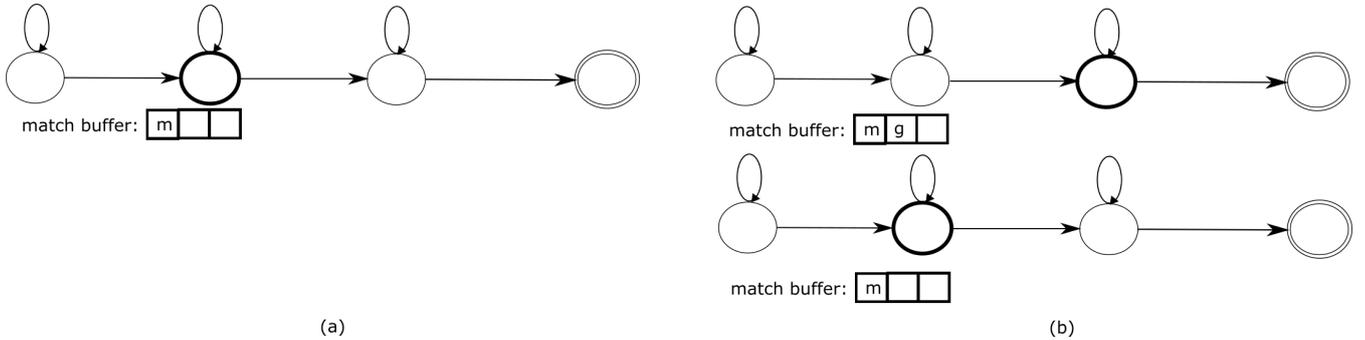


Figure 3: Non-deterministic evaluation of NFA for Example 1. (a) The sole NFA instance is currently at the second evaluation stage, with a single event in its match buffer. (b) A new event  $g$  arrives, and now the NFA instance can either (1) accept the new event as a part of the potential match and proceed to the next step, or (2) ignore it (by traversing a self loop) and keep waiting for a future event of the same name. The problem is solved by duplicating the instance and applying both moves.

buffer, whose current state is the initial state. Every event received on the input stream will be applied to all NFA instances. If the timing predicate is not satisfied on a given instance (i.e., the earliest event in the match buffer is not within the allowed time interval), the instance and the associated match buffer will be discarded. Otherwise, an event will either cause a single edge traversal on an unconditional *ignore* edge, or also an additional traversal on a *take* edge. In the former case the event will be ignored. In the latter case the instance will be duplicated, and both possible traversals will be executed on different copies.

#### 4. LAZY EVALUATION

In this section we present our main contribution, the lazy evaluation mechanism.

First, we will demonstrate the need for such a mechanism and show its effectiveness using the continuation of Example 1. Consider a scenario where on a certain day primitive events corresponding to  $a$  and  $b$  (MSFT and GOOG respectively) are very frequent, while events corresponding to  $c$  (AAPL) are relatively rare. More specifically, assume that within a time window  $t$  we receive 100 instances of MSFT stock events, denoted  $a_{p_1}^1, \dots, a_{p_{100}}^{100}$ , followed by 100 instances of GOOG stock events, denoted  $b_{p_{101}}^1, \dots, b_{p_{200}}^{100}$ , followed by a single instance of an AAPL stock event, denoted  $c_{p_{201}}^1$ . In addition, let us assume that there is only a single  $b_{p_i}^i$  event such that  $p_i < p_{201}$ . In such a case, an eager NFA will evaluate the condition  $a.price < b.price$  10,000 times, and the condition  $b.price < c.price$  for every pair of  $a$  and  $b$  that satisfied the first condition (up to 10,000 times). We may

substantially reduce the number of evaluations if we defer the match detection process until the single event for AAPL has arrived, then pair it with appropriate GOOG events, and finally check which of these pairs match a MSFT event. In this case we need to perform 100 checks of  $b.price < c.price$ , and an additional 100 checks of  $a.price < b.price$ , resulting in a total of 200 evaluations in comparison to at least 10,000 evaluations in the eager strategy. In addition, note that at every point in time, we hold a single partial match, as opposed to the eager mechanism, which may hold up to 10,000 partial matches.

The lazy evaluation model is able to take advantage of varying degrees of selectivity among the events in the sequence to significantly reduce the use of computational and memory resources. For the purpose of our discussion, *selectivity* of a given event name will be defined as an inverse of the frequency of arrival of events that can be matched to this name. We present the required modifications to the eager NFA model so that it can efficiently support lazy evaluation.

The idea behind lazy evaluation is to enable instances to store incoming events, and if necessary, retrieve them later for processing. To support this, an additional buffer, referred to as the *input buffer*, is associated with each NFA instance, and an additional action, referred to as *store*, is defined. When an edge with a *store* action is traversed, the event causing the traversal is inserted into the input buffer. The input buffer stores events in chronological order. Those events can then be accessed during later evaluation steps, using a modification on the *take* edge that we will define shortly.

An additional feature of lazy evaluation is that a sequence is constructed by adding events to partial matches in descending order of selectivity (rather than in the order specified in the sequence). From now on, we will refer to the order provided in the input query as the *sequence order*, and to the actual evaluation order as the *selectivity order*. As an example, consider the pattern from Example 1 again. Assume we wish to construct a lazy NFA that first matches  $b$ , then  $c$ , and finally  $a$ . In this case, our sequence order is  $a, b, c$  while our selectivity order is  $b, c, a$ .

Since events may be added to the match buffer in an order that is different from the sequence order, it is necessary to specify to which item in the sequence they match. To support this, the *take* action is modified to include an event name that will be associated with the event it inserts into the match buffer (the names are taken from the definition in the PATTERN block). The notation  $take(a)$  denotes that the name  $a$  will be associated with events inserted by this *take* action. In the above example, to construct a lazy NFA using the selectivity order  $b, c, a$ , we will assign  $take(b)$  edge to its first state,  $take(c)$  to its second state and  $take(a)$  to the third and final state.

Finally, the model must include a mechanism for efficient access to events in the input buffer. For that purpose, we change the semantics of the *take* action. Whereas in the eager NFA model an event accepted by this type of edge is always taken from the input stream, in the lazy NFA model we extend this functionality to also trigger a search inside the input buffer, which returns events to be examined for a current match. If the result of this search, combined with events appearing in the input stream, contains more than a single event with the required name, the sequence will be evaluated non-deterministically by spawning additional NFA instances.

Note that invoking a full scan of the entire input buffer on each *take* action of each NFA instance would be inefficient and redundant. It is not required since, in general, only a certain range of events in the input buffer are relevant to a given *take* edge. Searching for a potentially matching event in any other interval is unnecessary and will not result in a match.

We will demonstrate the above observation using the following example. Consider again the pattern from Example 1. We will show the necessity of limiting the search interval on two different selectivity orders:  $a, b, c$  and  $c, a, b$ .

1. Evaluate the sequence  $a, b, c$  using selectivity order  $a, b, c$ . For the first outgoing edge detecting  $a$ , no constraints can be defined and the event can be taken either from the input buffer or the input stream. Note however that, since at this stage the input buffer will contain no  $a$  instances, in fact only the input stream should be considered. At the next state and the next outgoing edge detecting  $b$ , we are only interested in events following the particular instance of  $a$  (which was detected at the previous state and is now located in the match buffer). By definition of the input buffer, however, at this stage it can only contain  $b$  events that arrived before  $a$ . Hence, there is no need to scan the input buffer, but only to wait for the arrival of  $b$  from the input stream. The same holds for  $c$ , which is detected at the *take* edge from the third to the final state.

2. Evaluate the sequence  $a, b, c$  using selectivity order  $c, a, b$ . For the first outgoing edge detecting  $c$ , no limitations can be formulated. It will only take events from the input stream, since the input buffer is empty. For the second outgoing *take* edge detecting  $a$ , we are limited to events preceding the already accepted  $c$  instance. Consequently, any  $a$  event arriving on the input stream will be irrelevant due to sequence order constraints. As for the input buffer, only the events that arrived before  $c$  are to be considered. Finally, examine the third edge detecting  $b$ . Since we are searching for an event which is required to precede an already arrived  $c$ , any possible match can only be found in the input buffer and not in the input stream. Moreover, since the pattern requires  $a$  to precede  $b$ , not all  $b$  events located in the input buffer are to be returned and evaluated, but only those succeeding the accepted  $a$  instance and preceding the accepted  $c$  instance located in the match buffer.

Figure 4 illustrates the two examples above.

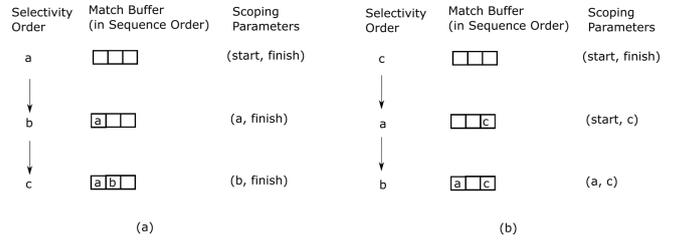


Figure 4: Scoping Parameters Example

Since the relevant range of events is always known in advance, the redundant operations can be avoided by providing a way to specify it for any such edge. To this end, we modify the definition of a take edge to include a pair of scoping parameters. Scoping parameters specify the exact behavior of an edge, defining the beginning and the end of the relevant scope respectively. For the purpose of this discussion, scope is defined as a time interval (possibly open and including future time) in which the event expected by a given edge is required to arrive. The scoping parameters specify whether the source of events considered by this edge should be the input buffer or the input stream. If the data should be received from the input buffer, the scoping parameters also indicate what part of the input buffer is applicable.

More formally, the scoping parameters of an edge  $e$  are denoted by  $e(s, f)$ , where  $s$  is the start of the scope and  $f$  is the end of the scope. The values of both parameters can be either event names or special keywords *start* or *finish*. When the value of some scoping parameter is an event name, an event with an appropriate name is examined in the match buffer, and its timestamp is used for deriving the actual scope as described below.

The parameter  $s$  can accept one of the following values:

- The reserved keyword *start*: in this case, events are taken from the beginning of the input buffer. This scoping parameter is applicable if no event preceding the event taken by this edge according to sequence order has already been handled by the NFA.
- A name of a primitive event: in this case, only events matched to names succeeding the corresponding event

from the match buffer in the *sequence order* are read from the input buffer.

The parameter  $f$  can accept one of the following values:

- A name of a primitive event: in this case, only events matched to names preceding the corresponding event from the match buffer in the *sequence order* are read from the input buffer,
- The reserved keyword *finish*: in this case, events are also received from the input stream.

We will demonstrate the definitions above on examples from the beginning of the section, illustrated also in Figure 4.

1. Evaluation of the sequence  $a, b, c$  using selectivity order  $a, b, c$ . For the first edge detecting  $a$ , the scoping parameters will be  $e_1$  (*start, finish*). For the next edge detecting  $b$ , the scoping parameters will be  $e_2$  ( $a, finish$ ). Finally, for the following edge detecting  $c$ , the scoping parameters will be  $e_3$  ( $b, finish$ ).
2. Evaluation of the sequence  $a, b, c$  using selectivity order  $c, a, b$ . For the first edge detecting  $c$ , the scoping parameters will be  $e_1$  (*start, finish*). For the next edge detecting  $a$ , the scoping parameters will be  $e_2$  (*start, c*). Finally, for the following edge detecting  $b$ , the scoping parameters will be  $e_3$  ( $a, c$ ).

To summarize, a combination of  $s$  and  $f$  defines the time interval for valid events for the given *take* edge, based on timestamps of events. This interval can also be unlimited from each of its sides. If unlimited from the left, all events in the input buffer are considered until the right delimiter. If unlimited from the right, all events in the input buffer are considered, starting from the left delimiter, and events from input stream (i.e., arriving as a *take* operation takes place) are considered as well.

The following sections will explain how scoping parameters are calculated for different types of lazy NFA.

## 5. CHAIN NFA

In this section we will formally define the first of two new NFA types, the chain NFA.

The chain NFA utilizes the constructs of the lazy evaluation model, evaluating events according to a selectivity order given in advance. It consists of  $n + 1$  states, arranged in a chain. Each of the first  $n$  states is responsible for detecting one primitive event in the pattern, and the last one is the accepting state. The states are sorted according to the given selectivity order, which we will denote by  $sel$ .

We will also denote by  $e_i$  the  $i^{th}$  event in  $sel$  and by  $q_i$  the corresponding state in the chain. The state  $q_i$  will have an outgoing edge  $take(e_i)$ , a *store* edge for all events which are yet to be processed (succeeding  $e_i$  in  $sel$ ), and an *ignore* edge for all already processed events (preceding  $e_i$  in  $sel$ ).

More formally, let  $E_i$  denote the set of outgoing edges of  $q_i$ . Let  $Prec_{ord}(e)$  denote all events preceding an event  $e$  in an order  $ord$ . Similarly, let  $Succ_{ord}(e)$  denote all events succeeding  $e$  in  $ord$ . Then,  $E_i$  will contain the following edges:

- $e_i^{ignore} = (q_i, q_i, ignore, Prec_{sel}(e_i), true)$ : any event whose name corresponds to one of the already taken events is ignored.

- $e_i^{store} = (q_i, q_i, store, Succ_{sel}(e_i), true)$ : any event that might be taken in one of the following states is stored in the input buffer.

- $e_i^{take} = (q_i, q_{i+1}, take, e_i, cond_i \wedge InScope_i)$ : an event with the name  $e_i$  is taken only if it satisfies the conditions required by the initial pattern (denoted by  $cond_i$ ) and is located inside the scope defined for this edge (denoted by a predicate  $InScope_i$ ).

The chain NFA will thus be defined as follows:

$$A = (Q, E, q_1, F),$$

where:

$$Q = \{q_i | 1 \leq i \leq n\} \cup \{F\}$$

$$E = \bigcup_{i=1}^n E_i$$

Figure 5 demonstrates the chain NFA for the pattern shown in Figure 1. For simplicity, *ignore* edges are omitted, as are  $InScope_i$  predicates.

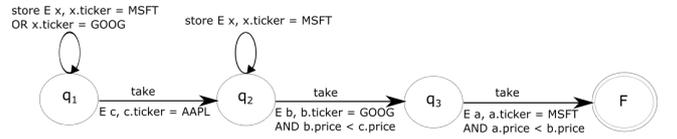


Figure 5: Chain NFA for Example 1

We will now define how scoping parameters for *take* edges of the chain NFA are calculated. Given a set  $E$  of events, let  $Latest_{ord}(E)$  be the latest event in  $E$  according to  $ord$ , and, correspondingly, let  $Earliest_{ord}(E)$  be the earliest event in  $E$  according to  $ord$ . Finally, let  $seq$  denote the original sequence order as specified by the input pattern.

The scoping parameters for a take edge  $e_i^{take}$  accepting a primitive event  $e_i$  will be defined as follows:

$$s(e_i^{take}) = \begin{cases} Latest_{sel}(Prec_{sel}(e_i) \cap Prec_{seq}(e_i)) & \text{if } Prec_{sel}(e_i) \cap Prec_{seq}(e_i) \neq \emptyset \\ start & \text{otherwise} \end{cases}$$

$$f(e_i^{take}) = \begin{cases} Earliest_{sel}(Prec_{sel}(e_i) \cap Succ_{seq}(e_i)) & \text{if } Prec_{sel}(e_i) \cap Succ_{seq}(e_i) \neq \emptyset \\ finish & \text{otherwise} \end{cases}$$

A formal proof of the equivalence of the eager NFA and the chain NFA was omitted due to space considerations. The correctness of this claim implies that any eager sequence NFA can be modified into a chain NFA using any selectivity order without affecting the language it accepts.

## 6. TREE NFA

Chain NFA described in the previous section may significantly improve evaluation performance, provided we know the correct order of selectivity. As shown in the examples above, the more drastic the difference between the arrival rates of different events, the greater the potential improvement.

There are, however, several drawbacks which severely limit the applicability of chain NFA in real-life scenarios. First, the assumption of specifying the selectivity order in advance

is not always realistic. In many cases, it is hard or even impossible to predict the actual selectivity of primitive events. Note that the described model is very sensitive to wrong guesses, as specifying a low-selectivity event before a high-selectivity event will yield many redundant evaluations and overall poor performance. Second, even if it is possible to set up the system with a correct selectivity order, we can rarely guarantee that it will remain the same during the run. In many real-life applications the data is highly dynamic, and arrival rates of different events are subject to change on-the-fly. Such diversity may cause an initially efficient chain NFA to start performing poorly at some point. Continual changes may come, for example, in the form of bursts of usually rare events.

To overcome these problems we introduce the notion of ad hoc selectivity. Instead of relying on a single selectivity order specified at the beginning of the run, we determine the current selectivity on-the-fly and modify the actual evaluation chain according to the order reflecting the current frequencies of the events. Our NFA will thus have a tree structure, with each of its nodes (states) “routing” the incoming events to the next “hop” according to this dynamically changing order. By performing these “routing decisions” at each evaluation step, we guarantee that any partial match will be evaluated using the most efficient order possible at the moment.

To implement the desired functionality, we require that each state have knowledge regarding the current selectivity of each event name. We will use the input buffer introduced above to this end. By its definition, the input buffer of a particular NFA instance contains all events that arrived from the input stream within the specified time window. For each event name, we will introduce a counter containing the current number of events matched with this event name inside the buffer. This counter will be incremented on each insertion of a new event with the corresponding name and decremented upon its removal.

Matching the pattern requires at least one event corresponding to each event name to be present in the input buffer. Hence, we will add a condition stating that no evaluation will be made by a given instance until all the counters are greater than zero. Only when all of the event counters are greater than zero does it make sense to determine the evaluation order, since otherwise the missing event(s) may not arrive at all and the partial matching process will be redundant. After the above condition is satisfied, we can derive the exact selectivity order based on the currently available data by sorting the counters.

The above calculation will be performed by each state on each matching attempt, and the resulting value will be used to determine the next step in the evaluation order. In terms of NFA, this means that a state needs to select the next state for a partial match based on the current contents of the input buffer. To this end, a state has several outgoing *take* edges as opposed to a single one in chain NFA. Each edge takes a different event name and the edges point to different states. We will call the NFA employing this structure a *tree NFA* and will formally define this model below.

Figure 6 illustrates a tree NFA for the pattern in Figure 1. For simplicity, ignore edges are omitted.

In formal terms, a tree NFA is structured as a tree of depth  $n-1$ , the root being the initial state and the leaves connected to the accepting state. Nodes located at each layer  $k$ ;  $0 \leq$

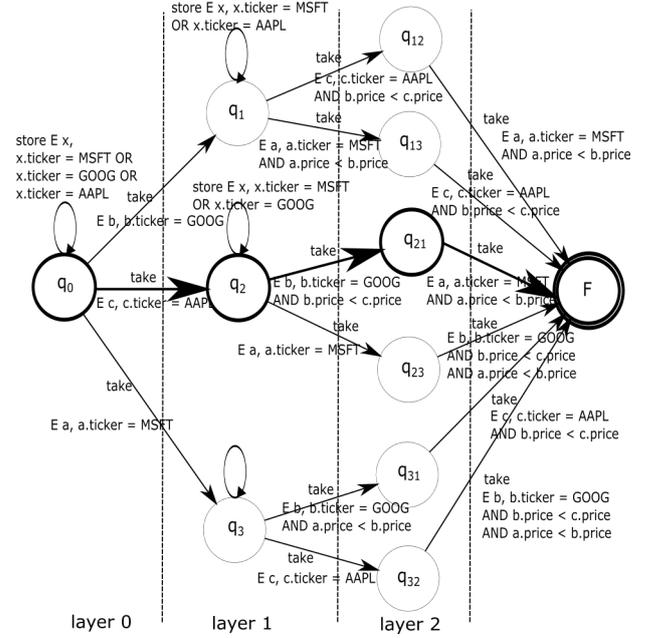


Figure 6: Tree NFA for Example 1

$k \leq n-1$  (i.e., all nodes in depth  $k$ ) are all states responsible for all orderings of  $k$  event names out of the  $n$  event names defined in the sequence. Each such node has  $n-k$  outgoing edges, one for each event name which does not yet appear in the partial ordering this node is responsible for. Those edges are connected to states at the next layer, responsible for all extensions of the ordering of this particular node to length of  $k+1$ . The only exceptions to this rule are the leaves, which have a single outgoing edge, connected directly to the final state.

For instance, in the example in Figure 6, layer 0 contains the initial state  $q_0$ , layer 1 contains states  $q_1, q_2, q_3$ , and layer 2 contains the states  $q_{12}, q_{13}, q_{21}, q_{23}, q_{31}, q_{32}$ .

More formally, the states for a tree NFA are defined as follows. Let  $O_k$  denote the ordered subsets of size  $k$  of the event names  $e_1, \dots, e_n$ . Let

$$Q_k = \{q_{ord} | ord \in O_k\}$$

denote the set of states at the layer  $k$  (note that  $Q_0 = \{q_0\}$ ). Then the set of all states of the tree NFA is

$$Q = \bigcup_{k=0}^{n-1} Q_k \cup \{F\}$$

$$q_0 = q_0.$$

To describe the edges and their respective conditions, some preliminary definitions are needed.

First, we will complete the definitions required for the scoping parameters. Since each state  $q_{ord}$  corresponds to some evaluation order prefix  $ord$ , we will set  $ord_e = ord$  for each outgoing edge  $e$  of  $q_{ord}$ . As mentioned earlier, it is enough for  $ord_e$  to be a partial order ending with  $\hat{e}$ . In other words, each *take* edge in the tree derives the corresponding scope for its target event name from the order used for reaching this edge.

Similarly to the chain NFA, the predicate  $InScope_{ord}(e)$  will denote that an event  $e$  is located within the corresponding scope  $(s(q_{ord}, e), f(q_{ord}, e))$ .

Let  $c_e$  denote the value of the counter of events associated with the name  $e$  in the input buffer. Let  $se(q_{ord}) = \min(\{c_e | e \notin ord\})$  denote the most selective (i.e., most infrequent) event in the input buffer during the evaluation step in which  $q_{ord}$  is the current state. Finally, we will define the predicate  $p_{ne}(q_{ord})$  (non-empty) as the condition on the input buffer of state  $q_{ord}$  to contain at least a single instance of each primitive event not appearing in  $ord$  and another predicate  $p_{se}(q_{ord}, e)$  to be true if and only if an event  $e$  corresponds to event type  $se(q_{ord})$ . Let  $E_{ord}$  denote the set of outgoing edges of  $q_{ord}$ . Then,  $E_{ord}$  will contain the following edges:

- $e_{ord}^{ignore} = (q_{ord}, q_{ord}, ignore, ord, true)$ : any event whose name corresponds to one of the already taken events (appearing in the ordering this state corresponds to) is ignored.
- For each primitive event  $e \notin ord$ :
  - $e_{ord,e}^{store} = (q_{ord}, q_{ord}, store, e, \neg p_{ne}(q_{ord}) \vee \neg p_{se}(q_{ord}, e))$ : when either the  $p_{ne}$  or  $p_{se}$  condition is not satisfied, the incoming event is stored into the input buffer.
  - $e_{ord,e}^{take} = (q_{ord}, q_{ord}, take, e, p_{ne}(q_{ord}) \wedge p_{se}(q_{ord}, e) \wedge cond_e \wedge InScope_{ord}(e))$ : if the contents of the input buffer satisfy the  $p_{ne}$  and  $p_{se}$  predicates and an incoming event with a name  $e$  (1) satisfies the conditions required by the initial pattern (denoted by  $cond_e$ ); and (2) is located within the scope defined for this state, it is taken into the match buffer and the NFA instance advances to the next layer of the tree.
- For states in the last layer (where  $|ord| = n$ ), the *take* edges are of the form  $e_{ord,e}^{store} = (q_{ord}, F, take, e, p_{ne}(q_{ord}) \wedge cond_e \wedge InScope_{ord}(e))$ .

The set of all edges for tree NFA is defined as follows:

$$E = \bigcup_{\{ord | q_{ord} \in Q\}} E_i,$$

and the NFA itself is defined as follows:

$$A = (Q, E, q_1, F),$$

where  $Q$  and  $E$  are as defined above.

It can be observed that a tree NFA contains all the possible chain NFAs for a given sequence pattern, with shared states for common prefixes. Thus, the execution of a tree NFA on any input is equivalent to the execution of some chain NFA on that input. The conditions on tree NFA edges are designed in such a way that the most selective event is chosen at each evaluation step. Hence, this chain NFA is always the one whose given selectivity order is the actual selectivity order as observed from the input stream. An example can be seen in Figure 6. Nodes and edges marked in bold illustrate the evaluation path for an input stream satisfying  $count(AAPL) \leq count(GOOG) \leq count(MSFT)$ , i.e., corresponding to the selectivity order  $c, b, a$ .

The scoping parameters for a tree NFA are calculated the same way as for a chain NFA, as described in Section 5.<sup>2</sup>

<sup>2</sup>Contrary to the chain NFA, the tree NFA does not have a predefined selectivity order  $sel$  to be used for calculating

## 6.1 Implementation Issues

When implementing the tree NFA, the number of states might be exponential in  $n$ . To overcome this limitation, we propose to implement lazy instantiation of NFA states – only those states reached by at least a single active instance will be instantiated and will actually occupy memory space. After all NFA instances reaching a particular state are terminated, the state will be removed from the NFA as well. Even though the worst case complexity remains exponential in this case, in practice there will be fewer changes in the event rates than there will be new instances created. This conclusion is supported by our experiments, which are explained in the following section.

## 7. EXPERIMENTAL EVALUATION

We evaluated the performance of chain and tree NFA in comparison to the eager model. Our metrics for this comparison and analysis of both evaluation mechanisms are the runtime complexity and the memory consumption.

As a measure of runtime complexity, we counted how many times a condition on an edge is evaluated. For instance, consider the pattern from Example 1 and two successive streams of events:  $a_{p=3}^1, b_{p=7}^1, c_{p=9}^1$  and  $a_{p=3}^1, b_{p=13}^2, c_{p=9}^1$ . The evaluation of the first stream will cost us exactly three operations (validation of conditions on edges  $q_1 \rightarrow q_2$ ,  $q_2 \rightarrow q_3$  and  $q_3 \rightarrow F$ ), while the second stream will cost only two ( $q_1 \rightarrow q_2$  and  $q_2 \rightarrow q_3$ ), since the condition on  $q_2 \rightarrow q_3$  is not satisfied and the evaluation stops at that point.

We measured memory consumption by two metrics, corresponding to the two kinds of data stored by the NFA during runtime. The first metric was the peak number of simultaneously active NFA instances, and the second was the peak number of buffered events waiting to be processed. Note that those metrics are not completely independent, as an NFA instance also includes a match buffer and an input buffer containing stored events.

All NFA models under examination (eager, chain and tree) were implemented in Java and integrated into the FINCoS framework [25]. FINCoS, developed at the University of Coimbra, is a set of benchmarking tools for evaluating the performance of CEP systems.

All experiments were run on a HP 2.53 Ghz CPU and 8.0 GB RAM. We used the real-world historical data of stock prices from the NASDAQ stock market, taken from [1]. This data spans a 5-year period, covering over 2100 stock identifiers with prices updated on a per minute basis. Each primitive event is of type 'Stock' and has the following attributes: stock identifier (ticker), timestamp, and current price. We also assumed that each event has an attribute specifying to which sector the stock belongs, e.g., hi-tech, finance or pharmaceuticals.

In order to support efficient detection of the pattern described below, preprocessing was applied to this preliminary data. For each event,  $h-1$  chronologically ordered previous prices of the respective stock were added as new attributes, constructing a history of  $h$  successive stock prices.

the scoping parameters. Instead, for an edge  $e_{ord,e}^{take}$  we will substitute  $sel$  with the partial order  $ord$ . This order is the effective selectivity order applied on the current input.

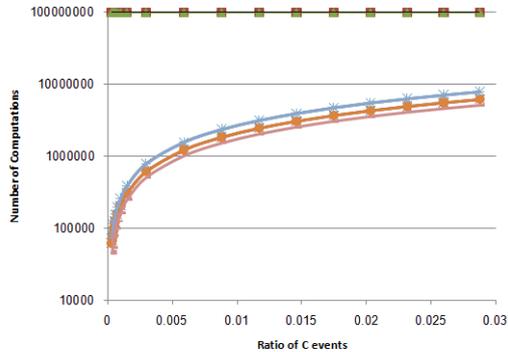
During all measurements, the detection pattern for the system was specified as follows: a sequence of three stock identifiers was requested, with each stock belonging to some predefined category. In addition, we required consecutive stocks in the sequence to be highly correlated (i.e., the Pearson correlation coefficient between stocks price histories was above some predefined threshold). The correlation was calculated for each pair of events based on a history list each event carries, built as described above. The final stock in a sequence was required to be a Google stock, the first stock belonged to the hi-tech sector, and the second stock belonged to the finance sector. The time window for event detection was set to the length of the price history.

Using the previously described SASE language, the aforementioned pattern can be declared in the following way:

```
PATTERN SEQ(Stock a, Stock b, Stock c)
WHERE (a.ticker∈Finance) AND (b.ticker∈Hi-Tech)
AND (c.ticker = GOOG) AND (Corr(a.history,b.history)>T)
AND (Corr(b.history,c.history)>T)
WITHIN h
```

In the described pattern, events  $a$  and  $b$  share approximately equal frequencies, which also fluctuated slightly over time, making each of the event types slightly more dominant part of the time. Event  $c$ , on the other hand, is significantly less frequent. One parameter of interest that affects the overall efficiency of the presented evaluation models is the relative frequency of  $c$  with respect to  $a$  and  $b$ , which we will denote as  $f_c$ . The lower the value of  $f_c$ , the larger the expected performance gain of our proposed lazy evaluation mechanisms. The value of  $f_c$  is controlled by modifying the input stream, either duplicating or filtering out  $c$  events.

In our first experiment, we compared the runtime complexity and memory consumption of the eager sequence NFA, all the possible chain NFAs, and the tree NFA.



**Figure 7: Comparison of NFAs by number of operations (logarithmic scale) for sequence  $a, b, c$**

Figure 7 describes the number of computations performed by each NFA as a function of  $f_c$ . The following observations can be made:

1. Eager NFA shows the same, very poor performance for any value of  $f_c$ .
2. Lazy chain NFAs constructed with  $c$  as the second or the third event (namely  $abc$ ,  $bac$ ,  $acb$  and  $bca$ ) display equally suboptimal performance because detecting the

pattern using these orders implies creation and manipulation of large numbers of NFA instances, just as with eager NFA.

3. Lazy chain NFAs constructed with  $c$  as a first event, namely  $cba$  and  $cab$ , perform one to three orders of magnitude better. This exactly matches our expectations, as starting the evaluation process only when the rarest event arrives allows us to significantly reduce the number of instances, and hence the number of calculations.
4. Tree NFA demonstrates slightly better performance than that of the best chain NFA ( $cab$  in our case). This minor improvement is due to the changes in the relative frequencies of  $a$  and  $b$  events, to which tree NFA was able to adapt as a result of its dynamic structure.

As the ratio of  $c$  events to all events grows and approaches 1, all the graphs are expected to eventually converge to the upper value. This is because, when all events in a pattern share the same frequency, no selectivity order is optimal (or, interchangeably, all orders are equally optimal), and thus changing the evaluation order will not improve performance.

In our next experiment we evaluated patterns with the most selective event  $c$  placed at the beginning or in the middle, producing the target sequences  $c, b, a$  and  $a, c, b$ . We used the same set of conditions as in the previous experiment. The results of the performance evaluation of the system when invoked on those patterns are shown in Figure 8. The main observation is that the performance of any lazy NFA is independent of the sequence order, as selectivity orders ending with  $c$  will always perform poorly, whereas those starting with  $c$  will show better results. The only notable difference is the performance of eager NFA, which significantly improves on the  $c, b, a$  pattern. The reason is that in this case the sequence order is also the most efficient selectivity order. It can be seen that, for any pattern, the tree NFA remains superior.

Now we proceed to the memory consumption comparison. As mentioned above, there are two different kinds of data stored by the NFA: instances and incoming primitive events. As presented in our theoretical analysis results, eager NFA tends to keep significantly larger numbers of instances in memory simultaneously than does lazy NFA. As for primitive events, lazy NFA stores them in the input buffer, while eager NFA keeps most of them inside the match buffers of the pending instances. Hence, memory requirements for buffering of events are virtually identical for all NFA types. This theoretical observation was also supported by our experiments. Therefore, in order to compare memory consumption, only the peak number of instances held simultaneously in memory should be considered.

Figure 9 demonstrates the peak number of instances generated by the different types of NFAs discussed above when detecting the sequence  $a, b, c$  on inputs of various sizes. Only some of the chain NFA graphs are shown. Other automata produced outputs very similar to one of the displayed ones and were omitted for the sake of clarity. It can be observed that:

1. Lazy chain NFAs with  $c$  as a first event require memory for a smaller number of instances than the other NFAs. This is because evaluation in these automata occurs only upon arrival of a  $c$  event, at which point

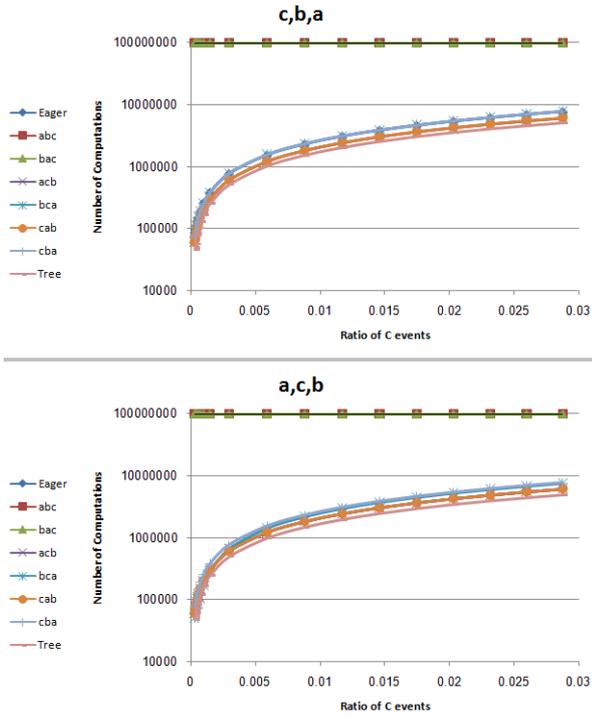


Figure 8: Comparison of NFAs by number of operations (logarithmic scale) for sequences  $c,b,a$  and  $a,c,b$

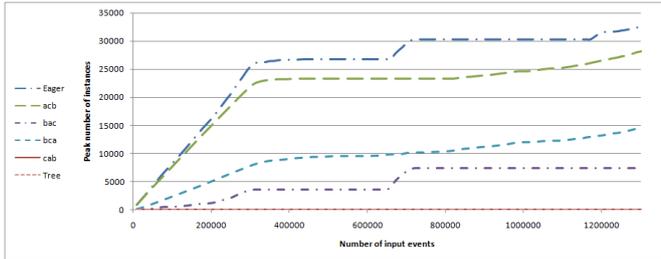


Figure 9: Comparison of NFAs by memory consumption for sequence  $a,b,c$

the whole match is already located in the input buffer. Hence, there is no need to wait for additional input from the stream and evaluation ends almost immediately in most cases.

2. Lazy chain NFAs corresponding to selectivity orders starting with  $a$  consume significantly more memory, which is comparable to the memory consumed by the eager NFA. As the previous graph shows, NFAs based on those orders use many instances simultaneously. The number of such instances is proportional to that of eager NFA; hence, they use approximately equivalent memory in terms of NFA instances.
3. Lazy chain NFAs corresponding to selectivity orders starting with  $b$  display better, yet still do not achieve optimal memory utilization due to selectivity of mutual conditions between  $a$  and  $b$ .

4. Memory consumption of the tree NFA is comparable to that of the most efficient chain NFA, also in keeping with our theoretical analysis.

In our last experiment we compared the performance of the NFAs discussed above on data with dynamically changing frequencies of all primitive events. For this experiment alone, synthetic data was used, generated using the FIN-CoS framework [25]. An artificial stream was produced in which the rarest event was switched after each 100,000 incoming events. Then, all NFAs were tested against this input stream, while after each 10,000 incoming events the number of computations was measured.

Figure 10 demonstrates the results. As in the previous graph, some of the chain NFAs were omitted due to very similar results. The x-axis represents the number of events from the beginning of the stream. It can be thought of as the closest estimate to the time axis. The y-axis represents the number of computations per 10,000 events.

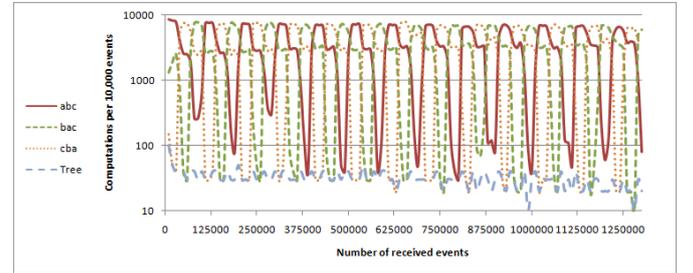


Figure 10: Comparison of NFAs by number of operations on highly dynamic input (logarithmic scale) for sequence  $a,b,c$

This figure illustrates the superiority of the tree NFA over its competitors and its high adaptivity to changes in event selectivity. At any single point there is one selectivity order that is the most efficient given the current event frequencies. The performance gain of the chain NFA based on that order over the other chain NFAs reaches up to two orders of magnitude. However, as soon as the event frequencies change, this NFA loses its advantage. On the other hand, the tree NFA shows consistent improvement over all chain NFAs regardless of the input selectivity.

## 8. CONCLUSIONS

This paper presented a lazy evaluation mechanism for efficient detection of complex sequence patterns. Unlike previous solutions, our system does not process the events in order of their arrival, but rather according to their descending order of selectivity. Two NFA topologies were proposed to implement the above concept. The chain NFA requires the selectivity order of the events in the sequence to be known in advance. The tree NFA utilizes an adaptive approach by computing the actual selectivity order on-the-fly. Our experimental results showed that both chain NFA and tree NFA achieve significant improvement over the eager evaluation mechanism in terms of performance and memory consumption.

## 9. ACKNOWLEDGMENTS

The research leading to these results has received partial funding from the [European Union's] Seventh Framework Programme [FP7-ICT-2013-11] under grant agreements n. [619491] and n. [619435]

## 10. REFERENCES

- [1] <http://www.eoddata.com>.
- [2] A. Adi and O. Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, 2004.
- [3] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 147–160.
- [4] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, 2008.
- [5] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.
- [6] A. Artikis, C. Baber, P. Bizarro, C. Canudas de Wit, O. Etzion, F. Fournier, P. Goulart, A. Howes, J. Lygeros, G. Paliouras, A. Schuster, and I. Sharfman. Scalable proactive event-driven decision making. *IEEE Technol. Soc. Mag.*, 33(3):35–41, 2014.
- [7] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: A high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 1100–1102. ACM.
- [8] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of xml documents with xpath expressions. *VLDB J.*, 11(4):354–379, 2002.
- [9] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
- [10] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaraqc: A scalable continuous query system for internet databases. *SIGMOD Rec.*, 29(2):379–390, 2000.
- [11] G. Cugola and A. Margara. Tesla: a formally defined event specification language. In *DEBS*, pages 50–61. ACM, 2010.
- [12] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, 2012.
- [13] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proceedings of the 10th International Conference on Advances in Database Technology*, pages 627–644. Springer-Verlag.
- [14] A. Demers, J. Gehrke, and B. P. Cayuga. A general purpose event monitoring system. In *In CIDR*, pages 412–422, 2007.
- [15] C. Dousson and P. L. Maigat. Chronicle recognition improvement using temporal focusing and hierarchization. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 324–329, 2007.
- [16] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, USA, 2010.
- [17] B. Gedik, H. Andrade, K. L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *SIGMOD Conference*, pages 1123–1134. ACM, 2008.
- [18] N. Giatrakos, A. Deligiannakis, M. Garofalakis, I. Sharfman, and A. Schuster. Distributed geometric query monitoring using prediction models. *ACM Trans. Database Syst.*, 39(2):16, 2014.
- [19] B. Gilburd, A. Schuster, and R. Wolff. Privacy-preserving data mining on data grids in the presence of malicious participants. In *13th International Symposium on High-Performance Distributed Computing*, pages 225–234, 2004.
- [20] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.
- [21] T. S. Group. Stream: The stanford stream data manager. Technical Report 2003-21, Stanford InfoLab, 2003.
- [22] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman. On supporting kleene closure over event streams. In *ICDE*, pages 1391–1393. IEEE, 2008.
- [23] A. Lazerson, I. Sharfman, D. Keren, A. Schuster, M. Garofalakis, and V. Samoladas. Monitoring distributed streams using convex decompositions. *Proc. VLDB Endow.*, 8(5):545–556, 2015.
- [24] Y. Mei and S. Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the 29th ACM SIGMOD Conference*, pages 193–206. ACM, 2009.
- [25] M. R. Mendes, P. Bizarro, and P. Marques. Fincos: Benchmark tools for event processing systems.
- [26] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.*, 29(2):282–318, 2004.
- [27] N. P. Schultz-Møller, M. M., and P. R. Pietzuch. Distributed complex event processing with query rewriting. In *DEBS*, 2009.
- [28] I. Sharfman, A. Schuster, and D. Keren. A geometric approach to monitoring threshold functions over distributed data streams. *ACM Trans. Database Syst.*, 32(4), 2007.
- [29] F. Wang and P. Liu. Temporal management of rfid data. In *Proceedings of the 31st International Conference on Very Large Data Bases, VLDB '05*, pages 1128–1139. VLDB Endowment, 2005.
- [30] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD Conference*, pages 407–418. ACM, 2006.
- [31] H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD*, pages 217–228, 2014.