Scalable Data Analytics Scalable Algorithms, Software Frameworks and Visualisation ICT-2013.4.2a

Project **FP7-619435 / SPEEDD**

Deliverable **D6.2**

Distribution **Public**

http://speedd-project.eu/

Computation and Communication Scalable Algorithms I

Daniel Keren, Arnon Lazerson, Mickey Gabel, Ilya Kolchinsky, Tsachi Sharfman, Assaf Schuster

Status: Final (Version 1.0)

June 2015

Project

Project ref.no.	FP7-619435
Project acronym	SPEEDD
Project full title	Scalable ProactivE Event-Driven Decision making
Porject site	http://speedd-project.eu/
Project start	February 2014
Project duration	3 years
EC Project Officer	Aleksandra Wesolowska

Deliverable

Deliverabe type	report
Distribution level	Public
Deliverable Number	D6.2
Deliverable title	Computation and Communication Scalable Algorithms I
Contractual date of delivery	M11 (December 2014)
Actual date of delivery	June 2015
Relevant Task(s)	WP6Tasks T1 & T2
Partner Responsible	TI
Other contributors	
Number of pages	38
Author(s)	Daniel Keren, Arnon Lazerson, Mickey Gabel, Ilya Kolchinsky, Tsachi Sharfman, Assaf Schuster
Internal Reviewers	
Status & version	Final
Keywords	Scalable Algorithms, Computational Overhead, Communication Minimization, Complex Event Processing, Monitoring Data and Event Streams, Models of Machine Learning

Contents

Executive Summary

The deliverable presents algorithmic contributions by means of scalable algorithms for complex event processing and for model tracking.

As part of the project goals for processing and manipulating rapid event streams, workpackage 6 deals with the development of novel scalable algorithmic approaches. The target is to present new ideas which will be able to process complex events with less overhead (by means of cpu time, memory requirements, I/O, etc.), and with less communication. Less overhead will provide vertical scalability, whereas reduced communication between distributed nodes will enable horizontal scalability.

We present in this deliverable two new algorithmic paradigms. The first of these paradigms may be able to reduce the amount of computation and the amount of local resources required for complex event processing. The second will enable distributed monitoring of sophisticated analytical and machine-learning models while using very low communication between participating nodes.

The proposed approaches will enable scalable computation and scalable implementation of the project's eventual integrated platform. The platform will then be able to handle many more events and using much less resources.

Initial findings show high potential for the proposed approaches. Overheads in preliminary experiments are reduced dramatically. The potential gains will be further studied in our explorations and experimentation. We will evaluate (and draw bounds on) the precise expected gain. We will experiment with data and in particular with shared evaluation data and with the project's Fraud use case.

1

Introduction

1.1 History of the Document

Version	Date	Author	Change Description
0.1	1/11/2014	Assaf Schuster (TI)	Set up of the document
0.3	2/11/2014	Ilya Kolchinsky (TI)	preliminary version of CEP paradigm
0.6	27/11/2014	Mickey Gabel (TI)	preliminary version of distributed model monitoring
0.9	30/11/2014	Assaf Schuster (TI)	Content integration
1.0	21/12/2014	Assaf Schuster (TI)	Fixing remarks by internal reviewer

1.2 Purpose and Scope of the Document

The purpose of this document is to outline preliminary algorithmic scalability contributions. Scalability, as the goal of SPEEDD/WP6, is a goal which immerge out of the need to handle big datasets using less resources, for instance when fraud detection models operate on millions of daily transactions. Horizontal scalability would mean that the size of the data mandates processing by a distributed system. Vertical scalability is required when the data, or the stream of arriving events, is processed on a single machine, hence better algorithms are necessary to meet the processing requirements. The approaches we present are both in the vertical scalability scope, where minimizing the use of local resources is the focus of the optimization, as well as in the horizontal scalability scope, which attempts to reduce the required communication to a minimum. During later stages of the project these contributions may be extended into complete solutions.

Two new approaches will be described. The first deals with the identification of complex events while spending less local resources, mainly cpu time and memory space. The basic idea is to look for the least probable event before scanning for the other events which compose the complex event. This reduces the number of scans and the amount of intermediate states which need be stored.

The second novel algorithmic approach allows to distributively check the validity of known models using uptodate data. Local constraints are developed to be checked at each participating node. As

long as the constraints hold at all nodes no communication is needed. When the new data requires training of new models, the local constraints are guaranteed to be violated and the nodes synchronize by communicating indicative states. Once a decision is made that the old model is no longer relevant, a new model may be trained, a model which is tailored to the newer data.

The application of the this last generic method will begin by applying it to linear regression. Linear regression may be used for prediction purposes, in order to distributively identify some global conditions which evolve and which require intervention. Then, at a later and more involved stage we will attempt to apply this method to classification via SVM (clearly applicable to our usecases).

1.3 Relationship with Other Documents

We are already working on the integration of the CEP detection technique into the SPEEDD architecture. This work will appear in later stages and later deliverables.

The work on horizontal scalability requires inter-client communication. This will rely on and make use of architectural enhancements of Proton and Storm which are taking place as part of our work in Ferarri project.

2

Lazy Evaluation Methods for Detecting Complex Events

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 interest is a sequence, where we are interested in identifying the arrival of a number of primitive events on the stream in a predefined order. Many popular CEP systems employ Non-deterministic Finite Automata (NFA) arranged in a chain topology for detecting sequences. Existing NFA-based mechanisms incrementally extend previously observed *prefixes* of a sequence until a match is reached. Consequently, each newly arriving event needs to be processed to determine whether a new prefix is to be initiated or an existing one is to be extended. This approach may be very inefficient when events at the beginning of the sequence are very frequent.

Our first contribution is to propose a chain topology NFA for detecting sequences in a lazy manner. This lazy mechanism 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 selectivity order to be defined in advance. We formally show that this tree-structured NFA is at least as efficient as the chain-structured NFA arranged in the best performing selectivity order. Finally, we present an experimental evaluation of queries on real-world stock trading data which demonstrates a performance gain of two orders of magnitude, while requiring only half of the memory resources.

2.1 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 Demers et al. (2006), RFID-based inventory management Wang and Liu (2005), click stream analysis Sadri et al. (2004), and electronic health record systems Harada and Hotta (2005). 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 $X^n_{p=y}$ 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, which was omitted from the notation for the sake of brevity). Say the trading firm would like to detect a sequence consisting of the events $A_{p=p1}$, $B_{p=p2}$, and $C_{p=p3}$ 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 (NFA), which are in turn used by the event processing engine. Wu et al. (2006), 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 consisting of the match prefix. Gyllstrom et al. (2008) propose additional operators to SASE such as iterations and aggregates . Demers et al. (2006, 2007) 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. The Complex Events Specification language Tesla Cugola and Margara (2010) extends previous works by offering fully customizable policies for event detection and consumption. NextCEP Schultz-Møller et al. (2009) enables distributed evaluation using NFA in clustered environment.

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:

$$A^1_{p=3}, A^2_{p=5}, A^3_{p=8}, B^1_{p=7}, B^2_{p=13}, C^1_{p=9}$$

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 $\{A_2B_1C_1\}$[1].

To the best of our knowledge, all previously proposed NFA matching mechanisms construct 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 if it starts a new prefix, or extends an existing one. In cases where the first events in a sequence pattern are very frequent, the NFA must maintain a large number of match prefixes which may eventually not lead to any matches. Since the number of match prefixes to be kept can grow exponential with the length of the sequence, such an approach may be very inefficient in terms of memory and computational resources.

In this document we propose new NFA based matching mechanisms that overcome 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

[1] An important aspect of the semantics of an event definition language is the consumption policy. The consumption policy specifies how to handle to a particular event once it is included in a match, i.e. whether it can still be reused for other matches, or should be discarded. For the purpose of our discussion in this work, we assume Reuse Consumption Policy, which means that an event instance can be included into an unlimited number of matches Etzion and Niblett (2010).

adding events in descending order of selectivity (rather than according to their order in the sequence). As a result, the number of partial matches held in memory is minimized. This also reduces computation time, since when processing a given event there are less partial matches to extend.

Our proposed solution relies on lazy evaluation mechanism. In contrast to previously proposed NFA based systems, which perform eager evaluation, the lazy evaluation mechanism can either process an event upon arrival, or store in a buffer referred to as the *input buffer* to be processed a at a later time if necessary. We present a new type of NFA that makes use of an input buffer to support lazy evaluation. In addition, we propose two types of NFA topologies for detecting sequence patterns, a chain NFA and a tree NFA.

A chain NFA requires specifying the selectivity order of the events in the sequence. For example, for constructing an automaton detecting the sequence A,B,C, it is necessary to specify that B is expected to be the most frequent, followed by A which is expected to be less frequent, followed by C which is expected to be least frequent.

A tree NFA also employs lazy evaluation, but it *does not* require specifying the selectivity order of the events in the sequence. We show that for every stream of events, a tree NFA is at least as efficient as the best performing chain NFA. Finally, we perform an experimental evaluation on real-world stock trading data which demonstrate that the tree NFA matching mechanism improves run-time performance by two orders of magnitude in comparison to existing solutions, while using only half the memory.

2.2 Related Work

The detection of complex events over streams is a very active research field in recent years Cugola and Margara (2012). The earliest systems designed for solving this problem fall under the category of Data Stream Management Systems. Most prominent examples include NiagaraCQ Chen et al. (2000), TelegraphCQ Chandrasekaran et al. (2003), Aurora Balakrishnan et al. (2004) and STREAM Group (2003). Those systems are based on SQL-like event specification languages with their focus being mainly on centralized stream-based processing. Borealis Abadi et al. (2005); Balazinska et al. (2008) is a fully distributed extension of Aurora, capable of balancing the workload among the participating nodes and handling runtime failures. Some works incorporated the traditional content-based publish-subscribe paradigm. Later, actual complex event processing systems were introduced. One example of an advanced CEP system is Amit Adi and Etzion (2004), based on a strongly expressive detecting language and featuring a component called situation manager, capable of processing notifications received from different sources in order to detect patterns of interest. SPADE Gedik et al. (2008) 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 the applications that need to handle large scale data streams. Cayuga Brenna et al. (2007); Demers et al. (2006, 2007) is a general purpose, high performance, single server CEP system developed at Cornell University. Its implementation focuses on multi-query optimization, which is directly applicable when our system is extended to handle multiple queries.

Apart from SASE, thoroughly discussed in this document, many other event specification languages were proposed. SASE+ Gyllstrom et al. (2008) is an expressive event processing language from the authors of SASE. This language extends the expressiveness of SASE, by including iterations and aggregates as possible parts of detecting patterns. CQL Arasu et al. (2006) 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)Brenna et al. (2007); Demers et al. (2006, 2007) is a declaration language used by Cauyga system, supporting patterns with Kleene closure and event selection strategies

including partition contiguity and skip till next match. TESLA Cugola and Margara (2010) is a newer declaration language, attempting to combine high expressiveness with a relatively small set of operators, achieving compactness and simplicity. In Barga et al. (2007) the authors present CEDR, an event streaming system that introduces a new temporal stream model, based on several different timings, thus aiming to unify and further enrich query language features. Even though our work focuses exclusively on sequence patterns as defined in SASE, extensions to other operators are possible, including those added to the standard set by the aforementioned languages.

Query rewriting as yet another optimization research avenue was widely studied as well. NextCEP Schultz-Møller et al. (2009) is a distributed complex event processing system, especially designed for query rewriting and distribution. Event patterns are specified in a high-level query language and, before being translated into event automata, are rewritten in a more efficient form. Automata are then distributed across a cluster of machines for detection scalability, while different operators of the same pattern are places on different hosts. Liu et al. (2010) describes NEEL, a CEP query language for expressing nested CEP pattern queries. Pattern rewriting rules are designed for pushing negation into inner subexpressions, and a normalization procedure based on these rules is defined for simplifying a nested complex event expression. In Ding et al. (2008), the authors attack the problem on exploiting event constraints to optimize CEP over large volumes streams by developing a runtime query unsatisfiability (RunSAT) checking technique that detects optimal points for terminating query evaluation. Neither of the works mentioned above explicitly addresses sequence patterns optimization by re-ordering the initial sequence, but instead focus on modifying the order of operations applied.

Unlike most recently proposed CEP systems, which use non-deterministic finite automata (NFA's) to detect patterns, ZStream Mei and Madden (2009) uses tree-based query plans for the representation of query patterns. By carefully designing the underlying infrastructure and algorithms, ZStream is able to unify the evaluation of sequence, conjunction, disjunction, negation, and Kleene closure as variants of the join operator. While some ideas discussed in this work are close to ours, it considers a tree-based model as opposed to our work, whose focus is on a model of finite automata.

To the best of our knowledge, the only work mentioning the concept of lazy evaluation in the context of event processing is Akdere et al. (2008). Here, the authors describe "plan-based evaluation", where, similarly to our work, temporal properties of primitive events can be exploited to reduce network communication costs. As this work focuses on network traffic minimization, we believe that it is orthogonal to ours.

A large volume of work has been devoted to scaling the CEP detection by distributing the tasks among several nodes. Lakshmanan et al. (2009) proposes a way to create an automatic partition of event processing entities (agents) into groups called "strata" by analyzing the semantic dependencies among the different agents using a stratification principle. Another work, Hirzel (2012), proposes a pattern syntax and translation scheme organized around the notion of partitions, thus allowing for easy parallelization of a query. Gu et al. (2007) describes an adaptive load diffusion algorithm to enable scalable processing of multiway windowed stream joins. The load diffusion is achieved by a set of semantics-preserving tuple routing algorithms. Even though our work does not explicitly address distributed environment, the described framework can be parallelized by applying a solution of such kind, since each NFA instance is completely independent of the other ones.

2.3 Notations and Terminology

In this section, we formally describe the eager NFA matching mechanism. We present a subset of the SASE language for defining sequence patterns. We formally describe the eager NFA matching mechanism, how a given sequence is compiled into an NFA, and how this NFA is used in runtime to

detect the pattern.

2.3.1 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. Additional operators proposed in the literature include Kleene closures, time windows, filtering and transforming rules, complex join operators, and aggregations. Examples for event specification languages include SASE Wu et al. (2006), SASE+ Gyllstrom et al. (2008), CQL Arasu et al. (2006), Cayuga Brenna et al. (2007), NextCEP Sadri et al. (2004). As described in the following section, patterns expressed in these languages will be compiled into a state machine for use by the detection mechanism.

The language we use in this document is based on SASE, thoroughly described in Agrawal et al. (2008). SASE combines a simple, SQL-like syntax with high degree of expressiveness, allowing to define a wide variety of patterns. A formal model exists which precisely describes the semantics and expressive power of the language. 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. In this work, we assume that all attributes are either numeric or categorical, however it is easy to extend the methods we present to other data types as well.

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 regarding the types of the participating events, and the relations among them. In this work we limit the discussion to sequence patterns. A sequence is defined using the operator *SEQ(A a, B b,...)*, 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 simple event, and the last one. This time interval is denoted by W.

```
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
```

2.3.2 The Eager Evaluation Mechanism

In this section we formally describe the structure of the eager NFA, and how it is used to detect patterns. Formally, an NFA automaton 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:

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

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

The runtime engine runs multiple instances of a NFA in parallel (equal to the number of valid prefixes over currently available primitive events in the time window), one for each partial match detected up to that point. Each NFA instance is associated with a *match buffer*. The match buffer is used for storing the primitive events constituting a partial match as we proceed through an automaton towards the final state. It is always empty at q_1, and more events are gradually added to it during the evaluation. 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. Note that the condition on an edge may also reference events in the match buffer.

The match buffer should be thought of as a logical construct. As discussed in Section 2.3.3. 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.

If, during the traversal of an NFA instance, the final state is reached, the content of a 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 associated match buffer are discarded.

The action associated with an edge is performed when the edge is traversed. The action can be one of the following (the actions are simplified versions of the ones defined for SASE Agrawal et al. (2008)):

- *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 throws it away instead of storing in any kind of buffer).

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.

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

The self loops for all event types have an *ignore* action. The edge leading from the n^{th} state to the next one has a *take* action with the event type of the n^{th} event in the sequence.

To describe the conditions on the edges, we define an auxiliary predicate, known as the *timing predicate*,[2] and denoted by p_t. The timing predicate checks whether the match buffer still adheres to the timing constraint, i.e. $p_t = t_{min} > now() - W$. The condition on self loops is p_1 (i.e. the events in the match buffer are still within the allowed time interval). The conditions in the WHERE part are mapped to the conditions on the connecting edges as follows:

[2] We do not use timed automata because the SASE model which we extend does not allow it.

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.

Figure 2.1. illustrates the NFA compiled for the pattern described, in Example 1. The edge from $q_1$to q_2 will only contain a part of the global condition considering 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.

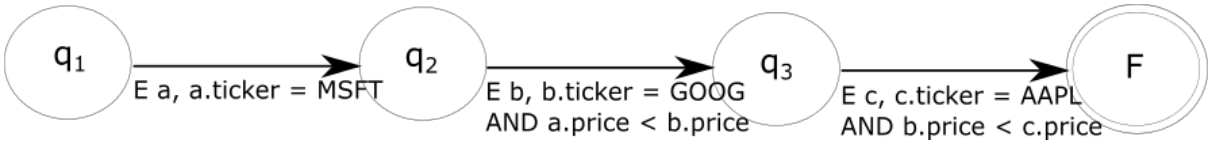

Figure 2.1: NFA for Example 1

An algorithm is described in Agrawal et al. (2008) for automatic conversion of simple SASE pattern queries (without iterations or compositions) into an NFA.

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 consists of the current state and a match buffer. Upon startup, the system consists of a single instance of the NFA at the initial state and an empty match buffer. Every event received on the input stream will be applied to all the NFA instances. In case 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 associated match buffer are discarded. In case the timing predicate is satisfied, an event will cause at least one edge traversal at each NFA instance (the self loop), but in some cases it may cause an additional traversal on a connecting edge. If only the self loop is traversed the event will be ignored. If, on the other hand, both the self loop and the connecting edge are traversed, the instance is duplicated, and on one copy the self loop is traversed (i.e. it remains unchanged), and on the other the connecting edge is traversed (i.e. the current state is modified to be the next state, and the incoming event is added to the match buffer).

2.3.3 Implementation Issues

Agrawal et al. Agrawal et al. (2008) describe a data structure referred to as the shared match buffer, for storing the content of multiple match buffers in a compact manner. The shared match buffer can be employed in both the eager and lazy NFA models to reduce memory consumption.

2.4 Lazy Evaluation

To demonstrate the effectiveness of the lazy evaluation mechanism, consider a scenario where in a certain day primitive events corresponding to a and b (stocks of MSFT and GOOG) are very frequent, while events corresponding to c (stocks of AAPL) are relatively rare. More specifically, assume that within a

time window of *t* we receive 100 instances of MSFT stock events denoted $A^1_{p_1}$,...,$A^{100}_{p_{100}}$, followed by 100 instances of GOOG stock events denoted $B^1_{p_{101}}$,...,$B^{100}_{p_{200}}$, followed by a single instance of AAPL stock event denoted $C^1_{p_{201}}$. In addition, let us assume that there is only a single $B^i_{p_i}$ 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 significantly 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.

In this section we describe the lazy evaluation model, which 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 having this name We begin by outlining the required modifications to the eager NFA model so that it can support lazy evaluation. We proceed to describing how a lazy *chain based NFA* can be constructed assuming we are given the selectivity order of the events participating in the sequence. We then present a lazy *tree based NFA* that *does not* require specifying the selectivity order of the events. Finally, we discuss the metrics we use to evaluate the computational and memory complexities of the detection methods.

2.4.1 Formal Definition of the Lazy NFA Model

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 *store* action is traversed, the event causing the traversal in 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 *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'll refer to the order provided in the input query as *sequence order*, and to the actual evaluation order as *selectivity order*. As an example, say we would like to detect the pattern $SEQ(A\,a, B\,b, C\,c, D\,d)$. In addition, assume we wish to construct a lazy NFA that first matches *a*, then *b*, followed by *d*, and finally *c*. In this case, our sequence order is A, B, C, D while our selectivity order is A, B, D, C.

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, *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 an above example, to construct a lazy NFA using the selectivity order A, B, D, C we'll assign *take(a)* edge to its first state, *take(b)* to its second state, *take(d)* to the third state and, finally, *take(c)* to the fourth and final state.

Finally, the model must include a mechanism for accessing events in the input buffer. For that purpose, we change the semantics of *take* action. Whereas in the eager NFA model an event accepted by this type of edge is always consumed from the input stream, we would like to extend this functionality by adding a search operation on the contents of the input buffer as well. In the lazy NFA model, a take action additionally triggers a search inside the input buffer, which returns events with appropriate

name to be examined for 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, evaluation will be executed non-deterministically by spawning additional NFA instances.

Continuing the example above, let our desired pattern be $SEQ\,(A, B, C, D)$ and let our selectivity order be A, B, D, C. Examine the state responsible for accepting an event named c. At this stage the match buffer contains some instance of each of the types A, B and D. The input buffer will contain all the events arrived from the input stream during evaluation, among them instances of C. Since we are searching for an event which is required to precede an already arrived event d, any possible match can only be found in the input buffer and not in the input stream. When the outgoing *take(c)* edge of the current state is evaluated, a search will thus be performed in the input buffer and all c events will be returned. Assume that the search returns three appropriate events. Then, three *take* transitions will be attempted by creating two new NFA instances.

Note that in the presented example the evaluation process as described is rather inefficient. First, since the pattern requires c event to precede d, there is no need to examine events from input stream. Obviously, since we already have d event in the match buffer, we can deduce that the only potential candidates for c have arrived in the past and are located in the input buffer. Second, since the pattern requires b event to precede c, not all c events located in the input buffer are to be returned and evaluated, but only those succeeding the particular b instance located in the match buffer. In general, a "blind" search through the whole input buffer and the input stream results in a very large number of operations, which effectively nullifies any benefits gained by using lazy evaluation.

In order to overcome this problem, we introduce a mechanism of *scoping parameters*, defined and discussed in the following subsection.

Scoping parameters

As demonstrated above, lazy evaluation NFA may introduce performance bottlenecks on executing searches in the input buffer during *take* actions. This drawback follows from the fact that, in general, only a certain range of events in the input buffer are of interest at a given state. Thus, scanning the whole buffer, which may contain very large numbers of events, would significantly slow down the evaluation. Moreover, the relevant range of events is always known is advance, and hence the redundant operations can be avoided by providing a way to specify it at any state.

To support this functionality, we modify the definition of a state to include a pair of *scoping parameters*, defining the beginning and the end of the relevant *scope* respectively. *Scope* for the purpose of this discussion is defined as a time interval (possibly open and including future time) in which the event expected at a given state is required to arrive. The scoping parameters specify whether the source of events considered by *take* edges associated with this state should be the input buffer or the input stream. In case the state should receive data from the input buffer, the scoping parameters also indicate what part of the input buffer is applicable to this state.

We will demonstrate the concept of scoping parameters using the following example. Let the detection pattern be $SEQ\,(A, B, C)$. We will show the necessity of defining a scope on two different selectivity orders: A, B, C and C, A, B.

1. Examine the evaluation of the sequence A, B, C using selectivity order A, B, C. For the first state 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 in this stage the input buffer will contain no A instances, in fact only the input stream should be considered. At the next state detecting B, we are only interested in events following A (which was detected by previous state and is now located in the match buffer). By definition of the input buffer however, it can only contain at this stage B

events which have 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. Consequently, the *scope* for this state should begin with the latest event in the match (A for our case), making all (earlier) events in the input buffer irrelevant. Exactly the same holds for C, which is detected at the third state.

2. Examine the evaluation of the sequence A, B, C using selectivity order C, A, B. For the first state detecting C, no limitations can be formulated, following the observations from previous example. For the second state detecting A, we are limited to events preceding the already accepted C. 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 which have arrived before C are to be considered. Therefore, the scope for this state should be open from the left (no constraint on starting point, search is to be conducted from the beginning of the buffer) and limited from the right by the timestamp of already matched C (search is to be conducted till the timestamp of C arrival). Finally, for the third state detecting B, we have bounds both from the left and from the right, as the event B is required to arrive after already accepted A and before C. Thus, timestamps of A and C will define the scope of this state and the relevant portion of the input buffer. Obviously, B cannot be accepted from the input stream, since its scope is bounded by C from the right.

More formally, the scoping parameters of a state q are denoted by $q(s, f)$, where s is the start of the scope and f is the finish of the scope. Both parameters values can be either events names or special keywords (see below). 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 starting scoping 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 handled in this state according to sequence order has already been handled by the NFA.
- A name of an event - in this case, only events with names succeeding the corresponding event from the match buffer in the *sequence order* are read from the input buffer, The scoping event to be used in this case is the latest event preceding the current event according to sequence order that has already been handled by the NFA.

The finishing scoping parameter f can accept one of the following values:

- A name of an event - in this case, only events with 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.

To summarize, a combination of both s and f unambiguously defines the time interval for events valid for *take* edge at the given state, based on timestamps of events in the sequence order. 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 *take* operation takes place) are considered as well.

Note that, apart from the first state, no state can possess a scope allowing for the event to be accepted both from the input buffer and the input stream. This follows from the observation that, if an event is

allowed to be taken from the input stream, it must succeed in the sequence order all of the events already located in the match buffer. It can be easily shown that the latest event in the match buffer will always be the most recent event accepted, and hence the starting scoping parameter will prevent the state from searching in the input buffer.

Before we can formalize the notion of scoping parameters, some preliminary definitions are needed.

Given an order ord and a particular event e included in this order, let $Prec_{ord}(e)$ denote all events preceding e in ord. Similarly, let $Succ_{ord}(e)$ denote all events succeeding e in ord. Additionally, given a set E of events and an order ord, 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 and sel denote the sequence order and the selectivity order, respectively.

Now we are ready to define the rules for setting scoping parameters. Let e_i be the i^{th} event in the selectivity order and let q denote the corresponding state in the chain. Then, the scoping parameters for q are defined as follows:

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

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

We'll demonstrate the definitions above on examples from the beginning of the section.

1. Evaluation of the sequence A, B, C using selectivity order A, B, C. For the first state detecting A, the scoping parameters will be $q_1(start, end)$. For the next state detecting B, the scoping parameters will be $q_2(A, end)$. Finally for the following state detecting C, the scoping parameters will be $q_3(B, end)$.

2. Evaluation of the sequence A, B, C using selectivity order C, A, B. For the first state detecting C, the scoping parameters will be $q_1(start, end)$. For the next state detecting A, the scoping parameters will be $q_2(start, C)$. Finally for the following state detecting B, the scoping parameters will be $q_3(A, C)$.

2.4.2 Chain Based NFA

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

The chain based NFA utilizes the constructs for the lazy evaluation model so that events are evaluated according to a given selectivity order (rather than according to the sequence order). It consists of $n+1$ states, where each of the first n states is responsible for detecting one primitive event in the pattern, and the last is the accepting state. The states are arranged according to the given *selectivity order*. We assume this order for now to be given in advance.

We'll denote by e_i the i^{th} event in the selectivity order and by q_i the corresponding state in the chain. The scoping parameters for q_i will be defined as specified above. Upon an arrival of an event a state is responsible for, it will apply the scoping parameters to find an instance of e_i which matches the events already located in the match buffer.

Let E_i denote the set of outgoing edges of q_i. 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 which may be potentially taken in one of the following states is stored into the input buffer.
- $e_i^{store} = (q_i, q_{i+1}, take, e_i, cond_i \wedge InScope_i)$ - an event with a 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 state (denoted by a predicate $InScope_i$).

The chain based NFA will thus be defined as follows:

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

where:

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

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

The following figure demonstrates the chain NFA for the pattern in Example 1. For simplicity, *ignore* edges are omitted.

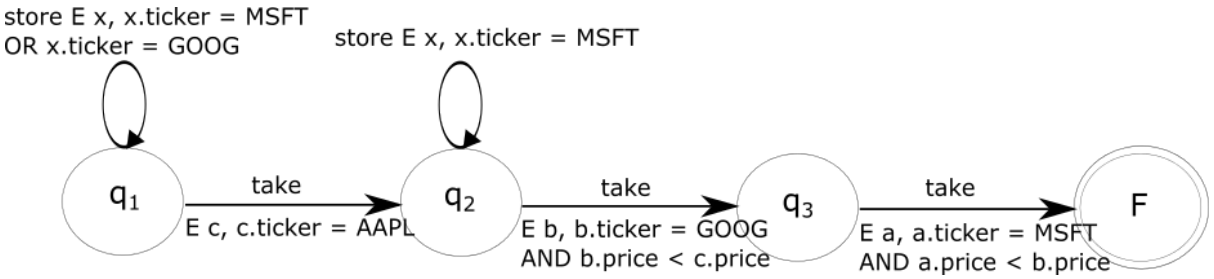

Figure 2.2: Chain based NFA for Example 1

The following theorem formally claims the equivalence of the eager NFA and the chain based NFA defined above.

Theorem 1: Given an order seq over a set of events names $e_1, \cdots, e_n$, a chain based NFA for any selectivity order sel over the same set of events names is equivalent to an eager NFA detecting a sequence in respect to seq.

Proof: the proof is by induction on all selectivity orders.

For the case $seq = sel$ (the sequence and the selectivity orders are identical) the scoping parameters of any state will point to the end of the input buffer, and all events will be taken from input stream, hence the conditions on edges will become the same as in the eager NFA. In addition, the order of states will be the same as in the eager NFA. Consequently, the transition function between states is the same, making the two automata identical.

For the induction step, assume that the claim holds for any selectivity order which can be obtained from seq by performing k swap operations. W.l.o.g. let A_k be some chain based NFA accepting the sequence seq in some selectivity order sel_k satisfying the condition above. We'll prove that performing an additional swap operation doesn't change the language accepted by the NFA, i.e. that any new chain based NFA A_{k+1} obtained from A_k by performing a swap between two event names is equivalent to A_k. Then, by induction hypothesis A_{k+1} is also equivalent to the original eager NFA.

Let A_{k+1} be identical to A_k with the events names E_i and E_j swapped. Assume w.l.o.g. that E_i precedes E_j both in sel_k (the opposite case can be proven symmetrically).

Let $e_1, \cdots, e_i, \cdots e_j, \cdots, e_n$ be a sequence accepted by A_k. Examine the state of A_k accepting e_i. There are two possibilities:

1. The event e_i was accepted from the input buffer by a corresponding *take* edge and matched to the events located in the match buffer at this point - then in A_{k+1}, when the state responsible for e_i is reached (we're guaranteed it will eventually be reached since this match is accepted by A_k and other than the swap between e_i and e_j no modifications were made), the event e_i will necessarily still be located in the input buffer - otherwise, events comprising a match are not located within the predefined time window, which is a contradiction. In addition, $Prec_{sel_k}(e_i) \subseteq Prec_{sel_{k+1}}(e_i)$, hence e_i will be valid in respect to the scoping parameters of the state accepting it and will be thus added to the match buffer.

2. The event e_i was accepted from the input stream by a corresponding *take* edge and matched to the events located in the match buffer at this point - then in A_{k+1}, the event e_i will either be located in the input buffer and become part of the match (by same observations used in (1)) or will eventually be accepted from the input stream. The latter is correct because: 1)all events comprising a match are located within the time window (hence e_i will arrive before the timeout); 2)the scoping parameters will necessarily allow to accept e_i from input stream in A_{k+1}. This follows from the observation that $Prec_{sel_{k+1}}(e_i) \cap Succ_{seq}(e_i)$must be empty in order to allow e_i and all subsequent events till e_j to be accepted in A_k.

To summarize the above, in any case e_i will be accepted to the match buffer during the evaluation in A_{k+1}.

Now, we'll examine the state of A_k accepting e_j. Again, there are two possibilities:

1. The event e_j was accepted from the input buffer by a corresponding *take* edge and matched to the events located in the match buffer at this point - then in A_{k+1}, the event e_i will either be located in the input buffer (arrived before the event preceding e_i in sel_k, and thus also preceding e_j in both selectivity orders) or will eventually be accepted from the input stream - since all events comprising a match are located within the time window, hence e_j will necessarily arrive before the timeout, and the scoping parameters of e_j in A_{k+1} will necessarily allow to accept the event from input stream (otherwise, it means that an event from $Succ_{seq}(e_i)$ is already located in the input buffer and hence the evaluation would not reach e_j in A_k, which is a contradiction).

2. The event e_j was accepted from the input stream by a corresponding *take* edge and matched to the events located in the match buffer at this point - then in A_{k+1}, when the state responsible for e_j is reached, the event e_j will never be located in the input buffer, and the evaluation will be stuck there until e_j arrives. As mentioned above, we're guaranteed that it will arrive before the timeout. In addition, $Prec_{sel_{k+1}}(e_j) \subseteq Prec_{sel_k}(e_j)$, hence the ending scoping parameter of the corresponding state in A_{k+1} will necessarily allow to accept e_j from the input stream. Consequently, $e_j$will be added to the match buffer upon its arrival.

Since no other modifications were made except for swapping between e_i and e_j, in can be concluded that A_{k+1} will accept the match.

The proof for the opposite direction (any match accepted by A_{k+1} is accepted by A_k) is symmetrical and will be omitted.

In conclusion, we have proven the equivalence between A_k and A_{k+1}, and by induction hypothesis A_{k+1} is also equivalent to the eager NFA A, which completes the proof. ■

2.4.3 Tree Based NFA

Chain based NFA described in the previous section may significantly improve evaluation performance, provided the correct order of selectivity. As could be seen from the examples above, the more drastic is the difference between arrival rates of different events, the greater the potential improvement is.

There are, however, several drawbacks which severely limit the applicability of chain based 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 large number of 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 it to 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, causing an initially efficient chain based automaton to start performing poorly from some point. Continual changes may come, for example, in the form of bursts of usually rare events.

In order to overcome the problems described above, we introduce a notion of ad-hoc selectivity. Instead of relying on a single selectivity order specified at the beginning of the run, the idea is to determine the current selectivity on-the-fly and modify the actual evaluation chain of a match according to the currently correct order. By checking and verifying the perfect selectivity order at each evaluation step (at each state on a path towards the accepting state) we guarantee that, for any partial match, its evaluation was executed in the best order possible at the moment.

To implement the desired functionality, we will make use of the input buffer introduced above. According to its definition, the input buffer of a particular NFA instance contains all events that arrived from the input stream within the specified time window. Hence, by maintaining counters on each event name, incrementing a corresponding counter on each insertion of a new event and decrementing it on event removal, we can derive the exact selectivity order on currently available data. These counters will be examined by each state on each matching attempt, and the resulting value will be used for making a decision regarding 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 achieve this, a state needs to possess several outgoing *take* edges (as opposed to a single one in chain based NFA), which operate in the exact same way but point at different states. In other words, an automaton has to possess the structure of a tree, whose branching factor and depth are equal to the number of event names in the pattern. We will call NFA employing this structure *tree based NFA* and will formally define this model below.

One important observation to be made regarding the aforementioned operation of inspecting the contents of the input buffer is the following one: there is no need to start the evaluation process unless there is at least one event corresponding to each event name in the pattern. Only when all of the events counters are greater than zero, it does make sense to decide regarding the evaluation order, since otherwise the missing event(s) may not arrive at all and partial matching process will be redundant. Thus, our model has to make sure no operations are performed while the input buffer does not contain at least a single event for each event name.

Figure 2.3 demonstrates the tree NFA for the pattern in Example 1. For simplicity, ignore edges are omitted.

We will now present the formal definition of tree based NFA.

The NFA is structured as a tree of depth $n-1$, the root of this tree being the initial state and the leaves connected to the accepting state. Nodes located at each layer k; $0 \leq k \leq n-1$ (i.e. all nodes in depth k) are all states responsible for all orderings of n events of length k. Each such node, possesses

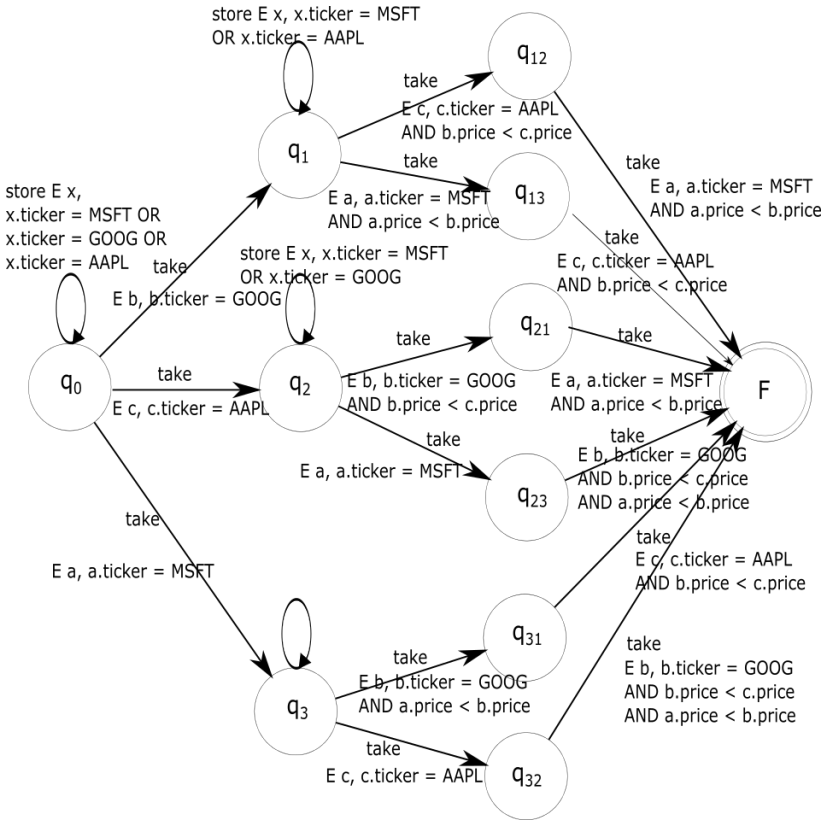

Figure 2.3: Tree based NFA for Example 1

$n-k$ 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 exception to this rule are the leaves, which has a single outgoing edge, connected directly to the final state.

More formally, the states for tree based NFA are defined as follows. Let O_k denote the set of all sequences of events $e_1, \cdots, e_n$ in the pattern with no repetitions. 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 based NFA is

$$Q = \bigcup_{k=0}^{n-1} Q_k \cup \{F\}$$
$$q_0 = q_\oslash$$

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

First, we'll define scoping parameters for states of tree based NFA. Since, as mentioned earlier, each state q_{ord} has an outgoing edge for each event $e \notin ord$, the scoping parameters will be defined separately for each such event. The definition is similar to the one used for chain based NFA, with the selectivity

order sel being removed and the partial order ord associated with a state used instead:

$$s\left(q_{ord}, e\right) = \begin{cases} Latest_{ord}\left(ord \cap Prec_{seq}\left(e\right)\right) & if\ ord \cap Prec_{seq}\left(e\right) \neq \{\} \\ start & otherwise \end{cases}$$

$$f\left(q_{ord}, e\right) = \begin{cases} Earliest_{ord}\left(ord \cap Succ_{seq}\left(e\right)\right) & if\ ord \cap Succ_{seq}\left(e\right) \neq \{\} \\ end & otherwise \end{cases}$$

Similarly to the chain based NFA, the predicate $InScope_{ord}\left(e\right)$ will denote that an event e is located inside the scope $\left(s\left(q_{ord}, e\right), f\left(q_{ord}, e\right)\right)$.

Let c_e denote the value of the counter of events associated with the name e in the input buffer. Let $se\left(q_{ord}\right) = min\left(\{c_e | e \notin ord\}\right)$ denote the most selective event according to the contents of the input buffer during the evaluation step in which q_{ord} is the current state (i.e. the event with the smaller amount of occurrences inside the input buffer). Finally, we'll define the predicate $p_{ne}\left(q_{ord}\right)$ (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 $p_{se}\left(q_{ord}, e\right)$ to be true if and only if an event$e$ corresponds to event type $se\left(q_{ord}\right)$.

Let E_{ord} denote the set of outgoing edges of q_{ord}. Then, E_{ord} will contain the following edges:

- $e_{ord}^{ignore} = \left(q_{ord}, q_{ord}, ignore, ord, true\right)$ - 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 $e \notin ord$:
  - $e_{ord,e}^{store} = \left(q_{ord}, q_{ord}, store, e, \neg p_{ne}\left(q_{ord}\right) \vee \neg p_{se}\left(q_{ord}, e\right)\right)$- while either p_{ne} or p_{se} condition is not satisfied, the incoming event is stored into the input buffer.
  - $e_{ord,e}^{take} = \left(q_{ord}, q_{ord,e}, take, e, p_{ne}\left(q_{ord}\right) \wedge p_{se}\left(q_{ord}, e\right) \wedge cond_e \wedge InScope_{ord}\left(e\right)\right)$ - if the contents of the input buffer satisfy p_{ne} and p_{se} predicates and an incoming event with a name e: a)satisfies the conditions required by the initial pattern (denoted by $cond_e$); b)is located inside 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

$$e_{ord,e}^{store} = \left(q_{ord}, F, take, e, p_{ne}\left(q_{ord}\right) \wedge cond_e \wedge InScope_{ord}\left(e\right)\right)$$

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

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

And the NFA itself is defined as follows:

$$A = \left(Q, E, q_1, F, \right)$$

where Q and E are as defined above.

2.4.4 Implementation Issues

The tree based NFA described in the previous section possesses an important drawback, which is the total number of states 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 will be terminated, the state will be removed from the NFA as well. Even though the worst case complexity remains exponential in this case, in practice, the relative event rates will change significantly less frequently than the rate of NFA instance creation (in other words, a change in relative frequencies of primitive events - which will lead to modification of our detection order and creation of new nodes - is very rare when compared to a creation of an NFA instance which occurs on any event arrival). This conclusion is supported by the experiments conducted by us, which are explained in the respectful section.

2.4.5 Evaluation Metrics

As a measure of runtime complexity, we count the number of times a condition on a connecting edge is evaluated. For instance, consider the pattern from Example 1 and two following streams of events: $A^1_{p=3}$,$B^1_{p=7}$,$C^1_{p=9}$ and $A^1_{p=3}$,$B^2_{p=13}$,$C^1_{p=9}$. 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.

Memory consumption consists of the weighted sum of two basic metrics - peak number of simultaneously active NFA instances and, for the case of NFA_{lazy} model, peak number of buffered events awaiting to be processed.

2.5 Optimality of the Tree Based NFA

In this chapter we will provide the formal proof of the following theorem.

Theorem 2: Let P be some sequence pattern, requesting sequence order seq. Let sel be a selection order under which the chain based NFA as defined above provides the best performance in terms of number of edge evaluations, and let A^{chain}_{sel} be that chain NFA. Let A^{tree}_{seq} be the tree based NFA for seq as defined above. Then, the performance of A^{tree}_{seq} is at least as good as that of A^{chain}_{sel}.

First, we'll make an important observation. It states that, no matter which NFA is in use, the amount of condition evaluations required for a successful match is always the same.

Lemma 1: given a subset of events from the input stream constituting a valid match for the pattern, the amount of condition evaluations performed during the detection of this match is identical for any chain based NFA and the tree based NFA, and equals $length\,(seq)$.

Proof: since the match is a successful one, its respectful NFA instance will reach the final state F. For a chain based NFA, by its definition there will be a single edge transition executed for each primitive event in the sequence, regardless of the selectivity order, since there is only a single valid path from the initial to the final state. For a tree based NFA, the length of any path from the initial to the final state equals the length of the sequence (as follows from its layers definition) and contains each primitive event exactly once (as follows from the condition $e \notin ord$ for any outgoing *take* edge of q_{ord}). Consequently, the statement above is true for any chain based NFA and for a tree based NFA.

Corollary: while comparing between two chain based NFA, or between a chain based NFA and a tree based NFA, in terms of edge conditions evaluated on a given input, only those combinations of

primitive events comprising unsuccessful matches are to be considered.

Before we proceed, we will formally define the framework for comparing a pair of NFA by number of edge evaluations on unsuccessful matches from the given stream of events. We'll assume all inter-event conditions to only be defined between a pair of primitive events. The definitions below and the proof can be easily extended to handle cases of conditions between multiple events.

Let S be some ordered input stream of primitive events $e_1, e_2, \cdots, e_n$. We'll assume all events in S to fall within the time window defined by the pattern (the proof for the case in which this assumption is dropped is very similar to the one shown below). We'll denote the number of primitive events of type e_k in S by c_k.

Let $m = (e_1, e_2, \cdots, e_n)$ denote some set of primitive events from S comprising an unsuccessful match, where $e_1, e_2, \cdots, e_n$ are event names corresponding to the primitive events in the sequence requested by the pattern. Then, by definition of m, there exists at least a single pair of primitive events e_i, e_j such that the mutual condition between e_i and e_j as defined at the WHERE part of the pattern is not satisfied. For the sake of simplicity and without loss of generality we'll assume there is exactly one such pair. We'll call it a *violating event pair* of m and denote by $e_i(m), e_j(m)$.

Assume m was received as an input for some particular instance of either chain based or tree based NFA. Let ord be the actual evaluation order in which primitive events are processed. Note that, for chain based NFA, this order will always be the predefined selectivity order sel, and for tree based NFA it will be inferred from the input during the run. Then, the automaton will recognize the match as failed when the latter of the two events in the violating pair will be evaluated. Formally, we can define the number of edge evaluations executed while processing m by order ord as follows:

$$cost_{ord}(m) = max\left(index_{ord}(e_i(m)), index_{ord}(e_j(m))\right)$$

Now we make our next important observation. Note that, for any pair e_i, e_j of events violating their mutual condition, any match located in the input stream S is unsuccessful. In other words, any combination of this pair with other events from the stream containing exactly one instance of each event type yields an unsuccessful match. Our next lemma will give an expression for the total number of edge evaluations performed as a result of having a violating pair e_i, e_j in the input stream. W.l.o.g. we assume that e_i precedes e_j in ord.

Lemma 2: For a violating pair e_i, e_j, denote by $U_{ord}(e_i, e_j)$ the total amount of unsuccessful matches evaluated by a chain based NFA or tree based NFA under order ord. Let $E_{ord}(e_i, e_j)$ denote the set of all primitive events preceding the latter of e_i and e_j (i.e. e_j) in ord. More formally, let

$$E_{ord}(e_i, e_j) = \{e_k | k \neq i \wedge k \neq j \wedge (index_{ord}(e_k) < index_{ord}(e_j))\}$$

Then, the amount of unsuccessful matches is:

$$U_{ord}(e_i, e_j) = \prod_{e_k \in E_{ord}(e_i, e_j)} c_k$$

Proof: assuming evaluation order ord, all NFA instances constructed to process partial matches containing e_i, e_j will be discarded once the latest of the violating pair is processed, since at that point the violation will be discovered. Until that point, any combination of preceding primitive events with the earlier of e_i, e_j will be treated as a unique partial match.

Corollary: The total amount of edge evaluations resulting from processing unsuccessful matches containing a violating pair e_i, e_j is:

$$cost_{ord}(e_i, e_j) = U_{ord}(e_i, e_j) \cdot cost_{ord}(m) = \left(\prod_{e_k \in E_{ord}(e_i, e_j)} c_k\right) \cdot index_{ord}(e_j)$$

To simplify the above definition and the continuation of the proof, we'll define:

$$\prod_{i,j} = \prod_{k=i}^{j} c_k$$

$$\prod_{j} = \prod_{1,j}$$

Now, the definition of $cost_{ord}\left(e_i, e_j\right)$ becomes:

$$cost_{ord}\left(e_i, e_j\right) = \frac{\prod_{j-1}}{c_i} \cdot index_{ord}\left(e_j\right)$$

Now, we will show that the order minimizing the average $cost_{ord}\left(e_i, e_j\right)$ over all possible selections of e_i, e_j is the one pushing more frequent events as far as possible.

Lemma 3: Given a sequence pattern and an input stream S with events $e_1, e_2, \cdots, e_n$ appearing $c_1, c_2, \cdots, c_3$ times respectively, the evaluation order minimizing the average $cost_{ord}\left(e_i, e_j\right)$ for any selection of a violating event pair e_i, e_j is the one in which events appear in descending order of selectivity, i.e.:

$$min_{ord}\left(AVG_{e_i,e_j}\left(cost_{ord}\left(e_i, e_j\right)\right)\right) = \left(e_{k_1}, e_{k_2}, \cdots e_{k_n}\right);\ c_{k_1} \leq c_{k_2} \leq \cdots \leq c_{k_n}$$

Proof: since the number of different choices of e_i and e_j is independent of the chosen order, we'll show the order maximizing the sum of all costs, and the same order will maximize the average as well.

Let $ord = e_1, e_2, \cdots, e_n$ be some order on the primitive events. Let $SUM\left(ord\right)$ denote the sum of all $cost_{ord}$ for this order, and let $SUM_j\left(ord\right)$ denote the sum of $cost_{ord}\left(e_i, e_j\right)$ for a given j and any selection of i. In other words,

$$SUM_j\left(ord\right) = \sum_{i=1}^{j-1} cost_{ord}\left(e_i, e_j\right)$$

$$SUM\left(ord\right) = \sum_{j=2}^{n} SUM_j\left(ord\right)$$

We'll explicitly calculate the expressions above:

$$SUM_j\left(ord\right) = \sum_{i=1}^{j-1} \frac{\prod_{j-1}}{c_i} \cdot j = \left(\sum_{i=1}^{j-1} \frac{1}{c_i}\right) \cdot \prod_{j-1} \cdot j$$

For simplicity, we'll denote $\alpha_j = \sum_{i=1}^{j} \frac{1}{c_i}$. Now we get:

$$SUM\left(ord\right) = \sum_{j=2}^{n} SUM_j\left(ord\right) = \sum_{j=2}^{n} \alpha_{j-1} \cdot \prod_{j-1} \cdot j = \sum_{j=1}^{n-1} \alpha_j \cdot \prod_{j} \cdot \left(j+1\right) =$$

$$= c_1 \cdot \left(2\alpha_1 + c_2 \cdot \left(3\alpha_2 + \cdots \left(\cdots \left(c_{n-1} \cdot n \cdot \alpha_{n-1}\right) \cdots\right)\right)\right)$$

Examining the expression above, it can be observed that the most dominant term is c_1, and hence assigning the minimal possible value to it will minimize the whole expression; the second most dominant term is c_2 and therefore it should accept the minimal value among the rest (i.e. the second minimal

value) etc. In other words, the order minimizing the expression above is the one in which events appear in descending order of selectivity.

Corollary 1: Assume that conditions selectivity is uniform over the primitive event types, i.e. there exists some $0 < \alpha < 1$ such that, for any e_k, the amount of events of this type participating in violating pairs is $\alpha \cdot c_k$. Then, the order minimizing the overall cost for unsuccessful matches is one in which events appear in descending order of selectivity.

Corollary 2: Given a sequence pattern and an input stream S, the most efficient chain based NFA (in terms of executed edge conditions evaluations) is the one one in which events appear in descending order of selectivity.

The only statement left to prove is that, under the above assumptions, the evaluation order chosen by a tree based NFA for input stream S is identical to the order shown to be the most effective, i.e. the descending order of events selectivity. This will prove that the tree based NFA performs (at least) as efficiently on any given input as does the best chain based NFA.

Lemma 4: Given a sequence pattern and an input stream S, the evaluation order in which primitive events from the stream will be processed is according to their descending order of selectivity.

Proof: By construction of tree based NFA, at each state q_{ord} the next transition is taken only if the respective event is the most selective among those left for evaluation (i.e. $p_{se}\left(q_{ord}, e\right)$ predicate is satisfied). Assuming identical rates between events at each evaluation stage, the decision made at each tree layer will reflect the actual rates between events in S, and hence the resulting evaluation order will be the descending order of selectivity.

The correctness of Theorem 2 immediately follows from Corollary 2 and Lemma 4, ■.

2.6 Experimental Evaluation

In this section, we present the results of experiments executed in order to evaluate the performance of chain-based and tree-based 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, measured in a ways defined above.

Both models under examination were implemented in Java and integrated into the FINCoS framework Mendes et al. . FINCoS is a set of benchmarking tools for evaluating performance of CEP systems, developed at the University of Coimbra.

All experiments were run on a HP 2.53 Ghz CPU and 8.0 GB RAM. The data set we used is the real-world historical data of stock prices data at the NASDAQ stock market, taken from EOD. 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 possesses the following attributes: stock identifier (ticker), timestamp and the current price of the stock.

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

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 require consecutive stocks in the sequence to be highly correlated (i.e. the Pearson correlation coefficient between histories of stocks prices is 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 is a Google stock, while the first stock belongs to some hi-tech company and, finally, the second stock belongs to some finance company. The time window for event detection was set to the length of prices 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

It can be observed that in the described pattern the final event, denoted as c, has a drastically smaller frequency than its predecessors do (a and b are assumed to have the same frequency give or take, when compared to that of c). One parameter of interest which affect the overall efficiency of the presented evaluation models is its relative frequency in respect to a and b, which we will denote as f_c. Intuitively, the lower is the value of f_c, the more we would expect the performance gain of our proposed lazy evaluation mechanisms to be.

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

Figure 2.4: Comparison of NFA by number of operations

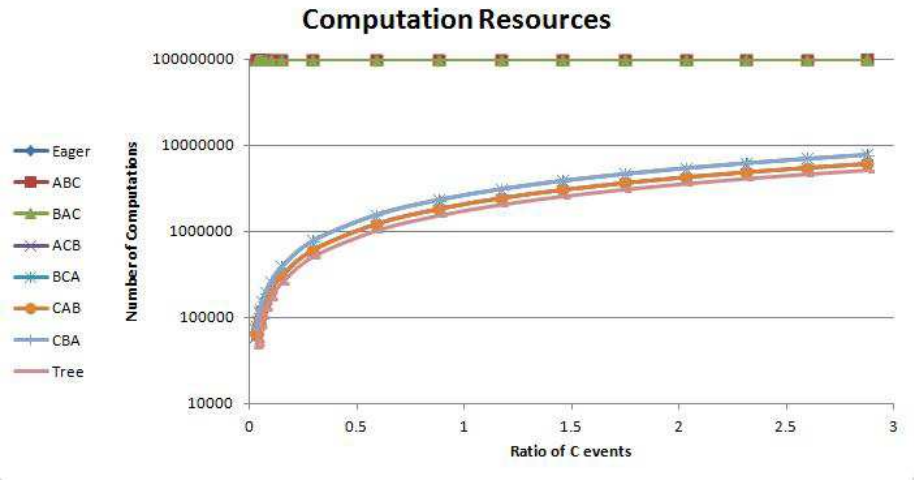

Figure 2.5: Comparison of NFA by memory consumption

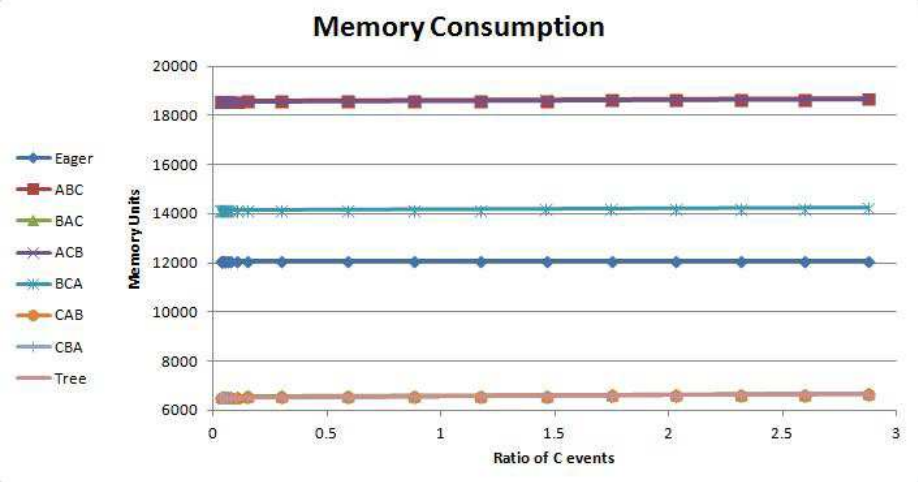

Figure 4 describes the amount of computations performed by each of the NFA as a function of f_{c} , while figure 5 presents the same comparison for memory consumption. It can be seen that both the tree based NFA and the best fitting chain based NFA for the given input (either C-B-A or C-A-B since C is a rare event while A and B share the same selectivity) achieve an improvement of one to three orders of magnitude relatively to the regular, eager NFA. In terms of memory the above NFA are more efficient by a constant factor of 3. On the other side, the chain NFA implementing selectivity orders which are suboptimal relatively to the given input never achieve worse performance than that of the eager NFA. The tree based NFA is always superior to all the other, by all metrics.

2.7 Future Work

The framework proposed is under initial implementation steps as part of SPEEDD/WP6 architecture work. Once implemented, data from SPEEDD usecases will be used as input for further evaluating the efficiency of the proposed solution. We strongly believe that in both traffic monitoring and fraud detection there is a lot of space for enhancing the framework in such a way to allow for online processing of very rapid streams. This will become especially important when traffic streams will be monitored using gps data streams (as planned for the simulator work) collected from millions of cellular phones. For fraud detection usecase scenarios, consider a future world with cardless shopping using cell phones, making the required processing infrastructure extremely ambitious.

Further improvements can be made to the solution proposed in this document. First, as of now, the only operator addressed is the sequence. Additional research is to be performed to apply the concepts of lazy evaluation and reordering by selectivity on other operators, including conjunction, disjunction, negation and more. Another enhancement to the described mechanism is taking into account the selectivity of filtering conditions. As of now, selectivity order is computed based solely on frequencies of arrival of primitive events. However, a very strict filtering condition can turn a frequent event into a rare one, and this is to be considered. Finally, extension of the presented system to distributed environment (horizontal scalability) is to be further researched.

3

Monitoring Distributed Models

3.1 Introduction

Statistical models are commonly used for prediction, interpretation, anomaly detection and more. For example, machine learning is used to produce models for the SPEEDD fraud use case. Similarly, regression models can be used to predict traffic in the traffic use case. Learning the models once is not enough, though; concept drifts can mean that the current model is no longer valid. Thus, in many real world applications it is necessary to periodically re-learn the model. This approach can be very wasteful, since learning algorithms are often orders of magnitude more demanding than applying an existing model. In this chapter we consider the following approach: monitor the quality of the current model, and only re-learn the model as needed.

The monitoring approach looks at incoming data and triggers an alert if the previously-learned model is too different from the model that *would have been built* given the current data, without actually paying the price of building the current model. This problem is made more difficult when data is distributed across several nodes, since both the existing model and the (hypothetical) current model are both *global* models – composed from the union of data at all nodes. Hence a distributed monitoring approach must deal with communication efficiency, in addition to the problem of how to effectively monitor the quality of a model without actually re-learning it.

We start with linear regression. Ordinary Least Squares regression is a well-known and very common regression model, and is useful both for predicting new values given old ones, but also for understanding behavior through discovered coefficients. Current work on distributed linear regression deals with making model learning more efficient by parallelizing model construction, i.e. the first approach. Given such a model β, we describe a method to *efficiently* monitor its on-going deviation from a hypothetical true global model – the second approach. Our monitoring approach is efficient both in terms of communication between nodes, and in terms of local computation at each node. Preliminary experiments on synthetic data show an order-of-magnitude reduction in communication.

3.2 Distributed Monitoring of Least Squares

3.2.1 Basics and Notations

Given n column vectors $\{X_1, \ldots, X_n\}$ in $\mathbb{R}^m$ where $n > m$, and response scalars $\{y_1, \ldots, y_n\}$, we seek the linear functional $\mathbb{R}^m \to \mathbb{R}$ given by an inner product with $\beta = (\beta_1, \ldots, \beta_m)^T$, which minimizes the square error between y_i to the mapping of X_i. In other words, we seek β which minimizes $\|X\beta - y\|^2$, where X is the $n \times m$ matrix of row vectors $X \triangleq (X_1, \ldots, X_n)^T$, and y is the column vector composed of response scalars $y \triangleq (y_1, \ldots, y_n)^T$.

The solution is known to be given by

$$\beta = A^{-1}c, \quad A \triangleq \sum_i X_i^T X_i, \quad c \triangleq Xy \tag{3.1}$$

Assume now that the data $\{(X_i, y_i)\}$ are dynamic and distributed between nodes $N_1 ... N_k$. It is trivial to see that the "global" A resp. c can be written as $\sum_j^k A_j$ resp. $\sum_j c_j^k$, where A_j, c_j are constructed at N_j. Therefore, we can write the global vector β as a function of the averages of A_j, c_j:

$$\beta = \left(\frac{\sum_j A_j}{k}\right)^{-1} \left(\frac{\sum_j c_j}{k}\right) = \left(\sum_j A_j\right)^{-1} \left(\sum_j c_j\right) = A^{-1}c \tag{3.2}$$

We shall denote initial values at the nodes (i.e. at time 0, when the monitored model was computed) by the superscript 0, but to avoid equation clutter, the global initial values will be denoted with a subscript. Values averaged over nodes (rather than summed) shall be denoted with $\hat{\cdot}$. Hence initial values

$$\hat{A}_0 = \frac{\sum_{j=1}^k A_j^0}{k}, \hat{c}_0 = \frac{\sum_{j=1}^k c_j^0}{k}, \hat{\beta}_0 = \hat{A}_0^{-1}\hat{c}_0 = A_0^{-1}c_0 = \beta_0 \quad ,$$

and current values

$$\hat{A} = \frac{\sum_{j=1}^k A_j}{k}, \hat{c} = \frac{\sum_{j=1}^k c_j}{k}, \hat{\beta} = \hat{A}^{-1}\hat{c} = A^{-1}c = \beta \quad .$$

3.2.2 Monitoring DLSQ with Convex Subsets

We want to impose *node-independent* condition on the data so that:

- As long as they hold, the global solution does not change by more than an allowed margin, e.g. it holds that $||\beta_0 - \beta|| \leq \epsilon$.
- The conditions are "as lenient as possible", i.e. we wish to minimize the number of violations.
- The conditions allow quick "violation recovery" (i.e. when the conditions are violated at a subset of the nodes, they can often be resolved with a low communication overhead, and not by collecting the data at all nodes).

Convex Subsets

We propose to solve the monitoring problem by means of "good" convex subsets of the dataspace: that is, to find a *convex* subset $\mathcal{C}$ in the space of matrices and vectors, such that $(0_A, 0_c) \in \mathcal{C}$, where 0_A resp. 0_c are the zero matrix resp. vector. We also want the time for a random walk starting at $(0_A, 0_c)$ to exit

$\mathcal{C}$ to be large: ass data slowly drifts over time, we want the total drift to remain in $\mathcal{C}$. It must also hold that

$$(\delta_A, \delta_c) \in \mathcal{C} \implies ||(\hat{A}_0 + \delta_A)^{-1}(\hat{c}_0 + \delta_c) - \hat{A}_0^{-1}\hat{c}_0|| \leq \epsilon \tag{3.3}$$

Given such a subset $\mathcal{C}$, the basic monitoring paradigm is simple. Given the initial values at node j when the current model was computed, as long as $(A_j - A_j^0, c_j - c_j^0) \in \mathcal{C}$, node j can remain silent. If all nodes area silent, then $\|\hat{\beta}_0 - \hat{\beta}\| = \|\beta_0 - \beta\| \leq \epsilon$. If a violation of the local condition does occur at any node j, some form of violation recovery must take place, for example recomputing the global model and restarting monitoring.

The correctness of this paradigm follows from the following observations. Write $(\hat{A}, \hat{c})$ as the average of local deviations:

$$\begin{aligned}(\hat{A}, \hat{c}) &= \frac{\sum_j (A_j, c_j)}{k} \\ &= \frac{\sum_j \left((A_j^0, c_j^0) + (A_j - A_j^0, c_j - c_j^0)\right)}{k} \\ &= (\hat{A}_0, \hat{c}_0) + \frac{\sum_j (A_j - A_j^0, c_j - c_j^0)}{k}\end{aligned} \tag{3.4}$$

And from $\mathcal{C}$'s convexity,

$$\forall j \ (A_j - A_j^0, c_j - c_j^0) \in \mathcal{C} \implies \frac{\sum_j (A_j - A_j^0, c_j - c_j^0)}{k} \in \mathcal{C} \tag{3.5}$$

Denote $(\hat{\delta}_A, \hat{\delta}_c) = \frac{\sum_j (A_j - A_j^0, c_j - c_j^0)}{k}$ and combine Eq. (3.4) with Eq. (3.5):

$$\forall j \ (A_j - A_j^0, c_j - c_j^0) \in \mathcal{C} \implies (\hat{A}, \hat{c}) = (\hat{A}_0, \hat{c}_0) + (\hat{\delta}_A, \hat{\delta}_c) \ , (\hat{\delta}_A, \hat{\delta}_c) \in \mathcal{C}$$

Substitute in Eq. (3.3) to finally obtain:

$$\begin{aligned}\forall j \ (A_j - A_j^0, c_j - c_j^0) \in \mathcal{C} \implies & \|(\hat{A}_0 + \hat{\delta}_A)^{-1}(\hat{c}_0 + \hat{\delta}_c) - \hat{A}_0^{-1}\hat{c}_0\| = \\ & \|\hat{\beta}_0 - \hat{\beta}\| = \\ & \|\beta_0 - \beta\| \leq \epsilon\end{aligned} \tag{3.6}$$

We next describe a method to find a convex subset $\mathcal{C}$ satisfying the condition of Eq. (3.3).

Some Mathematical Notions

We list some notions and well-known results on matrix norms.

Definition 1 *Let A be a matrix. Its* operator norm *(hereafter just* norm*) is defined as*

$$\sup_{x \neq 0} \frac{||Ax||}{||x||} \tag{3.7}$$

We use the L_2 norm throughout.

Lemma 1 *for a matrix A and vector x, $||Ax|| \leq ||A||\ ||x||$.*

Lemma 2 *The norm of a symmetric matrix equals the maximal absolute value of its eigenvalues.*

Lemma 3 *For two matrices A, B, it holds that $||A + B|| \leq ||A|| + ||B||, ||AB|| \leq ||A||\ ||B||$.*

Lemma 4 *For two symmetric matrices A, B it holds that $||AB|| = ||BA||$.*

Lemma 5 *If $||A|| < 1$, then $\lim_{n\to\infty} A^n = 0$, and $(I - A)^{-1}$ can be expanded in a Taylor series, $I + A + A^2 + A^3 \ldots$ (the* Neumann series *of A).*

Lemma 6 *Let A be a square, positive definite matrix, and δ a symmetric matrix of the same size as A and such that $||\delta A^{-1}|| < 1$. Then $A + \delta$ is invertible, and $||(A + \delta)^{-1}|| \leq \frac{||A^{-1}||}{1-||\delta A^{-1}||}$.*

Obtaining a Convex Subset

We now return to the main problem: find a "large" convex subset $\mathcal{C}$ such that

$$(\delta_A, \delta_c) \in \mathcal{C} \implies ||(\hat{A}_0 + \delta_A)^{-1}(\hat{c}_0 + \delta_c) - \hat{A}_0^{-1}\hat{c}_0|| \leq \epsilon.$$

Applying Lemma 6, we will hereafter assume that $||\delta_A \hat{A}_0^{-1}|| < 1$ (this assumption is not trivial, and we are actively working to mitigate it).

We start by using the triangle inequality to obtain the bound

$$||(\hat{A}_0 + \delta_A)^{-1}(\hat{c}_0 + \delta_c) - (\hat{A}_0 + \delta_A)^{-1}\hat{c}_0|| + ||(\hat{A}_0 + \delta_A)^{-1}\hat{c}_0 - \hat{A}_0^{-1}\hat{c}_0||$$

After some manipulations, and using Lemmas 1-6, this expression can be bounded by

$$\frac{||\hat{A}_0^{-1}\delta_c|| + ||\hat{A}_0^{-1}\delta_A\beta_0||}{1 - ||\hat{A}_0^{-1}\delta_A||} \tag{3.8}$$

This bound allows a simple derivation of the sought convex subset $\mathcal{C}$ as follows. First, replace it with the larger bound given in the left-hand side of

$$\frac{||\hat{A}_0^{-1}||\ ||\delta_c|| + ||\hat{A}_0^{-1}||\ ||\beta_0||\ ||\delta_A||}{1 - ||\hat{A}_0^{-1}||\ ||\delta_A||} \leq \epsilon \implies$$
$$||\hat{A}_0^{-1}||\ ||\delta_c|| + ||\hat{A}_0^{-1}||(||\beta_0|| + \epsilon)||\delta_A|| \leq \epsilon \tag{3.9}$$

and this constraint is convex. The main computational overhead at the nodes consists of computing $||\delta_A||$; this can perhaps be alleviated by using some bounds.

Better results can be obtained by using the bound in Eq. (3.8). Note that threshold constraints (upper bounds) on $||\hat{A}_0^{-1}\delta_c||$ and $||A_0^{-1}\delta_A\beta_0||$ are convex in δ_c, δ_A; also, the condition $||\delta_A\hat{A}_0^{-1}|| < 1$ is convex in δ_A. The constraint is then

$$||\hat{A}_0^{-1}\delta_c|| + ||\hat{A}_0^{-1}\delta_A\beta_0|| + \epsilon||\hat{A}_0^{-1}\delta_A|| \leq \epsilon \tag{3.10}$$

while this constraint is weaker (i.e. better) than the one in Eq. (3.9), it is computationally more demanding, requiring more matrix operations at the nodes.

3.2.3 Infinite and Sliding Window

We differentiate between two different variations for computing the global model. In the *infinite window* model the current hypothetical global model A is computed over all observations seen so far. In the *sliding window* model with width W, the global model A is computed over the last W observations only. Our approach supports both models. For the infinite window model each node i uses the bound with $\delta_{A_i} = A_i - A_i^0$ and $\delta_{c_i} = c_j - c_0$ where A_i and c_i are computed from all observations seen at the node. For the sliding window model we simply subtract the observations that left the sliding window from δ_{A_i} and δ_{c_i} (including observations used to compute $\hat{A}_0$ and $\hat{c}_0$, if needed).

3.3 Preliminary Evaluation

We evaluated communication performance of the monitoring algorithm using synthetic data. We used the infinite window variant with the simplest violation resolution protocol: global syncrhonization. Each node applies the local constraint from Eq. (3.9) to its own data. When a violation occurs at any node, it is reported to a *coordinator* node. The coordinator polls all nodes for their local data, computes a new global model b_0, and distributes it (with other associated initial values). Monitoring then resumes as normal. The main loop of each node is summarized in Alg. 1.

Algorithm 1 Node N_i main update loop with new vector x, scalar y.

$A_i \leftarrow x^T x$
$c_i \leftarrow x^T y$
$\delta_{A_i} \leftarrow A_i - A_i^0$
$\delta_{c_i} \leftarrow c_i - c_i^0$
if $\|\hat{A}_0^{-1}\|\|\delta_{ci}\| + \|\hat{A}_0^{-1}\|\left(\|\beta_0\| + \epsilon\right)\|\delta_A\| > \epsilon$ **then**
  Report violation to coordinator.
  Receive new β_0, $\hat{A}_0^{-1}$ from coordinator.
  $A_i^0 \leftarrow A_i$
  $c_i^0 \leftarrow c_i$
end if

We first simulated 1000 rounds with $k = 10$ nodes, each receiving a new vector x of size $m = 20$ and scalar y at each round. x is drawn i.i.d from $N(0, \sigma^2)$, and $y = \beta^T x + n$ where $n \sim N(0, 1)$ is gaussian white noise, $\sigma = 10$ is the strength of the signal, and β is the true model, drawn i.i.d $\beta \sim N[0, 1]$ (therefore $\|\beta\|^2 \sim \chi^2_{20}$)

We are interested in two metrics. The most important metric is communication fraction: the number of messages sent by the distributed algorithm divided by number of messages sent by a centralized algorithm (always total rounds $\times k$). The second metric is the model error metric $\|\beta_0 - \beta\|$; a correctly functioning monitoring algorithm ensures $\|\beta_0 - \beta\| \leq \epsilon$. Ideally, both communication fraction and model errors are low.

Figure 3.1 plots model error (blue) and rounds with violations (green vertical lines) for $\epsilon = 3$ (this is a fairly restrictive bound: $\Pr[\|\beta\| < 3] \cong 0.017$). We disregard the first 100 "warm-up" rounds, as we are interested in steady-state performance. Communication fraction is 0.112, and model error $\|\beta_0 - \beta\|$ is very low (below 0.005, far below ϵ), meaning that the algorithm maintains a very accurate approximation with only 11% of communication. We observe that as time passes violations become less and less frequent, as β_0 becomes a better estimate for the data. Conversely, algorithms that rely on centralization or periodic recomputation will still require fixed communication per round, regardless of the behavior of the data.

To explore how ϵ affects communication, we repeated the simulation with ϵ varying over a range. The results are shown in Figure 3.2. Again we disregard the first 100 rounds. Observe that for all $\epsilon > 0.5$ communication fraction is below 1. This ϵ is very restrictive, as $\Pr[\|\beta\| < 0.5] < 2.2 \times 10^{-16}$. Thus for almost any fixed true model β we can expect to get both communication saving and low error relative to $\|\beta\|$.

Finally, Figure 3.3 shows how monitoring behaves in the face of abrupt changes. We repeated the simulation, this time divided into 5 epochs, were β is randomized at the beginning of each epoch. As before, we disregard the first 100 rounds (half of the first epoch). We see now that error is much larger, expected the model abruptly changes every 200 rounds. Similarly, communication fraction is also larger,

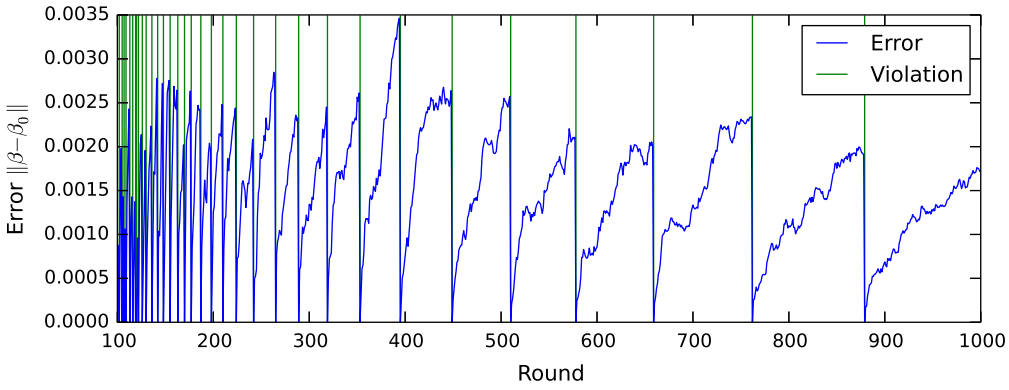

Figure 3.1: Error and violations for the infinite window case with $k = 10$ nodes, $m = 20$, $\epsilon = 3$ and $\sigma = 10$; communication fraction is 0.112.

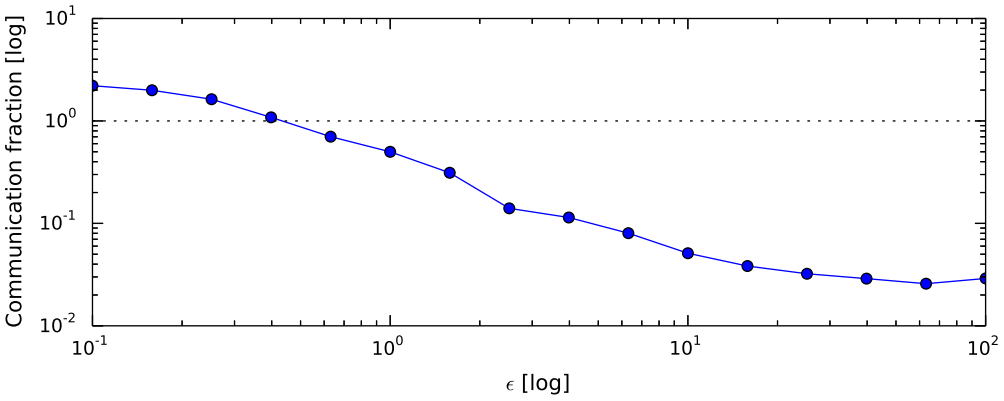

Figure 3.2: Error and violations for the infinite window case with $k = 10$ nodes, $m = 20$, $\epsilon = 3$ and $\sigma = 10$; communication fraction is 0.112.

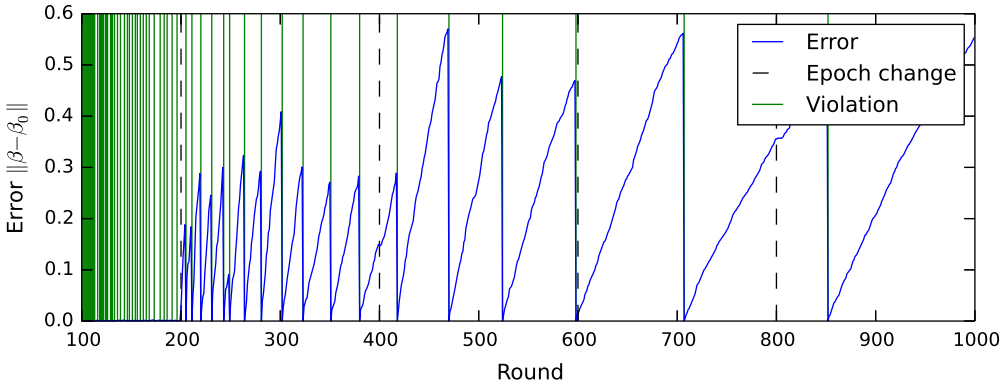

Figure 3.3: Error and violations for the infinite window case with 5 epochs. Communication fraction is 0.199.

at 0.199, since the algorithm now has to adapt more often. Despite the frequent changes, model error is still quite low, below 0.6, and communication is still one-fifth of centralized communication.

3.4 Conclusions and Future Directions

Consider the SPEEDD use case of fraud detection. Once a new fraud method appears, the previous models quickly become obsolete. We believe that the method proposed here aims at solving this problem, by monitoring model relevancy and by pointing at the need to compile a new model. In fact, this may also become relevant for traffic forecasting models, as well as for many other domains of application, as data sources become geo-distributed and ever increasingly rapid. The data from the use cases will thus be used to evaluate the proposed method.

We propose a communication-efficient monitoring algorithm for the least-squares regression models. By monitoring the deviation of the existing model from the true model, our approach is able to avoid costly communication and model computations. Each round, each node checks a simple local constraint on its own local data, and if it is satisfied, communication is avoided. If not, violation is resolved by collecting data from all nodes and computing a new global model. Note that our distributed *monitoring* approach can easily be combined with an efficient distributed *computation* technique, enjoying the best of both worlds.

There are several immediate and future steps. First, our bounds require $\|\delta_A \hat{A}_0^{-1}\| < 1$ which is not trivial; our next step is to mitigate this issue. Second, we can allocate "slack" to each node to avoid violations without the need for global syncrhonization. One possible scheme will have nodes with "good" matrix A_j (smallest eigenvalue is large) contributing more slack to nodes with more violations. In general, more sophisticated violation recovery approaches could perform local rather than global synchronization, for example by grouping nodes into a hierarchy. Finally, many more sophisticated algorithms internally rely on linear regression or an $(X^T X)^{-1}$ step. The question is therefore how to incorporate our approach.

4

Conclusions

In this document we have described initial algorithmic breakthroughs towards horizontal and vertical scalability. The first approach uses knowledge about the distribution of events which compose a target CEP. The lazy approach would first search for those events composing the complex target which rarely appear. It turned out such a lazy scheme may save lots of cycles and resources. The integration of these ideas into the SPEEDD architecture is under way, and will take the rest of the project to implement and experiment with. The data from the usecases will be used to evaluate the proposed technique, as explained above.

The second approach focus on horizontal scalability in monitoring analytical models. The ideas draw from previous and other EC projects (such as LIFT and Ferarri), but take them one step further beyond monitonring global functions. Indeed, the monitoring of sophisticated models (such as SVM) is both challenging and novel. This work will be pursued and refined in the rest of this project, as well as evaluated experimentally using appropriate evaluation data and the SPEEDD usecases data.

Bibliography

http://www.eoddata.com.

D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *In CIDR*, pages 277–289, 2005.

A. Adi and O. Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, May 2004. ISSN 1066-8888. doi: 10.1007/s00778-003-0108-y. URL http://dx.doi.org/10.1007/s00778-003-0108-y.

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*, SIGMOD '08, pages 147–160, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376634. URL http://doi.acm.org/10.1145/1376616.1376634.

M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, Aug. 2008. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=1453856.1453869.

A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006. ISSN 1066-8888. doi: 10.1007/s00778-004-0147-z. URL http://dx.doi.org/10.1007/s00778-004-0147-z.

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. doi: 10.1109/MTS.2014.2345131. URL http://dx.doi.org/10.1109/MTS.2014.2345131.

H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, Dec. 2004. ISSN 1066-8888. doi: 10.1007/s00778-004-0133-5. URL http://dx.doi.org/10.1007/s00778-004-0133-5.

M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1), 2008. URL http://dblp.uni-trier.de/db/journals/tods/tods33.html#BalazinskaBMS08.

R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, pages 363–374. www.cidrdb.org, 2007. URL http://dblp.uni-trier.de/db/conf/cidr/cidr2007.html#BargaGAH07.

M. Boley, M. Kamp, D. Keren, A. Schuster, and I. Sharfman. Communication-efficient distributed online prediction using dynamic model synchronizations. In *Proceedings of the First International Workshop on Big Dynamic Distributed Data, Riva del Garda, Italy, August 30, 2013*, pages 13–18, 2013. URL http://ceur-ws.org/Vol-1018/paper6.pdf.

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*, SIGMOD '07, pages 1100–1102, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-686-8. doi: 10.1145/1247480.1247620. URL http://doi.acm.org/10.1145/1247480.1247620.

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. URL http://dblp.uni-trier.de/db/conf/cidr/cidr2003.html#ChandrasekaranDFHHKMRRS03.

J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. *SIGMOD Rec.*, 29(2):379–390, May 2000. ISSN 0163-5808. doi: 10.1145/335191.335432. URL http://doi.acm.org/10.1145/335191.335432.

G. Cugola and A. Margara. Tesla: a formally defined event specification language. In J. Bacon, P. R. Pietzuch, J. Sventek, and U. Çetintemel, editors, *DEBS*, pages 50–61. ACM, 2010. ISBN 978-1-60558-927-5.

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, June 2012. ISSN 0360-0300. doi: 10.1145/2187671.2187677. URL http://doi.acm.org/10.1145/2187671.2187677.

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*, EDBT'06, pages 627–644, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-32960-9, 978-3-540-32960-2. doi: 10.1007/11687238_38. URL http://dx.doi.org/10.1007/11687238_38.

A. Demers, J. Gehrke, and B. P. Cayuga: A general purpose event monitoring system. In *In CIDR*, pages 412–422, 2007.

L. Ding, S. Chen, E. A. Rundensteiner, J. Tatemura, W.-P. Hsiung, and K. S. Candan. Runtime semantic query optimization for event stream processing. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 0:676–685, 2008. doi: http://doi.ieeecomputersociety.org/10.1109/ICDE.2008.4497476.

O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010. ISBN 1935182218, 9781935182214.

A. Friedman, I. Sharfman, D. Keren, and A. Schuster. Privacy-preserving distributed stream monitoring. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2013*, 2014. URL http://www.internetsociety.org/doc/privacy-preserving-distributed-stream-monitoring.

M. Gabel, A. Schuster, and D. Keren. Communication-efficient distributed variance monitoring and outlier detection for multivariate time series. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 37–47, 2014. doi: 10.1109/IPDPS.2014.16. URL http://dx.doi.org/10.1109/IPDPS.2014.16.

B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In J. T.-L. Wang, editor, *SIGMOD Conference*, pages 1123–1134. ACM, 2008. ISBN 978-1-60558-102-6.

N. Giatrakos, A. Deligiannakis, M. N. Garofalakis, I. Sharfman, and A. Schuster. Distributed geometric query monitoring using prediction models. *ACM Trans. Database Syst.*, 39(2):16, 2014. doi: 10.1145/2602137. URL http://doi.acm.org/10.1145/2602137.

T. S. Group. Stream: The stanford stream data manager. Technical Report 2003-21, Stanford InfoLab, 2003. URL http://ilpubs.stanford.edu:8090/583/.

X. Gu, P. S. Yu, and H. Wang. Adaptive load diffusion for multiway windowed stream joins. In R. Chirkova, A. Dogac, M. T. ?zsu, and T. K. Sellis, editors, *ICDE*, pages 146–155. IEEE, 2007. URL http://dblp.uni-trier.de/db/conf/icde/icde2007.html#GuYW07.

D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman. On supporting kleene closure over event streams. In G. Alonso, J. A. Blakeley, and A. L. P. Chen, editors, *ICDE*, pages 1391–1393. IEEE, 2008. URL http://dblp.uni-trier.de/db/conf/icde/icde2008.html#GyllstromADI08.

L. Harada and Y. Hotta. Order checking in a cpoe using event analyzer. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, CIKM '05, pages 549–555, New York, NY, USA, 2005. ACM. ISBN 1-59593-140-6. doi: 10.1145/1099554.1099700. URL http://doi.acm.org/10.1145/1099554.1099700.

M. Hirzel. Partition and compose: Parallel complex event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 191–200, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1315-5. doi: 10.1145/2335484.2335506. URL http://doi.acm.org/10.1145/2335484.2335506.

M. Kamp, M. Boley, D. Keren, A. Schuster, and I. Sharfman. Communication-efficient distributed online prediction by dynamic model synchronization. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2014, Nancy, France, September 15-19, 2014. Proceedings, Part I*, pages 623–639, 2014. doi: 10.1007/978-3-662-44848-9_40. URL http://dx.doi.org/10.1007/978-3-662-44848-9_40.

D. Keren, I. Sharfman, A. Schuster, and A. Livne. Shape sensitive geometric monitoring. *IEEE Trans. Knowl. Data Eng.*, 24(8):1520–1535, 2012. doi: 10.1109/TKDE.2011.102. URL http://doi.ieeecomputersociety.org/10.1109/TKDE.2011.102.

D. Keren, G. Sagy, A. Abboud, D. Ben-David, A. Schuster, I. Sharfman, and A. Deligiannakis. Geometric monitoring of heterogeneous streams. *IEEE Trans. Knowl. Data Eng.*, 26(8):1890–1903,

2014. doi: 10.1109/TKDE.2013.180. URL http://doi.ieeecomputersociety.org/10.1109/TKDE.2013.180.

G. T. Lakshmanan, Y. G. Rabinovich, and O. Etzion. A stratified approach for supporting high throughput event processing applications. In *DEBS*, 2009.

M. Liu, E. A. Rundensteiner, D. J. Dougherty, C. Gupta, S. Wang, I. Ari, and A. Mehta. Neel: The nested complex event language for real-time event analytics. In M. Castellanos, U. Dayal, and V. Markl, editors, *BIRTE*, volume 84 of *Lecture Notes in Business Information Processing*, pages 116–132. Springer, 2010. ISBN 978-3-642-22969-5. URL http://dblp.uni-trier.de/db/conf/birte/birte2010.html#LiuRDGWAM10.

A. S. M. L. M Silberstein, D Geiger. Scheduling mixed workloads in multi-grids: the grid execution hierarchy. In *High Performance Distributed Computing*, pages 291–302, 2006.

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.

M. R. Mendes, P. Bizarro, and P. Marques. Fincos: Benchmark tools for event processing systems.

M. G. I. S. A. S. N Giatrakos, A Deligiannakis. Distributed geometric query monitoring using prediction models. In *ACM Transactions on Database Systems (TODS)*, volume 39 (2), 16, 2014.

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, June 2004. ISSN 0362-5915. doi: 10.1145/1005566.1005568. URL http://doi.acm.org/10.1145/1005566.1005568.

G. Sagy, D. Keren, I. Sharfman, and A. Schuster. Distributed threshold querying of general functions by a difference of monotonic representation. *PVLDB*, 4(2):46–57, 2010. URL http://www.vldb.org/pvldb/vol4/p46-sagy.pdf.

N. P. Schultz-Møller, M. Migliavacca, and P. R. Pietzuch. Distributed complex event processing with query rewriting. In *DEBS*, 2009.

I. Sharfman, A. Schuster, and D. Keren. Aggregate threshold queries in sensor networks. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–10, 2007a. doi: 10.1109/IPDPS.2007.370297. URL http://dx.doi.org/10.1109/IPDPS.2007.370297.

I. Sharfman, A. Schuster, and D. Keren. A geometric approach to monitoring threshold functions over distributed data streams. *ACM Trans. Database Syst.*, 32(4), 2007b. doi: 10.1145/1292609.1292613. URL http://doi.acm.org/10.1145/1292609.1292613.

U. Verner, A. Mendelson, and A. Schuster. Scheduling periodic real-time communication in multi-gpu systems. In *23rd International Conference on Computer Communication and Networks, ICCCN 2014, Shanghai, China, August 4-7, 2014*, pages 1–8, 2014a. doi: 10.1109/ICCCN.2014.6911778. URL http://dx.doi.org/10.1109/ICCCN.2014.6911778.

U. Verner, A. Mendelson, and A. Schuster. Batch method for efficient resource sharing in real-time multi-gpu systems. In *Distributed Computing and Networking - 15th International Conference, ICDCN 2014, Coimbatore, India, January 4-7, 2014. Proceedings*, pages 347–362, 2014b. doi: 10.1007/978-3-642-45249-9_23. URL http://dx.doi.org/10.1007/978-3-642-45249-9_23.

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. ISBN 1-59593-154-6. URL http://dl.acm.org/citation.cfm?id=1083592.1083723.

E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors, *SIGMOD Conference*, pages 407–418. ACM, 2006. ISBN 1-59593-256-9.