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

Project	**FP7-619435 / SPEEDD**
Deliverable	**D6.4**
Distribution	**Public**

http://speedd-project.eu/

Computation and Communication Scalable Algorithms II

Daniel Keren, Mickey Gabel, Ilya Kolchinsky, Assaf Schuster

Status: Final (Version 0.2)

December 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	Alina Lupu

Deliverable

Deliverabe type	report
Distribution level	Public
Deliverable Number	D6.4
Deliverable title	Computation and Communication Scalable Algorithms II
Contractual date of delivery	M23 (December 2015)
Actual date of delivery	December 2015
Relevant Task(s)	WP6Tasks T1 & T2
Partner Responsible	TI
Other contributors	
Number of pages	51
Author(s)	Daniel Keren, Mickey Gabel, Ilya Kolchinsky, 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. This is the 2nd year version which includes the final versions as accepted for publication at highly prestigeous conferences.

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.

Findings show very high potential for the proposed approaches. Overheads in experiments are reduced dramatically, sometimes by orders of magnitude.

The work will proceed in focussing on implementation and integration of the proposed algorithmic breakthroughs, with SPEEDD data sources and use cases.

1 Introduction

1.1 History of the Document

Version	Date	Author	Change Description
0.1	1/11/2015	Assaf Schuster (TI)	Set up of the document
0.3	2/11/2015	Ilya Kolchinsky (TI)	preliminary version of CEP paradigm
0.6	23/11/2015	Mickey Gabel (TI)	preliminary version of distributed model monitoring
0.9	24/11/2015	Assaf Schuster (TI)	Content integration
1.0	7/12/2015	Assaf Schuster (TI)	Fixing remarks by internal reviewer

1.2 Purpose and Scope of the Document

The purpose of this document is to outline 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 or traffic monitoring operate on millions of daily transactions or measurements.

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 the last stage of the project these contributions will be implemented and integrated into actual working 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. This new approach was presented at DEBS 2015 and received the Best Research Paper Award.

The second novel algorithmic approach allows to check the validity of known models in a distributed manner 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 this 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. The method was presented at KDD 2015.

1.3 Relationship with Other Documents

This report is a continuation of Deliverable 6.2 in which preliminary version of the algorithmic breakthroughs were reported. Thus, this report contains more accurate, more formal, and more detailed version of the algorithms whose preliminary description was given in 6.2. Furthermore, the evaluation and experimentation of the algorithms, as well as their publication, was finalized during the 2nd year of the project and are reported in this document.

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

We are also working on the implementation of the horizontal scalability and its integration with SPEEDD traffic data. Again, this implementation work will be reported in later deliverables.

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

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

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)] and click stream analysis [Sadri et al. (2004)]. 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, omitted from the notation for brevity's sake). 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 (NFAs), which are in turn used by the event processing engine. Wu et al. [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 constituting the match prefix. Gyllstrom et al. [Gyllstrom et al. (2008)] propose additional operators for SASE, such as iterations and aggregates. Demers et al. [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. 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 NFAs in clustered environments.

An NFA detects sequences by maintaining at every point in time all the observed prefixes of the sequence until a match is detected. As an example, consider the following stream of events:

$$a^1_{p=3}, a^2_{p=5}, a^3_{p=8}, b^1_{p=7}, b^2_{p=13}, c^1_{p=9}.$$

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]

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

In this document we propose a new NFA based matching mechanism that overcomes this drawback. The proposed mechanism constructs partial matches starting from the most selective (i.e., least frequent) event, rather than from the first event in the sequence. In addition, partial matches are extended by adding events in descending order of selectivity (rather than according to their order in the sequence). This not

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

only minimizes the number of partial matches held in memory, but also reduces computation time, since there are fewer partial matches to extend when processing a given event.

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

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

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

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

The remainder of this part of the document is organized as follows. Section 2.2 describes related work. Section 2.3 briefly describes the eager NFA evaluation framework. It also provides the terminology and notations used throughout the work. In Section 2.4 we introduce the concepts and ideas of lazy evaluation, accompanied by intuitive explanations and examples. Formal definitions presented there prepare the ground for the rest of the document. In Section 2.5 we proceed to describe how a lazy *chain NFA* can be constructed using given frequencies of the participating events. We present a lazy *tree NFA* in Section 2.6. Section 2.7 contains the experimental evaluation.

2.2 Related Work

The detection of complex events over streams has become 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. Those systems are based on SQL-like specification languages and focus on processing data coming from continuous, usually multiple input streams. Examples include NiagaraCQ [Chen et al. (2000)], TelegraphCQ [Chandrasekaran et al. (2003)] and STREAM [Group (2003)]. Later, the need to analyze event notifications of interesting situations – as opposed to generic data – was identified. Then, *complex event processing systems* were introduced. One example of an advanced CEP system is Amit [Adi and Etzion (2004)], based on a strongly expressive detection language and 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 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 methods.

Apart from the SASE language, on which our mechanism is based, 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. 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 declarative language used by the 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 declarative language, attempting to combine high expressiveness with a relatively small set of operators, achieving compactness and simplicity. Even though our work focuses exclusively on sequence patterns, extensions to other operators are possible, including those added by the aforementioned languages.

Unlike most recently proposed CEP systems, which use non-deterministic finite automata (NFAs) to detect patterns, ZStream [Mei and Madden (2009)] uses tree-based query plans for the representation of query patterns. The careful design of the underlying infrastructure and algorithms makes it possible for ZStream to unify the representation of sequence, conjunction, disjunction, negation, and Kleene closure as variants of the join operator. While some of the ideas discussed in this work are close to ours, it is not based on state automata and employs matching trees instead.

Several works mention the concept of lazy evaluation in the context of event processing. In [Akdere et al. (2008)], the authors describe "plan-based evaluation," where, similarly to our work, temporal properties of primitive events can be exploited to reduce network communication costs. The focus of their paper is on communication efficiency, whereas our goal is to reduce computational and memory requirements. [Dousson and Maigat (2007)] discusses a mechanism similar to ours, including the concept of buffering incoming events into an intermediate storage. However, the authors only consider a setting in which the frequencies of primitive events are known in advance and do not change. An optimization method based on postponing redundant operations was proposed by [Zhang et al. (2014a)]. This work focuses on optimizing Reuse Consumption Policy queries by dividing evaluation into a shared part (pattern construction) and a per-instance part (result construction). The main goal of the authors is to improve the performance of Kleene closure patterns and solve the problem of imprecise timestamps. In comparison, our work focuses solely on sequence pattern matching.

The concept of lazy evaluation has also been proposed in the related research field of online processing of XML streams. [Chan et al. (2002)] describes an XPath-based mechanism for filtering XML documents in stream environments. This mechanism postpones costly operations as long as possible. However, the goal in this setting is only to detect the presence or absence of a match, whereas our focus is on finding all possible matches between primitive events. In [Green et al. (2004)], a technique for lazy construction of a DFA (Deterministic Finite Automaton) on-the-fly is discussed. This work is motivated by the problem of exponential growth of automata for XPath pattern matching. Our work solves a different problem of minimizing the number of runtime NFA instances rather that the size of the automaton itself. In addition, while there is some overlap in the semantics of CEP and XPath queries, they were designed for different purposes and allow different types of patterns to be defined.

2.3 Eager Evaluation

In this section we present a subset of the SASE language for defining sequence patterns. SASE itself is thoroughly discussed in [Agrawal et al. (2008)]. We formally describe the eager NFA matching mechanism, how a given sequence is compiled into an NFA, and how this NFA is used at runtime to detect the pattern. Here we also introduce the notations and terminology to be used in later sections.

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. As described in Section 2.3.2, patterns expressed in these languages will be compiled into a state machine for use by the detection mechanism.

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

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

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

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

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

As an example, consider the pattern presented in Example 1. There is a single event type, which we will denote by *E*. This event type has two data attributes: a categorical attribute called "ticker," which represents the stock for which the event has occurred, and a numerical attribute called "price," which is the price of the stock. Assuming the stocks *a*, *b*, and *c* are MSFT, GOOG and AAPL respectively, this pattern can be declared in SASE, as depicted in Figure 2.1.

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

Figure 2.1: SASE specification of a pattern from Example 1

2.3.2 The Eager Evaluation Mechanism

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

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

where:

- Q is a set of states;
- E is a set of directed edges, which can be of several types, as described below;

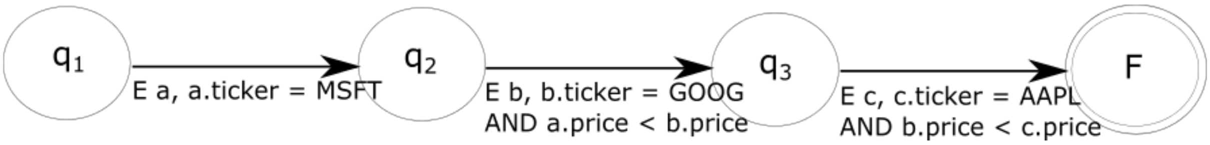


Figure 2.2: NFA for Example 1

- q_1 is an initial state;
- F is a final accepting state.

An edge is defined by the following tuple:

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

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

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

The *action* associated with an edge is performed when the edge is traversed. It can be one of the following (the actions listed below are simplified versions of the ones defined for SASE [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 discards it instead of storing it in any kind of buffer).

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

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

Figure 2.2 illustrates the NFA compiled for the pattern in Figure 2.1. Note that the final state can only be reached by executing three *take* actions; hence, successful evaluation will produce a match buffer containing three primitive events comprising the detected match.

The match buffer should be thought of as a logical construct. As discussed by Agrawal et al. [Agrawal et al. (2008)], there is no need to allocate dedicated memory for each match buffer, since multiple match buffers can be stored in a compact manner that takes into account that certain events may be included in many buffers.

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

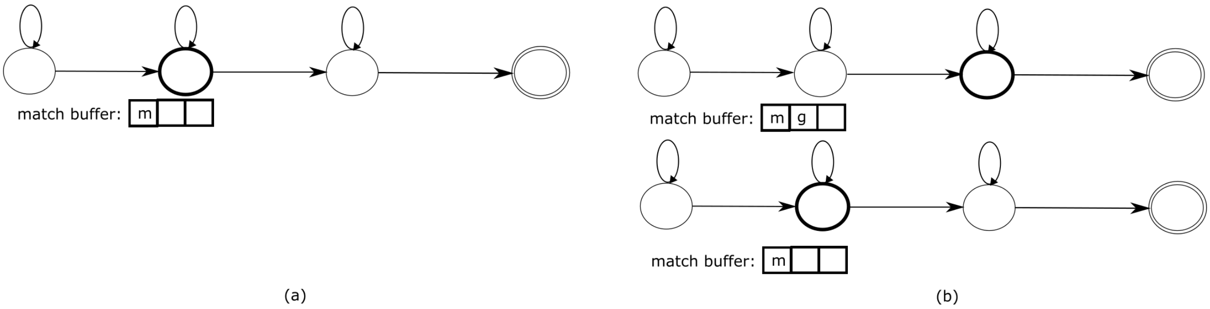


Figure 2.3: Non-deterministic evaluation of NFA for Example 1. (a) The sole NFA instance is currently at the second evaluation stage, with a single event in its match buffer. (b) A new event g arrives, and now the NFA instance can either (1) accept the new event as a part of the potential match and proceed to the next step, or (2) ignore it (by traversing a self loop) and keep waiting for a future event of the same name. The problem is solved by duplicating the instance and applying both moves.

edges, the instance will be replicated $n-1$ times. On each of the resulting n instances a different edge will be traversed. As an example, consider the situation described in Figure 2.3. In 2.3a, there is some instance of an NFA from Figure 2.2 with an event *m* in its match buffer, currently in state q_2 (we mark the current state of an instance with bold border). In 2.3b, an event $g, g.ticker = GOOG$ has arrived. This event triggers the traversal of two edges, namely the outgoing *take* edge and the outgoing *ignore* edge. As a result, one new instance will be created to allow both traversals to occur.

Eager Sequence NFA Structure

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

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

The self-loops for all event names have an *ignore* action. The edge leading from the k^{th} state to the next one has a *take* action with the event name of the k^{th} event in the sequence. The purpose of the self-loops is to allow detection of all possible combinations of events. This is achieved by exploiting non-deterministic behavior as illustrated by Figure 2.3.

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

1. For each clause of the CNF, let i denote the index of the latest primitive event it contains (in the specified order of appearance in the pattern).

2. The condition on the edge connecting the i^{th} state with the following state is a conjunction of all the CNF clauses with the index i and the timing predicate.

For example, consider constructing a sequence NFA for the pattern in Figure 2.1. The edge from q_1 to q_2 will only contain a part of the global condition on *a*, the next edge will specify the constraint on *b* and the mutual constraint on *a* and *b*, and, finally, the final edge towards the accepting state will validate the constraint on *c* and the mutual constraint on *c* and *b*.

Figure 2.2 demonstrates the result of applying the construction process described above on the pattern in Figure 2.1.

Runtime Behavior

As described above, the pattern detection mechanism consists of multiple NFA instances running simultaneously, where each instance represents a partial match. Each NFA instance contains the current state and a match buffer. Upon startup, the system creates a single instance with an empty match buffer, whose current state is the initial state. Every event received on the input stream will be applied to all NFA instances. If the timing predicate is not satisfied on a given instance (i.e., the earliest event in the match buffer is not within the allowed time interval), the instance and the associated match buffer will be discarded. Otherwise, an event will either cause a single edge traversal on an unconditional *ignore* edge, or also an additional traversal on a *take* edge. In the former case the event will be ignored. In the latter case the instance will be duplicated, and both possible traversals will be executed on different copies.

2.4 Lazy Evaluation

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

First, we will demonstrate the need for such a mechanism and show its effectiveness using the continuation of Example 1. Consider a scenario where on a certain day primitive events corresponding to *a* and *b* (MSFT and GOOG respectively) are very frequent, while events corresponding to *c* (AAPL) are relatively rare. More specifically, assume that within a time window *t* we receive 100 instances of MSFT stock events, denoted $a^1_{p_1}$,...,$a^{100}_{p_{100}}$, followed by 100 instances of GOOG stock events, denoted $b^1_{p_{101}}$,...,$b^{100}_{p_{200}}$, followed by a single instance of an 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 substantially reduce the number of evaluations if we defer the match detection process until the single event for AAPL has arrived, then pair it with appropriate GOOG events, and finally check which of these pairs match a MSFT event. In this case we need to perform 100 checks of *b.price <c.price*, and an additional 100 checks of *a.price <b.price,* resulting in a total of 200 evaluations in comparison to at least 10,000 evaluations in the eager strategy. In addition, note that at every point in time, we hold a single partial match, as opposed to the eager mechanism, which may hold up to 10,000 partial matches.

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

The idea behind lazy evaluation is to enable instances to store incoming events, and if necessary, retrieve them later for processing. To support this, an additional buffer, referred to as the *input buffer,* is associated with each NFA instance, and an additional action, referred to as *store,* is defined. When an edge with a *store* action is traversed, the event causing the traversal is inserted into the input buffer.

The input buffer stores events in chronological order. Those events can then be accessed during later evaluation steps, using a modification on the *take* edge that we will define shortly.

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

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

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

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

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

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

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

precede b, not all b events located in the input buffer are to be returned and evaluated, but only those succeeding the accepted a instance and preceding the accepted c instance located in the match buffer.

Figure 2.4 illustrates the two examples above.

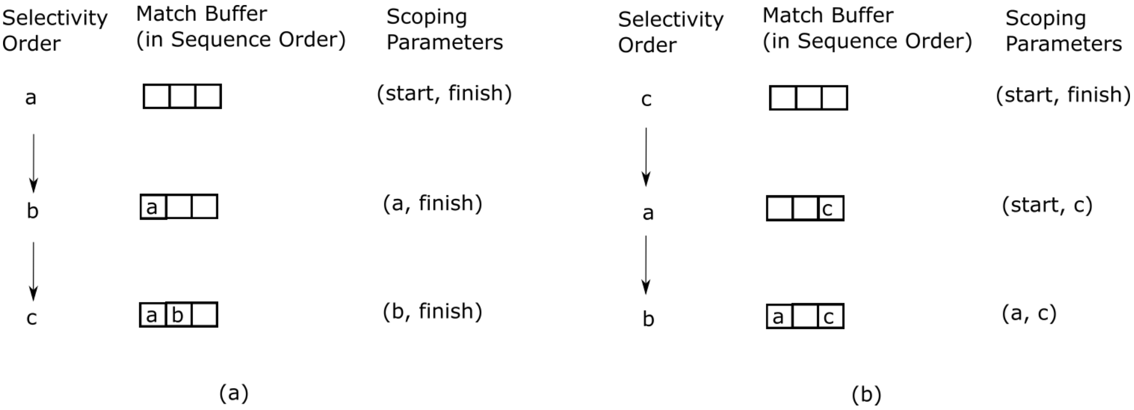


Figure 2.4: Scoping Parameters Example

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

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

The parameter s can accept one of the following values:

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

The parameter f can accept one of the following values:

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

We will demonstrate the definitions above on examples from the beginning of the section, illustrated also in Figure 2.4.

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

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

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

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

2.5 Chain NFA

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

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

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

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

- $e_i^{ignore} = (q_i, q_i, ignore, Prec_{sel}\,(e_i)\,, true)$: any event whose name corresponds to one of the already taken events is ignored.
- $e_i^{store} = (q_i, q_i, store, Succ_{sel}\,(e_i)\,, true)$: any event that might be taken in one of the following states is stored in the input buffer.
- $e_i^{take} = (q_i, q_{i+1}, take, e_i, cond_i \wedge InScope_i)$: an event with the name e_i is taken only if it satisfies the conditions required by the initial pattern (denoted by $cond_i$) and is located inside the scope defined for this edge (denoted by a predicate $InScope_i$).

The chain NFA will thus be defined as follows:

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

where:

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

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

Figure 2.5 demonstrates the chain NFA for the pattern shown in Figure 2.1. For simplicity, *ignore* edges are omitted, as are $InScope_i$ predicates.

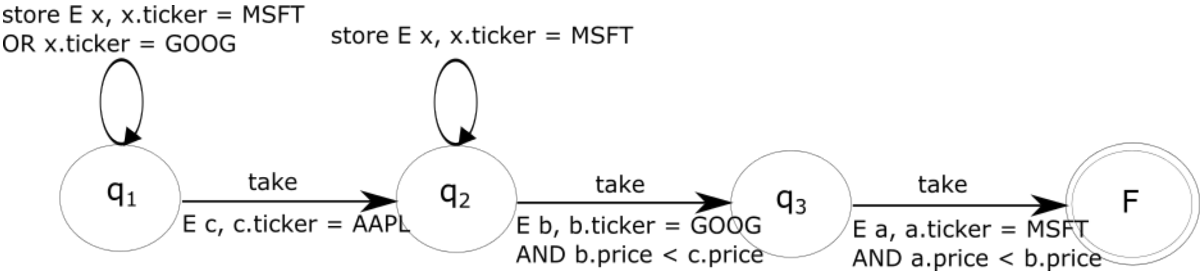


Figure 2.5: Chain NFA for Example 1

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

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

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

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

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

2.6 Tree NFA

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

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

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

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

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

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

Figure 2.6 illustrates a tree NFA for the pattern in Figure 2.1. For simplicity, ignore edges are omitted.

In formal terms, a tree NFA is structured as a tree of depth $n-1$, the root being the initial state and the leaves connected to the accepting state. Nodes located at each layer k; $0 \leq k \leq n-1$ (i.e., all nodes in depth k) are all states responsible for all orderings of k event names out of the n event names defined in the sequence. Each such node has $n-k$ outgoing edges, one for each event name which does not yet appear in the partial ordering this node is responsible for. Those edges are connected to states at the next layer, responsible for all extensions of the ordering of this particular node to length of $k+1$. The only exceptions to this rule are the leaves, which have a single outgoing edge, connected directly to the final state.

For instance, in the example in Figure 2.6, layer 0 contains the initial state q_0, layer 1 contains states q_1, q_2, q_3, and layer 2 contains the states $q_{12}, q_{13}, q_{21}, q_{23}, q_{31}, q_{32}$.

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

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

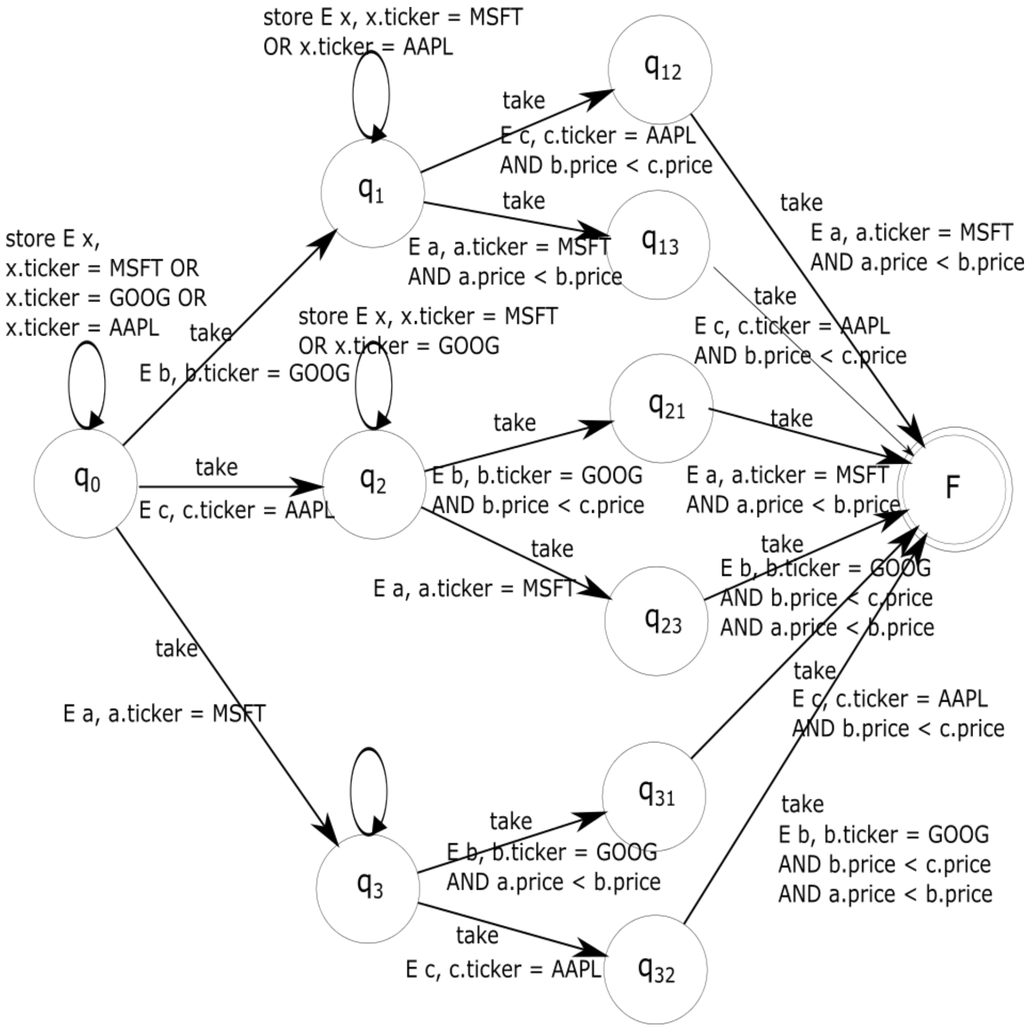


Figure 2.6: Tree NFA for Example 1

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

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

$$q_0 = q_{\emptyset}.$$

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

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

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

Let c_e denote the value of the counter of events associated with the name e in the input buffer. Let $se(q_{ord}) = min(\{c_e | e \notin ord\})$ denote the most selective (i.e., most infrequent) event in the input buffer during the evaluation step in which q_{ord} is the current state. Finally, we will define the predicate $p_{ne}\left(q_{ord}\right)$ (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:

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

The set of all edges for tree 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.

It can be observed that a tree NFA contains all the possible chain NFAs for a given sequence pattern, with shared states for common prefixes. Thus, the execution of a tree NFA on any input is equivalent to the execution of some chain NFA on that input. The conditions on tree NFA edges are designed in such a way that the most selective event is chosen at each evaluation step. Hence, this chain NFA is always the one whose given selectivity order is the actual selectivity order as observed from the input stream. An example can be seen in Figure 2.6. Nodes and edges marked in bold illustrate the evaluation path for an input stream satisfying $count(AAPL) \leq count(GOOG) \leq count(MSFT)$, i.e., corresponding to the selectivity order *c,b,a*.

The scoping parameters for a tree NFA are calculated the same way as for a chain NFA, as described in Section 2.5.[2]

[2] Contrary to the chain NFA, the tree NFA does not have a predefined selectivity order *sel* to be used for calculating the scoping parameters. Instead, for an edge $e^{take}_{ord,e}$ we will substitute *sel* with the partial order *ord*. This order is the effective selectivity order applied on the current input.

2.6.1 Implementation Issues

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

2.7 Experimental Evaluation

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

As a measure of runtime complexity, we counted how many times a condition on an edge is evaluated. For instance, consider the pattern from Example 1 and two successive streams of events: $a^1_{p=3}$,$b^1_{p=7}$,$c^1_{p=9}$ 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.

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

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

All experiments were run on a HP 2.53 Ghz CPU and 8.0 GB RAM. We used the real-world historical data of stock prices from the NASDAQ stock market, taken from [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 has the following attributes: stock identifier (ticker), timestamp, and current price. We also assumed that each event has an attribute specifying to which sector the stock belongs, e.g., hi-tech, finance or pharmaceuticals.

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

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

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

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

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

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

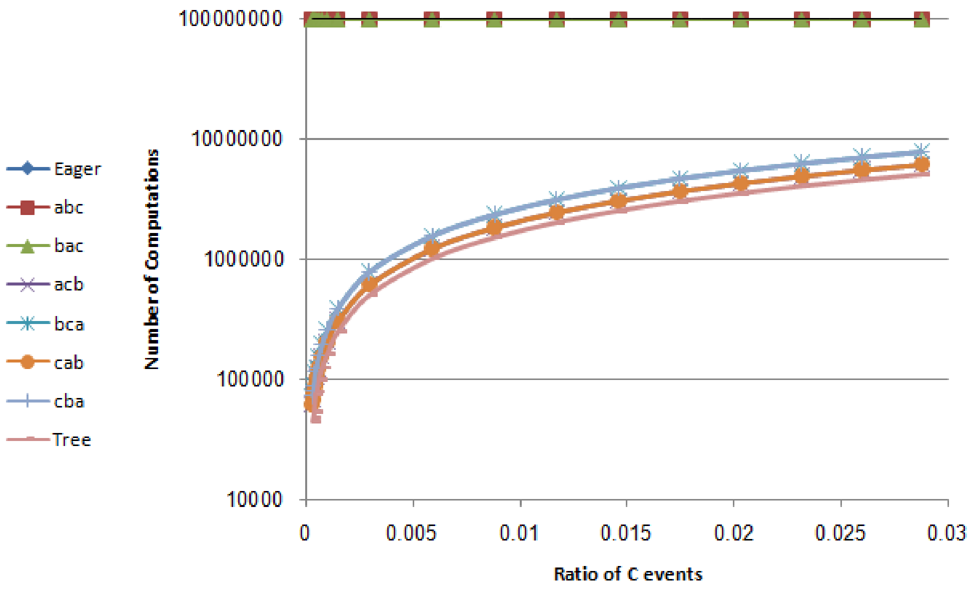


Figure 2.7: Comparison of NFAs by number of operations (logarithmic scale) for sequence *a,b,c*

Figure 2.7 describes the number of computations performed by each NFA as a function of f_c. The following observations can be made:

1. Eager NFA shows the same, very poor performance for any value of f_c.
2. Lazy chain NFAs constructed with *c* as the second or the third event (namely *abc*, *bac*, *acb* and *bca*) display equally suboptimal performance because detecting the pattern using these orders implies creation and manipulation of large numbers of NFA instances, just as with eager NFA.
3. Lazy chain NFAs constructed with *c* as a first event, namely *cba* and *cab*, perform one to three orders of magnitude better. This exactly matches our expectations, as starting the evaluation process only when the rarest event arrives allows us to significantly reduce the number of instances, and hence the number of calculations.

4. Tree NFA demonstrates slightly better performance than that of the best chain NFA (*cab* in our case). This minor improvement is due to the changes in the relative frequencies of *a* and *b* events, to which tree NFA was able to adapt as a result of its dynamic structure.

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

In our next experiment we evaluated patterns with the most selective event *c* placed at the beginning or in the middle, producing the target sequences *c,b,a* and *a,c,b*. We used the same set of conditions as in the previous experiment. The results of the performance evaluation of the system when invoked on those patterns are shown in Figure 2.8. The main observation is that the performance of any lazy NFA is independent of the sequence order, as selectivity orders ending with *c* will always perform poorly, whereas those starting with *c* will show better results. The only notable difference is the performance of eager NFA, which significantly improves on the *c,b,a* pattern. The reason is that in this case the sequence order is also the most efficient selectivity order. It can be seen that, for any pattern, the tree NFA remains superior.

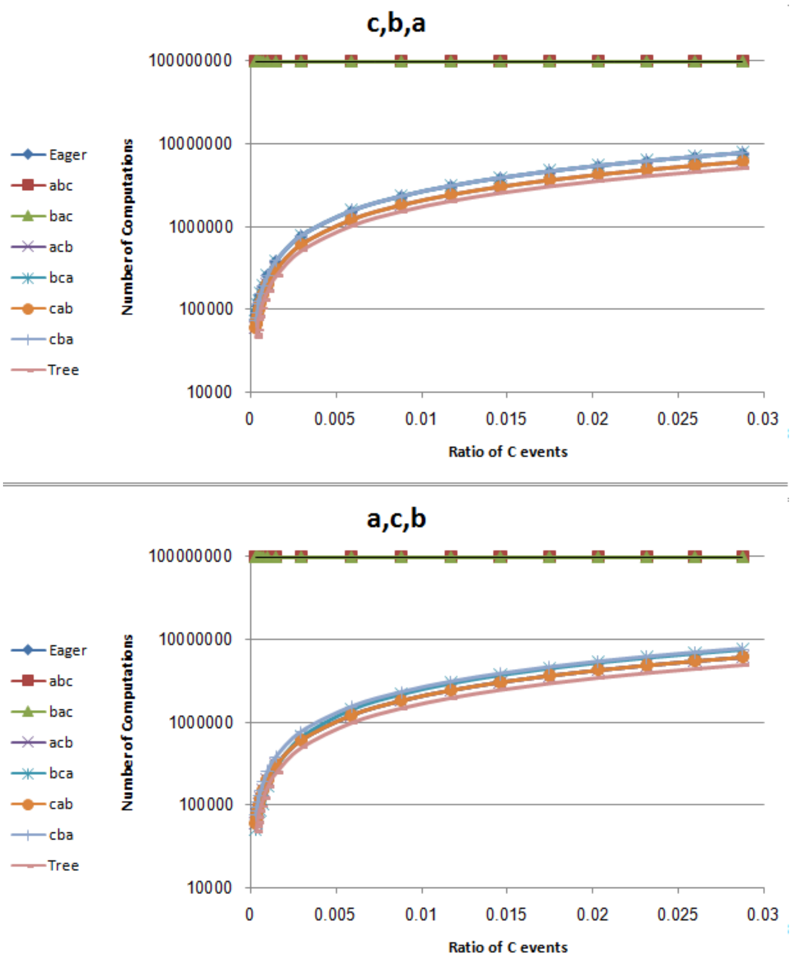


Figure 2.8: Comparison of NFAs by number of operations (logarithmic scale) for sequences *c,b,a* and *a,c,b*

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

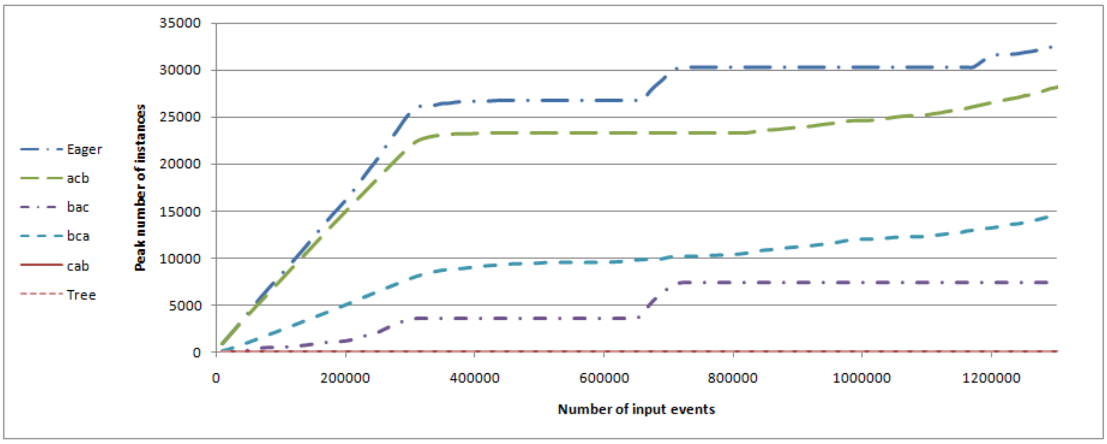


Figure 2.9: Comparison of NFAs by memory consumption for sequence *a,b,c*

Figure 2.9 demonstrates the peak number of instances generated by the different types of NFAs discussed above when detecting the sequence *a,b,c* on inputs of various sizes. Only some of the chain NFA graphs are shown. Other automata produced outputs very similar to one of the displayed ones and were omitted for the sake of clarity. It can be observed that:

1. Lazy chain NFAs with *c* as a first event require memory for a smaller number of instances than the other NFAs. This is because evaluation in these automata occurs only upon arrival of a *c* event, at which point the whole match is already located in the input buffer. Hence, there is no need to wait for additional input from the stream and evaluation ends almost immediately in most cases.

2. Lazy chain NFAs corresponding to selectivity orders starting with *a* consume significantly more memory, which is comparable to the memory consumed by the eager NFA. As the previous graph shows, NFAs based on those orders use many instances simultaneously. The number of such instances is proportional to that of eager NFA; hence, they use approximately equivalent memory in terms of NFA instances.

3. Lazy chain NFAs corresponding to selectivity orders starting with *b* display better, yet still do not achieve optimal memory utilization due to selectivity of mutual conditions between *a* and *b*.

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

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

Figure 2.10 demonstrates the results. As in the previous graph, some of the chain NFAs were omitted due to very similar results. The x-axis represents the number of events from the beginning of the stream. It can be thought of as the closest estimate to the time axis. The y-axis represents the number of computations per 10,000 events.

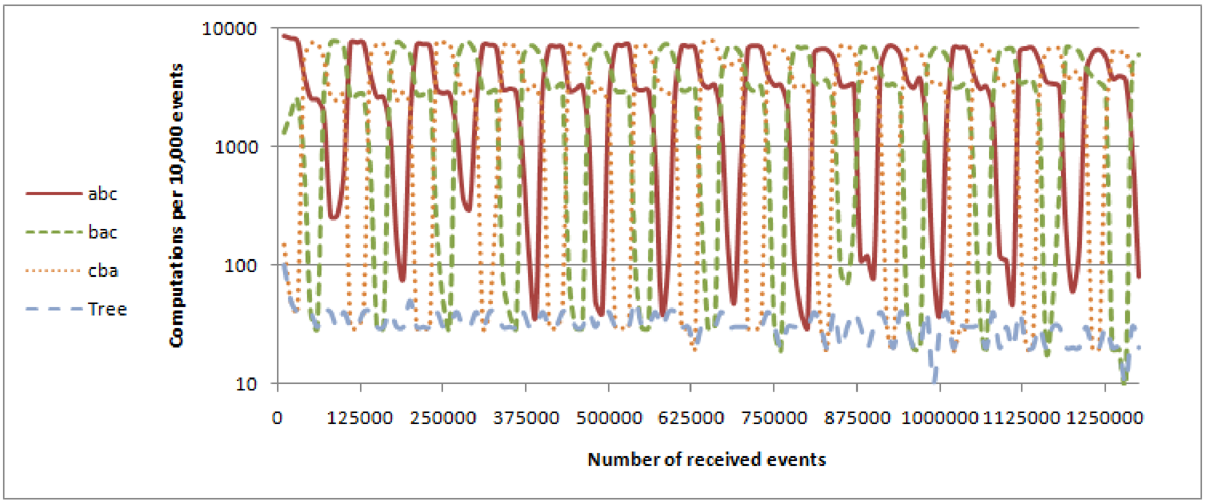


Figure 2.10: Comparison of NFAs by number of operations on highly dynamic input (logarithmic scale) for sequence *a,b,c*

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

2.8 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 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. As will be descrived below, experiments on SPEEDD data show dramatic reduction in communication volume (hence, improvement in scalability).

3.2 Problem Definition

Let $\{(x_1, y_1), \dots, (x_n, y_n)\}$ be a set of n observation pairs of $m < n$ independent variables and one dependent variable, where x_i are column vectors in $\mathbb{R}^m$, and y_i are the corresponding response scalars. We seek a linear transformation $\beta \in \mathbb{R}^m$, $\beta = (\beta_1, \dots, \beta_m)^T$, that minimizes the sum of squared errors between y_i to the mapping of x_i. In other words, we seek a model β that minimizes $\|X\beta - y\|^2$, where X is the $n \times m$ matrix of row vectors $X \triangleq (x_1^T, \dots, x_n^T)^T$, and y is the column vector composed of response scalars $y \triangleq (y_1, \dots, y_n)^T$.

The optimal solution to this convex problem, known as *ordinary least squares* (OLS), is given by Hayashi (2000)

$$\beta = \left(X^T X\right)^{-1} X^T y \quad . \tag{3.1}$$

3.2.1 Monitoring OLS of Distributed Streams

Assume that the observations $\{(x_i, y_i)\}$ are distributed across k nodes, and that these observations are dynamic – they change over time, as nodes receive new observations that replace older ones. As data evolves, it is possible that the previously computed model no longer matches the current true model. We wish to maintain an accurate estimation β_0 of the current global OLS model, β. The question is then when to update the model.

The simplest way is to update β every time a new observation arrives at the nodes, using a straightforward or incremental procedure. Though this gives the most accurate model, it is also wasteful. It requires communicating the update every time, and potentially disseminating the updated model to all nodes. It is especially wasteful when the current global model is similar to the old one.

Another simple solution is the periodic algorithm: sending updates once every T times Wolff et al. (2009); Bhaduri and Kargupta (2008) guarantees a reduction in communication. The problem is that a fixed update schedule must balance communication and error *a priori*. For large T the estimate error may be unbounded for a long interval, yet if model changes are infrequent we waste communication.

Recent approaches monitor the prediction error $|y - X\beta_0|$, where X, y are the current observations Song et al. (2013); Bhaduri and Kargupta (2008), the model's R^2 fit Bhaduri et al. (2011), or prediction error between divergent local models and the hypothetical global model Kamp et al. (2014b).

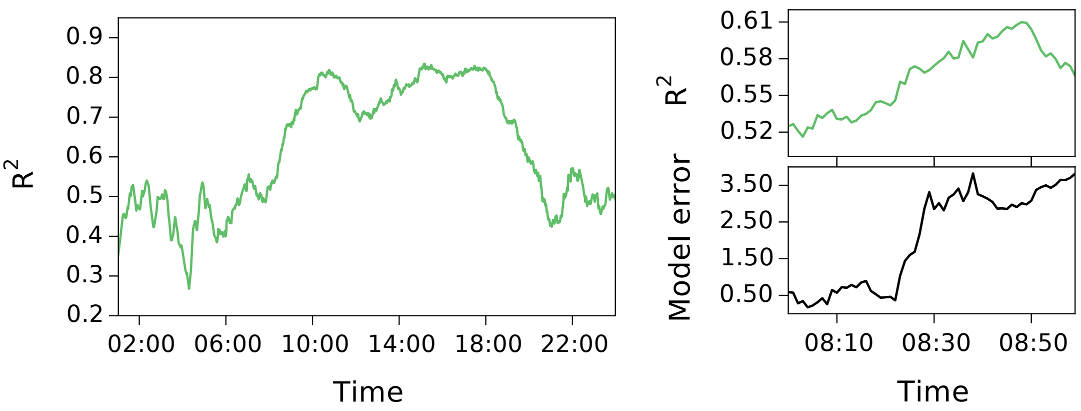
Monitoring prediction error is not always sufficient, however. First, prediction is not the only application of regression. In some settings Hayashi (2000); Ronen et al. (2014) we are interested in model coefficients, rather than prediction performance. Yet prediction error may be small even when the difference between models is large. Consider the following example in $m = 3$ dimensions, with the precomputed model $\beta_0 = (1, 2, 3)^T$, the current model $\beta = (1, 1, 1)^T$, and with the observation $x = (-0.95, 2.05, -0.95)^T, y = \beta^T x = 0.15$. In this case the prediction error is small, $|x^T \beta_0 - y| = 0.15$, yet the models are very different: $\|\beta_0 - \beta\| = 2.236$.

Monitoring model fit is also tricky. Figure 3.1a shows the R^2 fit of the true model β in an interpolation problem (described in Section 3.5.2). The fit of the true model varies widely, and it is not clear where to set the R^2 monitoring threshold. Figure 3.1b shows an example where both the model error $\|\beta - \beta_0\|$ and the fit of the monitored model β_0 are increasing.

Thus, we aim to monitor the model estimation error itself. Let β_0 be the existing model, previously computed at some point in the past (the synchronization time), and let β be the hypothetical OLS model from current observations[1]. Given an error threshold ϵ, our goal is to raise an alert if

$$\|\beta_0 - \beta\| > \epsilon \quad ,$$

[1] β is hypothetical since we don't actually compute it.



(a) Model β fit

(b) β_0 fit and error

Figure 3.1: (a) Model fit for the traffic dataset from Section 3.5.2, and (b) comparison with model error.

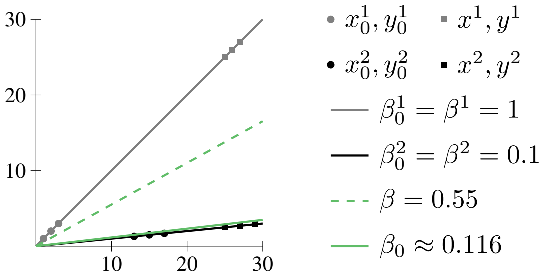


Figure 3.2: Monitoring distributed OLS models is difficult. Current local models β^1, β^2 are identical to the precomputed models β_0^1, β_0^2 but the combined global model is very different, $\beta - \beta_0 = 0.44$.

while minimizing communication. Note that monitoring model error is a more general approach: limiting model error allows us to bound prediction error $|x^T\beta_0 - x^T\beta|$ through Cauchy-Schwarz but not vice versa. Indeed, Lopes and Sayed (2006) estimate the expected model error and use it to get expected prediction error.

3.3 Monitoring Distributed Least Squares With Convex Subsets

Monitoring distributed OLS models is difficult because the global model cannot be inferred from the local model at each node. Let β^j be the current local model in node j, i.e. the model computed from the local subset (in j) of the global data; β_0^j denotes the previously computed local model. Even when all current local models β^j are similar to the precomputed local models β_0^j, the current global model β may be very different from the precomputed model β_0. Consider the example in Figure 3.2 with $k = 2$ nodes and $m = 1$. The global model deviation is very large, $\beta - \beta_0 = 0.44$, even though local models are identical: $\beta^1 = \beta_0^1$ and $\beta^2 = \beta_0^2$. In other words, similarity of current local models to their respective precomputed models does not imply that the current global model is similar to the precomputed model.

To overcome this difficulty, we turn to *geometric monitoring*. Geometric monitoring Keren et al. (2014, 2012) is a communication-efficient approach that monitors whether a function of distributed data streams crosses a threshold. The key idea is to impose constraints on local data at the nodes, rather than on the function of the global aggregate. Given a function of the average of all local data and the threshold, we compute a convex *safe zone* for each node. As we show below, convexity plays a key role in the correctness of this scheme. As long as local data stay inside the safe zones, we guarantee that the function of the global average does not cross a threshold. Nodes communicate only when local data drifts outside the safe zone, which we call a *safe zone violation*. Once that happens, violations can be resolved, for example by gathering data from all nodes and recomputing β_0 and the safe zones.

To summarize, we want to impose conditions on the local data at each node so that as long as they hold, $\|\beta - \beta_0\| \leq \epsilon$. The conditions should be as "lenient" as possible – we wish to minimize the number of violations.

3.3.1 Notation

Define $A \triangleq \sum_{i=1}^{n} x_i x_i^T = X^T X$ and $c \triangleq \sum_{i=1}^{n} x_i y_i = X^T y$, and rewrite Eq. (3.1) as $\beta = A^{-1}c$. Let O_j be the set of local observations in node j, meaning O_j's are disjoint and their union is the set of global observations. The global matrix A can be written as the sum of local matrices $A = \sum_{j=1}^{k} A^j$, where A^j is constructed from the local observations at node j: $A_j \triangleq \sum_{(x,y)\in O_j} xx^T$. Similarly, $c = \sum_{j=1}^{k} c^j$ where c^j is constructed from the local observations at node j: $c_j \triangleq \sum_{(x,y)\in O_j} xy$. Therefore, we can rewrite Eq. (3.1) as a function of the sums of A^j, c^j:

$$\beta = \left(\sum_j A^j\right)^{-1} \left(\sum_j c^j\right) = A^{-1}c \tag{3.2}$$

In our notation we use $\{A^j, c^j\}_k$ instead of the original observations $\{x_i, y_i\}_n$. Let $A_0 = \sum_{j=1}^{k} A_0^j$ and $c_0 = \sum_{j=1}^{k} c_0^j$ be the global sums of local values at nodes during the last sync time (when β_0 was computed), and $A = \sum_{j=1}^{k} A^j, c = \sum_{j=1}^{k} c^j$ be the current values. We define the local *drifts* as the deviation of local data from its initial values during sync: $\Delta^j = A^j - A_0^j$ and $\delta^j = c^j - c_0^j$.

We can now express global β and β_0 as a function of the averages of A^j, c^j and A_0^j, c_0^j. This will allow us to bound model changes inside a convex subset. Recall $\beta = A^{-1}c$. Similarly, $\beta_0 = A_0^{-1}c_0$. Values averaged over nodes (rather than summed) shall be denoted with a "hat": $\hat{Z} = \frac{1}{k}\sum_{j=1}^{k} Z_j$ for some value Z_j for each node j.

Hence initial average values of A, c and the local model are:

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

and their current values are

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

Note $(\frac{1}{k}A)^{-1} = kA^{-1}$ thus $\hat{\beta} = \hat{A}^{-1}\hat{c} = A^{-1}c = \beta$ and likewise $\hat{\beta}_0 = \beta_0$. In other words, we can compute the OLS model from the averages of local A^j, c^j rather than the sums:

$$\beta = \left(\frac{1}{k}\sum_j A^j\right)^{-1} \left(\frac{1}{k}\sum_j c^j\right) = \hat{A}^{-1}\hat{c} \tag{3.3}$$

3.3.2 Convex Safe Zones

We propose to solve the monitoring problem by means of "good" convex subsets, called *safe zones*, of the data space. Each node monitors its own drift: as long as current values at local nodes (A^j, c^j) are sufficiently similar to their values at sync time (A_0^j, c_0^j), β_0 is guaranteed to be close to β.

Formally, we define a convex subset $\mathcal{C}$ in the space of matrix-vector pairs, such that $(0_{m\times m}, 0_m) \in \mathcal{C}$ and

$$(\Delta, \delta) \in \mathcal{C} \implies \|(\hat{A}_0 + \Delta)^{-1}(\hat{c}_0 + \delta) - \hat{A}_0^{-1}\hat{c}_0\| \leq \epsilon \quad , \tag{3.4}$$

for any drift (Δ, δ), where $0_{m\times m}$ and 0_m are the $m \times m$ zero matrix and length m zero vector. Ideally, $\mathcal{C}$ should be "big": as local data slowly drifts over time, it is desirable that drifts remain in $\mathcal{C}$ (otherwise communication is needed). Convexity plays a key role in our paradigm: if all drifts are in $\mathcal{C}$, then their average is also in $\mathcal{C}$.

Given such a subset $\mathcal{C}$, the basic monitoring paradigm is simple. As long as $(A^j - A_0^j, c^j - c_0^j) \in \mathcal{C}$, node j can remain silent. If all nodes are 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.

We now prove the correctness of the paradigm.

Lemma 1. *Let $\mathcal{C}$ be a convex subset that satisfies Eq. (3.4). If $(\Delta^j, \delta^j) \in \mathcal{C}$ for all j, then $\|\beta - \beta_0\| \leq \epsilon$.*

Proof. Express $\hat{A}, \hat{c}$ using the average of local deviations:

$$\begin{aligned}
(\hat{A}, \hat{c}) &= \frac{1}{k}\sum_j (A^j, c^j) \\
&= (\hat{A}_0, \hat{c}_0) + \frac{1}{k}\sum_j (A^j - A_0^j, c^j - c_0^j) \\
&= (\hat{A}_0, \hat{c}_0) + \frac{1}{k}\sum_j (\Delta^j, \delta^j)
\end{aligned} \tag{3.5}$$

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

$$\forall j\ (\Delta^j, \delta^j) \in \mathcal{C} \implies \frac{1}{k}\sum_j (\Delta^j, \delta^j) \in \mathcal{C} \tag{3.6}$$

Denote $(\hat{\Delta}, \hat{\delta}) = \frac{1}{k}\sum_j (\Delta^j, \delta^j)$ and rewrite Eq. (3.5) and (3.6):

$$\begin{aligned}
(\hat{A}, \hat{c}) &= (\hat{A}_0, \hat{c}_0) + (\hat{\Delta}, \hat{\delta}) \\
\forall j\ (\Delta^j, \delta^j) \in \mathcal{C} &\implies (\hat{\Delta}, \hat{\delta}) \in \mathcal{C}
\end{aligned}$$

Substitute in Eq. (3.4) to finally obtain:

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

which completes the proof. □

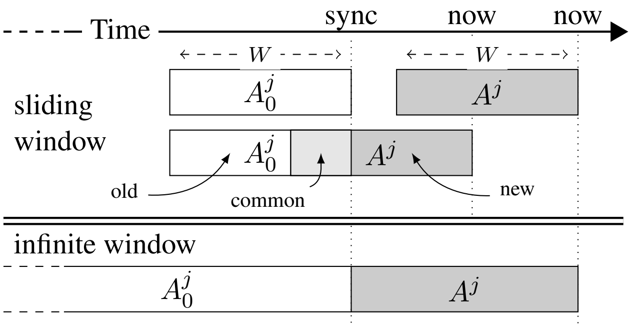


Figure 3.3: Sliding and infinite window models. When A^j overlaps A_0^j, $\Delta^j = A^j - A_0^j = \sum_{\text{new}} x_i x_i^T - \sum_{\text{old}} x_i x_i^T$.

3.3.3 Infinite and Sliding Window

We differentiate between two different variations for computing the global model: sliding window and infinite window. In the *sliding window* model, β is computed from the last W samples seen at each node, and similarly β_0 is computed from the last W samples before sync. Conversely, in the *infinite window* model β is computed over all observations seen thus far, while β_0 is computed from all observations seen until last sync. Figure 3.3 illustrates these two models. Though the sliding window is clearly more practical, the infinite window model may be useful in some settings and so we discuss both.

Sliding Window In the sliding window model each node computes A^j from the W samples seen at node j, while A_0^j (and hence $\hat{A}_0$) is built from the last W samples before sync. Computing Δ^j and δ^j, however, requires substracting observations that left the sliding window. If A_0^j, c_0^j and A^j, c^j do not overlap (Figure 3.3, top), then clearly $\Delta^j = A^j - A_0^j$ and $\delta^j = c^j - c_0^j$. It is also possible, however, that the current window overlaps the window used to build β_0. Figure 3.3 (middle) illustrates this case: Δ^j, δ^j become the sum of new samples from A^j, c^j minus the sum of old (non-overlapping) samples from A_0^j, c_0^j.

The convex constraint $\mathcal{C}$ on (Δ, δ) for this model is:

$$\epsilon\|\hat{A}_0^{-1}\Delta\| + \|\hat{A}_0^{-1}\delta\| + \|\hat{A}_0^{-1}\Delta\beta_0\| \le \epsilon \quad , \tag{3.7}$$

where $\|A\|$ is the L_2 *operator norm* of the matrix A. The derivation of the convex constraint $\mathcal{C}$ is quite technical, and the details are available in Section 3.4.1.

Algorithm 1 Node j update with new observation x, y.

```
$(A^j\ ,\ c^j) \leftarrow (A^j + x^T x\ ,\ c^j + x^T y)$
Insert new $x, y$ to head of sliding window.
Retrieve old $x_w, y_w$ exiting end of sliding window.
$(A^j\ ,\ c^j) \leftarrow (A^j - x_w^T x_w\ ,\ c^j - x_w^T y_w)$
$(\Delta^j\ ,\ \delta^j) \leftarrow (A^j - A_0^j\ ,\ c^j - c_0^j)$
if $\epsilon\|\hat{A}_0^{-1}\Delta^j\| + \|\hat{A}_0^{-1}\delta^j\| + \|\hat{A}_0^{-1}\Delta^j\beta_0\| \le \epsilon$ then
    Report violation to coordinator.
    Receive new $\beta_0$, $\hat{A}_0^{-1}$ from coordinator.
    $(A_0^j\ ,\ c_0^j) \leftarrow (A^j\ ,\ c^j)$
end if
```

Alg. 1 shows the resulting monitoring algorithm each node runs. Note monitoring does not require any matrix inversions. Each node applies the local constraint from Eq. (3.7) to its own data. When a violation occurs at any node, it is reported to a *coordinator* node.

The coordinator (Alg. 2) polls all nodes for their local data, computes a updated global model β_0 and distributes it to all nodes, along with updated $\hat{A}_0^{-1}$ used in the constraint. Monitoring then resumes. This is the simplest violation resolution protocol, but our method is compatible with recent communication reduction techniques from the field of distributed streams, such as reference point prediction Giatrakos et al. (2012), individualized constraints or slack Keralapura et al. (2006); Gabel et al. (2014a), and local violation resolution Keren et al. (2014). Similarly, our distributed monitoring approach can easily be combined with an efficient distributed computation technique to compute $\hat{A}_0^{-1}, \beta_0$, enjoying the best of both worlds. The current model can be computed during sync using any of several existing algorithms, be they exact, iterative, or distributed Guestrin et al. (2004); Lopes and Sayed (2006).

Algorithm 2 Coordinator violation resolution algorithm.

```
1: Poll all nodes for $A^j, c^j$.
2: Compute updated $\hat{A}_0^{-1}, \beta_0$ from $A^j, c^j$ and distribute.
```

Infinite Window In this model the local drifts of each node i are $\Delta^j = A^j - A_0^j$ and $\delta^j = c^j - c_0$ as before, but A^j and c^j are computed from all observations ever seen at the node. We can use the same convex constraint from Section 3.3.3, but in this case Δ^j grows indefinitely, and so the condition $\|\hat{A}_0^{-1}\Delta\| < 1$ is not easy to satisfy, and may cause frequent synchronizations. Instead, we start from Eq. (3.9) and develop a more lenient constraint for this model. The resulting algorithm will be similar to Alg. 1, but without lines 2–4 and with the updated constraint in line 6. The coordinator algorithm is the same.

The convex constraint for the infinite window case is

$$\|\hat{A}_0^{-1}\delta\| + \|\hat{A}_0^{-1}\hat{c}_0\| \leq \epsilon \quad . \tag{3.8}$$

Section 3.4.2 details its derivation.

Note that δ accumulates more samples as time passes, while $\hat{A}_0$ remains fixed. As δ's "weight" (number of samples) grows beyond $\hat{A}_0$'s, the constraint no longer holds and syncrhonization is needed. One way to avoid this is to replace $\|\hat{A}_0^{-1}\delta\|$ in Eq. 3.8 with $\|\Delta^{-1}\delta\|$, which is correct (using the same line of arguments in Section 3.4.2). Alternatively, note that after each sync the samples from all δ^j's are added to the new $\hat{A}_0$, so its "weight" is roughly doubled. Thus $\hat{A}_0$'s weight grows exponentially, and synchronizations become increasingly rare.

3.3.4 Norm Constraint and the Sliding Window

The sliding window model constraint Eq. (3.7) requires $\|\hat{A}_0^{-1}\Delta^j\| < 1$ (embodied as $\epsilon\|\hat{A}_0^{-1}\Delta^j\| + \cdots \leq \epsilon$). This requirement depends only the independent variables X^j, and does not depend in any way on the dependent variable y^j. It is quite possible that β is close to β_0, yet the norm constraint is violated, incurring extra communication. Fortunately, for many reasonable data distributions, if window size W is linear in the number of independent variables m then the norm constraint is satisfied almost surely. The details are in Section 3.4.3.

The analysis assumes that consecutive observations in the data stream are independent. Consider, however, the case where some variables come from an over-sampled sensor, or measure slowly changing phenomena. In such cases $\|\Delta\|$ grows faster, linear in the number of identical observations, and

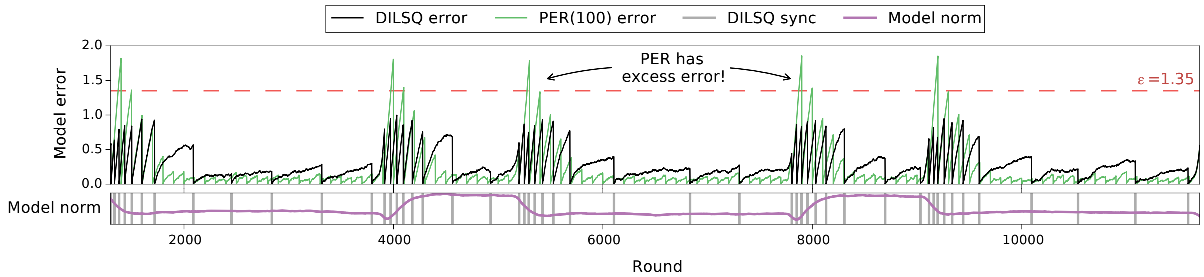


Figure 3.4: DILSQ model error (black) and syncs (bottom vertical lines) per round, compared to PER(100) error (green), for $k = 10$ simulated nodes with $m = 10$ dimensions, and threshold $\epsilon = 1.35$. Both algorithms reduce communication to 1%, but DILSQ only syncs when β changes (bottom purple line shows $\|\beta\|$). PER(100) syncs every 100 rounds, but is unable to maintain error below the threshold (dashed horizontal line).

will overwhelm A_0^{-1} faster. This can result in more frequent violations of the constraint, hence more communication. Such cases can be mitigated by increasing the window size W, subsampling (since data changes slowly anyway), or by the use of *generalized least squares* Hayashi (2000) with an appropriate scaling matrix for the time series process (Section 3.3.5).

3.3.5 Regularization and Variants

Our scheme generalizes very well to more sophisticated least squares variants Hayashi (2000). We show two examples.

In *regularized least squares* the minimized function includes a regularization term to mitigate the effects of outliers and avoid overfitting. A commonly used form is *Tikhonov regularization*, also known as *ridge regression*, which finds β that minimizes $\|X\beta - y\|^2 + \|R^T R\beta\|^2$, where R is a suitable regularization matrix R. For $R = 0$ the problem reduces to ordinary least squares, and for $R = \lambda I$ it reduces to L_2 regularization. The optimal solution to this problem is

$$\beta = \left(X^T X + R^T R\right)^{-1} X^T y \quad .$$

This solution is quite similar to Eq. (3.1) and indeed we can monitor it using the same technique: compute $B_0^j = A_0^j + \frac{1}{k} R^T R$ and the resulting $B_0, \hat{B}_0$, and use them in Lemma 1 instead of $A_0^j, A_0, \hat{A}_0$.

Similarly, *generalized least squares* handles correlated measurements and errors by minimizing the Mahalanobis distance $(Y - X\beta)^T S^{-1}(Y - X\beta)$, where S is the covariance matrix of the residuals (errors). Again, GLS reduces to OLS if $S = I$. As before, we can monitor the optimal solution

$$\beta = \left(X^T S^{-1} X\right)^{-1} X^T \tilde{y} \text{ , where } \tilde{y} \triangleq S^{-1} y$$

by monitoring $B = \sum_i S^{-1} x_i x_i^T$ and $d = \sum_i x_i \tilde{y}_i$. GLS is particularly useful in time series analysis, where S is the process' structured covariance (or autocorrelation) matrix Saudargienė (1999).

3.4 Deriving Constraints

This section contains detailed technical proofs.

3.4.1 Sliding Window Constraint

To find a convex subset $\mathcal{C}$ satisfying the condition of Eq. (3.4), we first review some notions and well-known results on norms of real matrices Roman (1995). We use the L_2 norm throughout.

Definition 1. *Let A be a matrix. Its* operator norm, *or* spectral norm, *hereafter just* norm, *is defined as*

$$\|A\| = \sup_{x \neq 0} \frac{\|Ax\|}{\|x\|}$$

It follows that for a matrix A and vector x, $\|Ax\| \leq \|A\|\|x\|$.

Moreover, for any two matrices A, B: $\|A + B\| \leq \|A\| + \|B\|$ and $\|AB\| \leq \|A\|\|B\|$. If A is a symmetric matrix, $\|A\| = \max_i |\lambda_i|$ where λ_i are the eigenvalues of A. Additionally, if A, B are symmetric then $\|AB\| = \|BA\|$.

Lemma 2. *For square A with $\|A\| < 1$, (I-A) is invertible, the inverse being the* Neumann series *Miller (1981): $(I - A)^{-1} = I + A + A^2 + A^3 + \ldots$.*

Lemma 3. *If A is square and $\|A\| < 1$, then*

$$\|(I + A)^{-1}\| = \|I - A + A^2 - A^3 + \ldots\| \leq \frac{1}{1 - \|A\|} \quad .$$

Proof. Apply Lemma 2 and the triangle inequality:

$$\begin{aligned} \|\,(I + A)^{-1}\,\| &= \|I - A + A^2 - A^3 + A^4 - \ldots\| \\ &\leq \|I\| + \|A\| + \|A^2\| + \cdots \leq \frac{1}{1 - \|A\|} \quad , \end{aligned}$$

since it is the sum of a geometric series. □

We begin by subtracting and adding $(\hat{A}_0 + \Delta)^{-1}\hat{c}_0$ to the bounded expression in Eq. (3.4):

$$\begin{aligned} &\|(\hat{A}_0 + \Delta)^{-1}(\hat{c}_0 + \delta) - \hat{A}_0^{-1}\hat{c}_0\| = \\ &\qquad \|(\hat{A}_0 + \Delta)^{-1}\delta + \left((\hat{A}_0 + \Delta)^{-1} - \hat{A}_0^{-1}\right)\hat{c}_0\| \quad . \end{aligned}$$

Applying the triangle inequality, we obtain:

$$\underbrace{\|(\hat{A}_0 + \Delta)^{-1}\delta\|}_{E_1} + \underbrace{\left\|\left((\hat{A}_0 + \Delta)^{-1} - \hat{A}_0^{-1}\right)\hat{c}_0\right\|}_{E_2} \quad . \tag{3.9}$$

Next, note that

$$(\hat{A}_0 + \Delta)^{-1} = \left(\hat{A}_0\left(I + \hat{A}_0^{-1}\Delta\right)\right)^{-1} = \left(I + \hat{A}_0^{-1}\Delta\right)^{-1}\hat{A}_0^{-1}$$

and, assuming $\|\hat{A}_0^{-1}\Delta\| < 1$, we apply Lemma 2 to obtain:

$$\begin{aligned} &(\hat{A}_0 + \Delta)^{-1} \\ &\quad = \left(I - \hat{A}_0^{-1}\Delta + \hat{A}_0^{-1}\Delta\hat{A}_0^{-1}\Delta - \ldots\right)\hat{A}_0^{-1} && (3.10) \\ &\quad = \hat{A}_0^{-1} - \hat{A}_0^{-1}\Delta\hat{A}_0^{-1} + \hat{A}_0^{-1}\Delta\hat{A}_0^{-1}\Delta\hat{A}_0^{-1} - \ldots && (3.11) \end{aligned}$$

Note the assumption $\|\hat{A}_0^{-1}\Delta\| < 1$ is not trivial, and we discuss it in Section 3.3.4 and Section 3.4.3.

We now apply Eq. (3.10) and Lemma 3 to E_1 in Eq. (3.9):

$$
\begin{aligned}
E_1 &= \|(\hat{A}_0 + \Delta)^{-1}\delta\| \\
&= \left\| \left(I - \hat{A}_0^{-1}\Delta + \hat{A}_0^{-1}\Delta\hat{A}_0^{-1}\Delta - \ldots \right) \hat{A}_0^{-1}\delta \right\| \\
&\leq \|I - \hat{A}_0^{-1}\Delta + \hat{A}_0^{-1}\Delta\hat{A}_0^{-1}\Delta - \ldots\| \|\hat{A}_0^{-1}\delta\| \\
&\leq \frac{\|\hat{A}_0^{-1}\delta\|}{1 - \|\hat{A}_0^{-1}\Delta\|}
\end{aligned}
\tag{3.12}
$$

Similarly, we apply Eq. (3.11) to E_2:

$$
\begin{aligned}
E_2 &= \left\| \left((\hat{A}_0 + \Delta)^{-1} - \hat{A}_0^{-1} \right) \hat{c}_0 \right\| \\
&= \left\| \left(\hat{A}_0^{-1} - \hat{A}_0^{-1}\Delta\hat{A}_0^{-1} + (\hat{A}_0^{-1}\Delta)^2\hat{A}_0^{-1} - \cdots - \hat{A}_0^{-1} \right) \hat{c}_0 \right\| \\
&= \left\| - \left(\hat{A}_0^{-1}\Delta + (\hat{A}_0^{-1}\Delta)^2 - \ldots \right) \hat{A}_0^{-1}\hat{c}_0 \right\| \\
&= \left\| \left(I + \hat{A}_0^{-1}\Delta - (\hat{A}_0^{-1}\Delta)^2 + \ldots \right) \hat{A}_0^{-1}\Delta\beta_0 \right\|
\end{aligned}
\tag{3.13}
$$

Applying Lemma 3 to Eq. (3.13) we obtain:

$$
\begin{aligned}
E_2 &\leq \left\| I + \hat{A}_0^{-1}\Delta - (\hat{A}_0^{-1}\Delta)^2 + \ldots \right\| \|\hat{A}_0^{-1}\Delta\beta_0\| \\
&\leq \frac{\|\hat{A}_0^{-1}\Delta\beta_0\|}{1 - \|\hat{A}_0^{-1}\Delta\|}
\end{aligned}
\tag{3.14}
$$

Substituting Eq. (3.12) and (3.14) in Eq. (3.9) and rearranging, we arrive at the convex constraint $\mathcal{C}$ on (Δ, δ):

$$
\epsilon\|\hat{A}_0^{-1}\Delta\| + \|\hat{A}_0^{-1}\delta\| + \|\hat{A}_0^{-1}\Delta\beta_0\| \leq \epsilon \quad . \tag{3.15}
$$

This convex constraint allows us to apply Lemma 1. Satisfying Eq. (3.15) guarantees Eq. (3.4) is also satisfied, since the bounded expression is larger. Moreover, this bound is a subset of $\|\hat{A}_0^{-1}\Delta\| < 1$, a necessary condition for correctness, meaning we don't have to check it explicitly.

3.4.2 Infinite Window Constraint

A matrix A is *positive definite*, denoted $A \succ 0$, if $x^T A x > 0$ for all non-zero vectors x. This implies a partial ordering of square matrices: we denote $A \succ B$ if $A - B \succ 0$. Note $A \succ B \succ 0 \implies \|A\| > \|B\|$. Moreover, $A \succ B \succ 0 \implies B^{-1} \succ A^{-1} \succ 0$. Finally, observe that $\|(A+B)^{-1}u\| \leq \|A^{-1}u\|$, since $A + B \succ A$ and therefore $A^{-1} \succ (A+B)^{-1}$. Similarly, $\| \left((A+B)^{-1} - A^{-1} \right) u\| = \| \left(A^{-1} - (A+B)^{-1} \right) u\| \leq \|A^{-1}u\|$.

We apply the above to Eq. (3.9). Note that by construction, $\Delta^j = \sum_{i \in \mathcal{S}_j} x_i x_i^T$, where $\mathcal{S}_j$ is the set samples seen by node j since the last sync time, is symmetric and positive definite. Similarly, $\hat{A}_0$ is symmetric positive definite by construction. Thus, $E_1 = \|(\hat{A}_0 + \Delta)^{-1}\delta\| \leq \|\hat{A}_0^{-1}\delta\|$, and $E_2 = \left\| \left((\hat{A}_0 + \Delta)^{-1} - \hat{A}_0^{-1} \right) \hat{c}_0 \right\| \leq \|\hat{A}_0^{-1}\hat{c}_0\|$.

The final convex constraint for the infinite window case is therefore

$$
\|\hat{A}_0^{-1}\delta\| + \|\hat{A}_0^{-1}\hat{c}_0\| \leq \epsilon \quad . \tag{3.16}
$$

3.4.3 Window Size And Dimensions

We will show that sliding window W linear in m will avoid overwhelming $\|\hat{A}_0^{-1}\Delta^j\|$ in Eq. (3.7).

For any matrix A, denote its largest and smallest eigenvalues by $\lambda_{\text{max}}(A)$ and $\lambda_{\text{min}}(A)$. Recall that $\hat{A}_0 = \frac{1}{k}A_0^j$ and that in the sliding window model[2], $\Delta^j = A^j - A_0^j$, and that all these matrices are symmetric by construction. Moreover, if A is symmetric then $\|A\| = \sqrt{\lambda_{\text{max}}(A^T A)} = |\lambda_{\text{max}}(A)|$. Finally, $\lambda_{\text{max}}(A^{-1}) = \frac{1}{\lambda_{\text{min}}(A)}$, and therefore

$$\|\hat{A}_0^{-1}\| = \left\|\frac{1}{k}A_0^{-1}\right\| = \frac{k}{|\lambda_{\text{min}}(A_0)|} = \frac{k}{\lambda_{\text{min}}(A_0)} \quad .$$

Applying the above to the norm constraint:

$$\begin{aligned}\|\hat{A}_0^{-1}\Delta^j\| \leq \|\hat{A}_0^{-1}\|\|\Delta^j\| &= \frac{k}{\lambda_{\text{min}}(A_0)}|\lambda_{\text{max}}(A^j - A_0^j)| \\ &\leq \frac{k}{\lambda_{\text{min}}(A_0)}|\lambda_{\text{max}}(A^j) - \lambda_{\text{min}}(A_0^j)| \quad . \end{aligned} \tag{3.17}$$

The last step is obtained from Knutson and Tao (2001):

$$A, B \text{ symmetric} \implies \lambda_{\text{max}}(A+B) \leq \lambda_{\text{max}}(A) + \lambda_{\text{max}}(B)$$

and since $\lambda_{\text{max}}(-B) = -\lambda_{\text{min}}(B)$.

The bound in Eq. (3.17) depends on the distribution of the data. Assume the elements of X are drawn i.i.d from $N(0,1)$, then the *Marchenku-Pastur law* Marčenko and Pastur (1967) limits the spectrum of the *Wishart matrix* $X^T X$.

Lemma 4. *Let $X \in \mathbb{R}^{w \times m}$ drawn as above such that $\frac{m}{W}$ converges to $0 < b \leq 1$ as W and m grow to infinity*[3]*. Let $M = \frac{1}{W}X^T X$, and denote its largest and smallest eignevalues by $\lambda_{max}(M)$, $\lambda_{min}(M)$. Then almost surely*

$$\lambda_{max}(M) \to \left(1 + \sqrt{b}\right)^2, \; \lambda_{min}(M) \to \left(1 - \sqrt{b}\right)^2 \quad .$$

Bai and Yin Bai and Yin (1993) extended this result to *any* zero-mean distribution with unit variance and finite fourth moment Rudelson and Vershynin (2010). These can be achieved using Gabel et al. (2014a), for example.

Note $A_0 = \sum_1^k A_0^j = \tilde{X_0}^T \tilde{X_0}$, where $\tilde{X_0} \in \mathbb{R}^{kW \times m}$ is the concatenation of all local data matrices. Applying Lemma 4 to Eq. (3.17), we obtain

$$\begin{aligned}\frac{k|\lambda_{\text{max}}(A^j) - \lambda_{\text{min}}(A_0^j)|}{\lambda_{\text{min}}(A_0)} &= \frac{kW\left|\lambda_{\text{max}}(\frac{1}{W}A^j) - \lambda_{\text{min}}(\frac{1}{W}A_0^j)\right|}{kW\lambda_{\text{min}}(\frac{1}{kW}A_0)} \\ &= \frac{|\lambda_{\text{max}}(\frac{1}{W}A^j) - \lambda_{\text{min}}(\frac{1}{W}A_0^j)|}{\lambda_{\text{min}}(\frac{1}{kW}A_0)} \quad ,\end{aligned}$$

which converges almost surely to

$$f_k(b) \triangleq \frac{|(1+\sqrt{b})^2 - (1-\sqrt{b})^2|}{\left(1 - \sqrt{\frac{b}{k}}\right)^2} = \frac{4\sqrt{b}}{\left(1 - \sqrt{\frac{b}{k}}\right)^2} \quad . \tag{3.18}$$

[2] We discuss the worst case, when A^j, A_0^j do not overlap. When they do, Δ^j's effective window size is less than W.

[3] Trivially, if $W = \frac{m}{b}$.

In practice, Eq. (3.7) is the sum of 3 norms, so we require $\|\hat{A}_0^{-1}\Delta^j\| < \frac{1}{3}$. Solving $0 < f_k(b) < \frac{1}{3}$ for b with $k > 1$ yields

$$\frac{m}{W} \leq g_k \triangleq 72k^2 + k - 24k^{\frac{3}{2}} - 4\sqrt{3}\sqrt{108k^4 + 15k^3 - 72k^{\frac{7}{2}} - k^{\frac{5}{2}}}.$$

For given $k > 1$, selecting $W \geq \frac{m}{g_k}$ guarantees $\|\hat{A}_0^{-1}\Delta^j\| < \frac{1}{3}$ almost surely. The constant $\frac{1}{g_k}$ grows slowly: for $k = 2$, the window size W must be at least $\frac{1}{g_2} \approx 111.06m$; for $k = 10$, $W \geq \frac{1}{g_{10}} \approx 129.02m$; and for $k = 100$, $W \geq \frac{1}{g_{100}} \approx 139.22m$. In fact, g_k converges: $lim_{k\to\infty} g_k = \frac{1}{144}$, so a window size of $W \geq 144m$ is sufficient for any k.

3.5 Evaluation

We evaluated performance of our monitoring algorithm, DILSQ, for DIstributed Least SQuare monitor, using simulations with two synthetic and two real-world distributed datasets. For each dataset, we run through the data, simulate the nodes (Alg 1) and the coordinator (Alg 2), count messages, and keep track of the resulting true models β and the current monitored models β_0. Our simulations use discrete time (rounds), and we use the OLS variant of our algorithm with sliding window (Section 3.3.3), except for the gas sensor dataset which uses the GLS variant (Section 3.3.5).

Our baseline is the *naive algorithm*, where each node sends every new measurement to a centralized location each round. We compare DILSQ to the T-periodic algorithm, denoted PER(T), a simple sampling algorithm that sends updates every T rounds. Though PER cannot guarantee maximum error, it can achieve arbitrarily low communication.

Our main performance metric is communication, measured in *normalized messages* – the average messages sent per round by each node Bhaduri et al. (2011). Note that communication of the naive algorithm is always 1. When calculating and reporting results, we skip the first (incomplete) window (or the first epoch for the drift dataset described below).

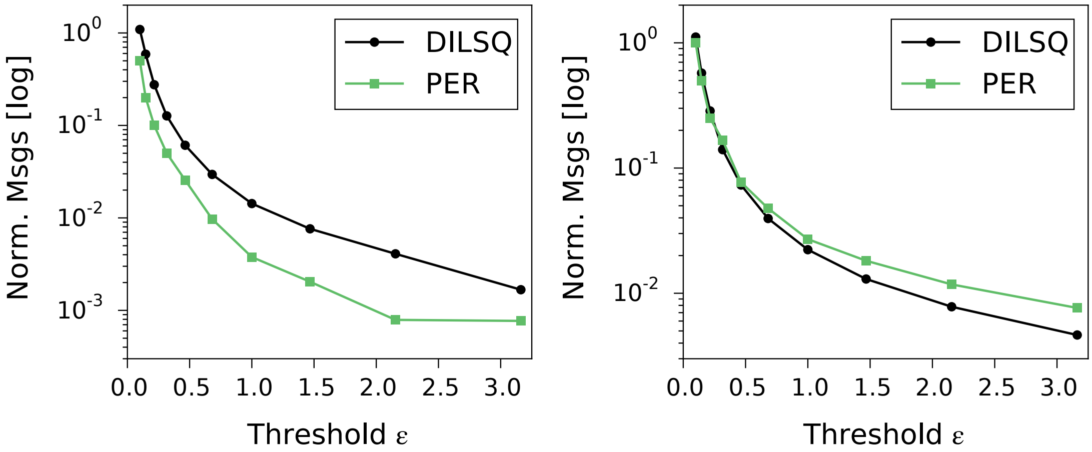
DILSQ is designed to communicate as little as possible while always maintaining maximum model error below ϵ. It *guarantees* maximum model error below the user-selected threshold ϵ, but PER does not. Hence, when comparing the two, we find *a posteriori* the maximum period T (hence minimum communication) for which the maximum error of PER(T) is equal or below that of DILSQ. Note this gives PER an unrealistic advantage. First, in a realistic setting we cannot know *a priori* the optimal period T. Second, model changes in realistic settings are not necessarily stationary: the rate of model change may evolve, which DILSQ will handle gracefully while PER cannot.

3.5.1 Synthetic Datasets

We use two types of synthetic dataset. In the *fixed* dataset, the true model $\beta_{\text{true}} \in \mathbb{R}^m$ is fixed, with elements drawn i.i.d from $N[0,1]$[4]. We generate R rounds with k nodes, each receiving at each round a new data vector x of size m and scalar y. x is drawn i.i.d from $N(0,1)$, and $y = x^T\beta_{\text{true}} + n$ where $n \sim N(0,\sigma^2)$ is Gaussian white noise of strength σ. In the *drift* dataset the coefficients of β_{true} change rapidly during 25% of one *epoch*, and are fixed during the rest of the epoch. We generate observations for E epochs using the same procedure. For each experiment we generate new data.

Default parameter values are $k = 10$ nodes, $m = 10$ dimensions, noise magnitude $\sigma = 10$ (to generate interesting results given the large window), window size $W = 1300$ and maximum error

[4] therefore $\|\beta\|^2 \sim \chi^2_m$



(a) Fixed dataset

(b) Drift dataset

Figure 3.5: Communication for DILSQ (black) and periodic algorithm tuned to achieve same max error (green) at different threshold values. DILSQ communication on fixed model drops to zero for more permissive ϵ (not shown on logarithmic scale).

threshold $\epsilon = 0.5$, which is quite strict[5]. We generate $R = 16900$ rounds for the fixed dataset, or $E = 5$ epochs of 3900 rounds each for drift dataset.

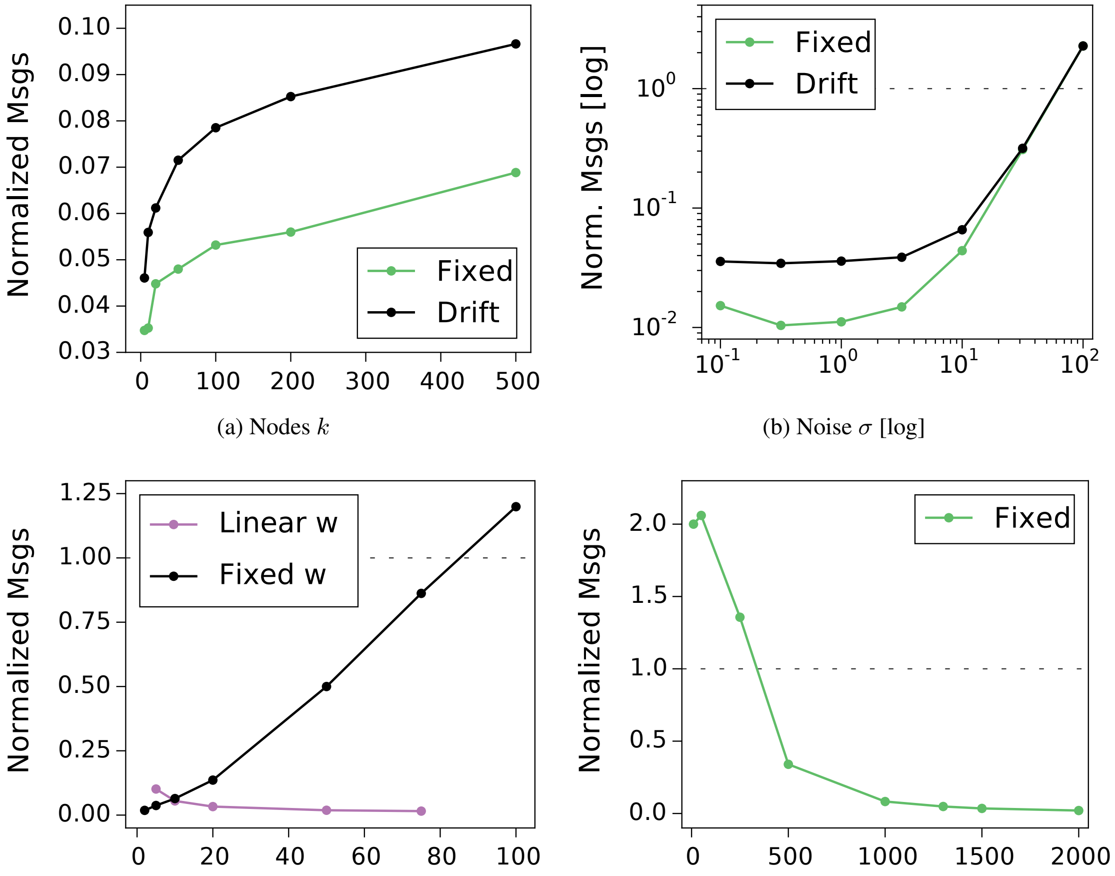
Figure 3.4 shows the behavior of the monitoring algorithm over such a simulation on the drift dataset with $\epsilon = 1.35$ and 3 epochs. For this configuration, DILSQ achieves communication of 0.01 messages per node per round, and the model error is always below the threshold. Conversely, the equivalent PER(100) algorithm is unable to maintain the error below the threshold, which would require a higher update frequency. When model changes in β are large and frequent DILSQ performs more syncrhonizations, resulting in updated $\beta_0 = \beta$ that decreases the error. When β is stable (it is never truly constant due to noise), syncrhonizations are much rarer. The periodic algorithm, on the other hand, syncrhonizes every 100 iterations even during the periods where β changes very little.

Effect of Threshold

Figure 3.5 shows the communication required for different threshold levels for the DILSQ algorithm, and the minimal communication required to match DILSQ using the PER algorithm with optimal period, as discussed above.

For the fixed model dataset (Figure 3.5a) neither algorithm needs to sync very often to provide an accurate estimate. While PER appears to require lesser communication here, note that the true model is *fixed*, and that PER has an unrealistic advantage of choosing the optimal period after seeing all data. When the model is fixed there is no reason to monitor at all: had there been no noise, a single initial synchronization would have been sufficient, regardless of threshold. PER is simply the minimal *posteriori* sampling rate to overcome noise. Indeed, for more permissive threshold values (or smaller noise magnitude σ) both DILSQ and PER achieve *zero* communication (beyond initial sync) for the fixed dataset (not shown in this log-scale figure).

[5] Given that elements of both β_0 and β are i.i.d $N(0, \sigma)$, then $\frac{\|\beta - \beta_0\|}{\sqrt{2}\sigma} \sim \chi_m$. The probability that a random $e = \beta - \beta_0$ will overwhelm ϵ is $P = 1 - \text{CDF}_{\chi_m}(\frac{\epsilon}{\sqrt{2}\sigma}) > 1 - 10^{-8}$.



(a) Nodes k

(b) Noise σ [log]

(c) Dimension m

(d) Window size W

Figure 3.6: Communication vs. different parameters for the fixed (green) and drift (black) datasets. Default parameter values are given in Section 3.5.1. (a) shows DILSQ is scalable: communication increases slowly with number of nodes. (b) shows communication is fairly constant when noise is small ($\sigma < 10$). Comm. is zero for fixed model at low noise (not shown). (c) shows the required window size W is linear in m: communication does not increase when W is suitably sized (purple). (d) shows performance with fixed dataset. If $W < 144m$ data periodically saturates the norms in Eq. (3.7).

Performance on the drift dataset (Figure 3.5b) is more interesting. When ϵ is very strict, both algorithms perform roughly the same, with normalized messages of 0.25–0.75. As ϵ grows DILSQ develops an increasing advantage over PER with optimal period. The optimal period must be low enough to match the quickly changing model, and is wasteful on the intervals where β is quiescent. For our dataset, β_{true} is constant during roughly 75% of each epoch. For datasets with larger quiescent periods (or smaller window), the advantage of DILSQ will be even larger.

Scalability

Figure 3.6 explores how performance of DILSQ scales with different parameters.

Figure 3.6a shows communication for different values of the number of nodes k. We observe communication increases slowly, remaining below 10% even with 500 nodes.

Figure 3.6b shows normalized messages obtained at different noise magnitudes. Below a certain

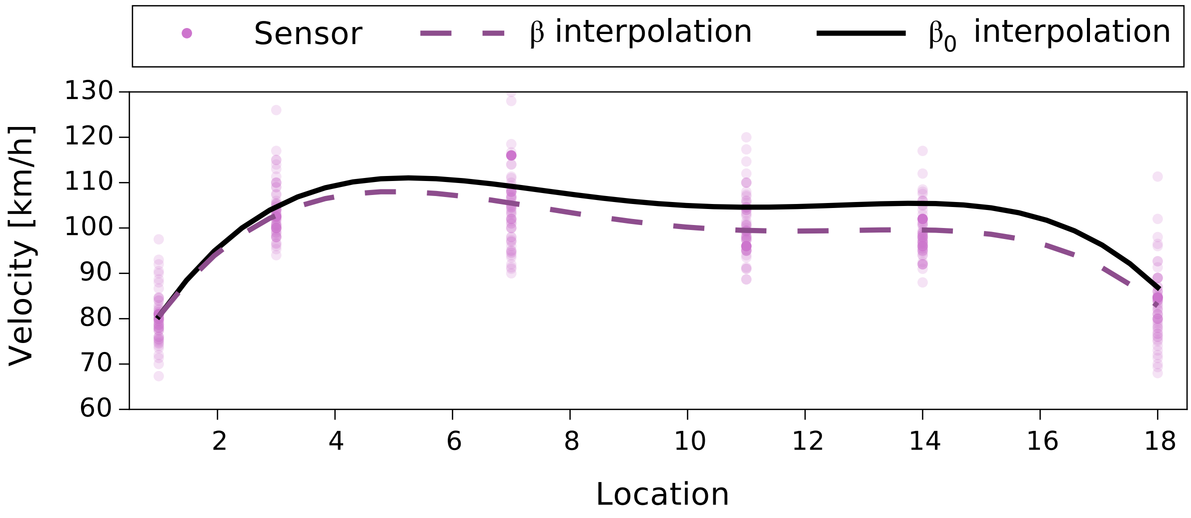


Figure 3.7: Velocity measurements from 8am–9am (pink dots), and interpolated velocity at 9am: true model (dashed) and DILSQ approximation (black).

level of noise, communication is fairly constant, reflecting the choice of threshold ϵ. At lower values of noise (not shown), DILSQ requires no communication for the fixed model dataset, beyond the first window of W observations.

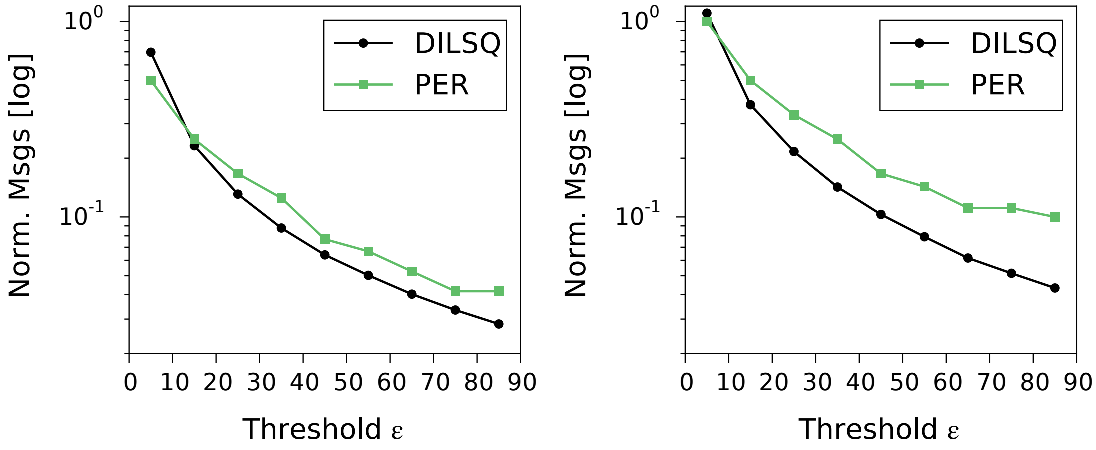
Figure 3.6c compares communication with the number of independent variables m on the drift dataset, confirming our analysis in Section 3.3.4. When window size W is fixed, communication grows linearly with dimension m. However, if W grows linearly with m, we see that communication remains very low (and in fact decreases a little). In both cases we keep epoch length to be $3W$ to maintain the same rate of change of β across the window.

Similarly, Figure 3.6d shows what happens when the window size is too small compared to the value predicted in Section 3.3.4. It depicts communication obtained on the fixed dataset, as a function of window size W. As window size decreases below $144m$ (see Section 3.4.3), constraint violations are more frequent as data periodically overwhelms the norms in Eq. (3.7). As we will see below, in practical settings a much lower W can be used, since data values have finite ranges, change slowly, and model changes are more frequent.

3.5.2 Traffic Monitoring

Consider the following interpolation problem: given periodical traffic measurements (average velocity every minute) from a small number of sensors embedded along a long road, we wish to infer the current average velocity at every point along the road. We aim to solve this problem using polynomial regression. Note that in this case we have no good way to measure the true error of our model, since we do not have sensors in other locations. Moreover, as Figure 3.1 (derived from the same data below) shows, monitoring model fit (R^2) is also problematic (Section 3.2.1). Instead, we rely on the fact that we can limit the model error $\|\beta - \beta_0\|$.

We used two weeks' worth of velocity data collected during November 2014 from $k = 6$ sensors located along the Grenoble south ring in France Morbidi et al. (2014); Canudas De Wit et al. (2015). Reported measurements of each sensor are aggregated once per minute, and when measurements are



(a) Window size $W = 60$

(b) Window size $W = 30$

Figure 3.8: Communication for DILSQ (black) and periodic algorithm (green) on the traffic dataset at different ϵ values.

unavailable average velocity was assumed to be unchanged. The road is composed of several sections, and we model it as the interval $[1, 18]$, where the sensors are located at $l \in \{1, 3, 7, 11, 14, 18\}$. Since we are doing polynomial regression, the data from every sensor at location l is always $x = [1, l, l^2, l^3, l^4]$, and y is the velocity measured by the sensor. Given model β built from measurements from the last hour ($W = 60$), the interpolated velocity at location $i \in [1, 18]$ is $[1, i, i^2, i^3, i^4]\beta$.

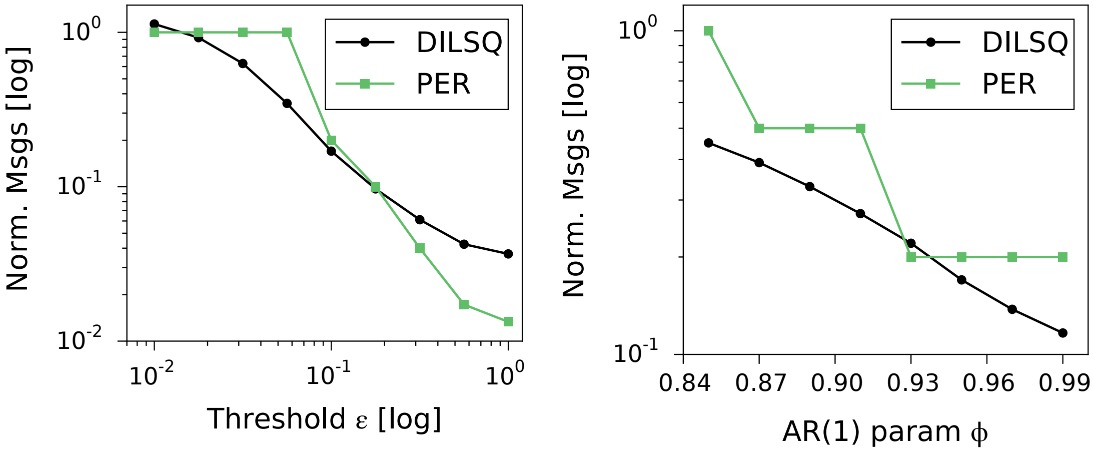
Figure 3.7 shows the result of one such a prediction for 9am on Nov 1 2014, produced using $\epsilon = 25$. The pink dots represent average velocity measurements of each sensor between 8am to 9am. The dashed purple line is velocity interpolated using the polynomial defined by the exact least squares model β, while the black line is interpolated using the polynomial from the DILSQ approximation β_0. Observe the resulting interpolation is fairly accurate, with errors below 10km/h across of range of interpolated positions[6].

Figure 3.8 explores the communication of DILSQ and matching PER with various levels of ϵ, for window sizes 60 and 30. DILSQ is superior to PER across all ranges except the unrealistically strict $\epsilon = 5$ (average $\|\beta\|$ is roughly 100). For one hour window, DILSQ obtains 0.12 normalized messages for $\epsilon = 25$ used in Figure 3.7, and can reduce communication to 0.03 for $\epsilon = 85$. For a much smaller window size of half an hour, DILSQ requires more communication but still achieves considerable communication reduction: it requires 20% communication for $\epsilon = 25$ and can use as little as 5% for $\epsilon = 85$. Finally, we observe that the communication gap between DILSQ and PER increases considerably with smaller window size, as β changes more quickly and is more sensitive to noise.

3.5.3 GLS on Gas Sensor Time Series

Data in this experiment consists of measurements collected by an array of 16 chemical sensors recorded at a sampling rate of 25Hz for 5 minutes, resulting in 7500 data points for each sensor. This dataset is described in Ziyatdinov et al. (2015), and is publicly available Lichman (2013). The original goal in Ziyatdinov et al. (2015) is to identify certain gas classes given high-level frequency features. Since

[6] Note there are many data points per location, and the curve tries to minimize error across them all. This, along with our use of a 4th degree polynomial, account for the smooth interpolation curve.



(a) Threshold ϵ (b) AR(1) parameter ϕ

Figure 3.9: Communication for DILSQ and periodic algorithm with same max model error at various ϵ and ϕ values for gas sensors data.

the original target variable is nominal and fixed throughout the run in each experiment, we defined a different regression problem. We divided the 16 sensors to $k = 4$ "nodes", where in each node three sensors serve as the data x while the remaining sensor serves as the response y. We also added a constant variable 1 to x, to allow intercept in the model, hence $m = 4$. The regression task is therefore to predict the value of the 4th sensor in each node using the first three.

Note that in this setting measurement errors cannot be assumed to be independent, so an OLS models is ill-suited here. Instead, we assume errors are an AR(1) process and monitor the generalized least squares model Hayashi (2000). We used an AR(1) parameter value $\phi = 0.95$ for the autocorrelation matrix Saudargienė (1999). Average $\|\beta\|$ is 0.3, so we use $\epsilon = 0.1$, resulting in 0.17 normalized messages for DILSQ. We note that using an OLS model with the same ϵ resulted in 1.15 normalized messages – the OLS model had to be updated very frequently as it was unstable.

Figure 3.9a shows the obtained communication for various ϵ values in the range [0.01,1]. For $\epsilon < 0.1$ DILSQ is clearly superior: PER must communicate every round ($T = 1$) in order to match DILSQ, which achieves communication between 0.2 and 1 (for $\epsilon = 0.01$). When ϵ is more permissive, however, PER is superior and can obtain the same maximum error with less communication: with an extremely permissive $\epsilon = 1$, DILSQ requires 0.04 normalized messages while PER requires 0.015 for the same maximum error (though, of course, optimal T must be known *a priori* to achieve this performance).

Figure 3.9b shows communication at different values of ϕ. DILSQ is almost always superior to PER by a large margin. Surprisingly, the optimal ϕ in terms of communication lies somewhere between 0.99 and 1 (but below 1, since we know OLS achieves poor performance). ϕ can be fitted using OLS regression of the residuals, and it would be interesting to see whether tuning ϕ for minimal communication results in the same value.

3.6 Related Work

Due to the ubiquity of linear regression, a great deal of research was dedicated to solving for the regression model not only in a centralized setting, but over distributed systems as well; for a comprehensive survey, see Sayed (2014). Typically, the distributed nodes compose a graph, each holding a portion of the data, and the goal is to solve for the regression model of the aggregated data. It is well-known that the accurate solution involves calculating a matrix-vector pair from the data (denote it A, c), and then calculating $A^{-1}c$. Since the global matrix-vector pair can be expressed as the sum of local pairs at the nodes, a path is defined over the graph, and the global pair is obtained by traversing this path; a *Hamiltonian path* is desirable, in order to reduce the time required to traverse the graph Lin et al. (2014). Spanning trees have also been applied to this end Paskin et al. (2005). Eventually, the local estimates at the nodes converge, via message passing with neighbors, to a global consensus Mateos and Giannakis (2012). In Tu and Sayed (2012) it was suggested that diffusion strategies outperform consensus-seeking methods.

Variants include taking advantage of the global matrix's sparseness in order to reduce traffic Guestrin et al. (2004), and gradient-based methods run either sequentially or with some degree of parallelism Mateos et al. (2010); Lopes and Sayed (2006); Sayed (2014); Yang and Brent (2004). Such techniques were also applied in online distributed learning, where the sought classifier can sometimes be expressed as the solution of a linear regression problem Zhang et al. (2014b).

While efficient solutions were developed for *computing* the linear regression model over distributed nodes, there are, to the best of our knowledge, only very few papers dealing with *monitoring* it – that is, imposing *local* conditions which imply that the *global* solution did not change by more than a predefined amount since the last time it was computed (Section 3.3). In Song et al. (2013), a heuristic is applied, and the nodes do not broadcast if the newly arriving data conforms with the current model up to some tolerance. In Bhaduri and Kargupta (2008) distributed monitoring was applied to monitor the prediction error (Section 3.2.1) and quadratic fit error R^2 Bhaduri et al. (2011), but not the error in the model itself. In Gupta et al. (2013), a one-dimensional regression problem is addressed – monitoring the ratio of two aggregated variables. We address the general, high-dimensional problem.

3.6.1 Distributed Monitoring

The last decade witnessed a sharp increase in work on imposing local conditions for monitoring the value of a function defined over distributed nodes. While the general problem is NP-complete Keren et al. (2014), considerable progress has been made for real-life problems. Most work dealt with the simpler cases of linear functions Keralapura et al. (2006); Kashyap et al. (2008), as well as monotonic functions Michel et al. (2005). Some papers addressed non-linear problems, e.g., monitoring the value of a single-variable polynomial Shah and Ramamritham (2008), and analysis of eigenvalue perturbation Huang et al. (2007). Jelasity et al. (2005) describes a gossip-based protocol for monitoring several aggregates (some non-linear), which eventually converges to the monitored value, but cannot guarantee user-specified error bounds. In Sharfman et al. (2007b) a geometric approach for monitoring arbitrary functions over distributed streams was proposed, and later extended and generalized Keren et al. (2012); Lazerson et al. (2015). However, nearly all work on geometric monitoring addressed functions which are either polynomials (typically quadratic), or defined by compositions of polynomial with simple functions such as medians and quotients. To the best of our knowledge, the problem addressed in this report – monitoring the linear regression *model* (as opposed to its *fit error*) – was never addressed over a distributed setting. Note that the monitored function contains the highly complicated operation of matrix inversion, which is not linear or convex, and which, when written explicitly, becomes intractable even for relatively low dimensions (e.g., the analytic expression for the inverse of a 20×20 matrix involves polynomials with 20! monomials). Therefore, a straightforward application of previous work on geometric monitoring is impossible.

3.7 Conclusions and Future Directions

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. Our distributed *monitoring* approach can easily be combined with an efficient distributed *computation* technique, enjoying the best of both worlds.

Consider the SPEEDD use case of traffic forecasting models. We have shown that even in the case that these models rely on many sources of data (e.g., all GPS devices in a city), it is possible to keep track of the predictor model and its matching to the incoming data streams. In this sense, we have shown that the monitoring system is scalable to as many input sources as needed. The method is also applicable to many other domains of application, as data sources become geo-distributed and ever increasingly rapid.

4 Conclusions

In this document we have described two 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. This result was accepted to publication in DEBS 2015 and received the Best Research Paper Award. The integration of these ideas into the SPEEDD architecture (using open source platform) is under way, and will take the rest of the project to implement and experiment with. We are also working on extending this new algorithmic framework into more elaborate queries, as well as other challenges such as multiple contiguous queries.

The second approach focuses 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. The paper describing this approach was published in the prestigeous conference KDD 2015. The implementation and integration of the technique will be completed towards the end of SPEEDD project, and will use real-life traffic SPEEDD data. The algorithmic breakthrough will be pursued to online monitoring of other interesting machine models, such as LDA-based classification.

Bibliography

http://www.eoddata.com.

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.

Z. D. Bai and Y. Q. Yin. Limit of the smallest eigenvalue of a large dimensional sample covariance matrix. *Ann. Prob.*, 1993. ISSN 0091-1798. doi: 10.1214/aop/1176989118. URL http://projecteuclid.org/euclid.aop/1176989118.

K. Bhaduri and H. Kargupta. An efficient local algorithm for distributed multivariate regression in peer-to-peer networks. In *Proc. SDM*, 2008. doi: 10.1137/1.9781611972788.14. URL http://dx.doi.org/10.1137/1.9781611972788.14.

K. Bhaduri, K. Das, and C. Giannella. Distributed monitoring of the R^2 statistic for linear regression. In *Proc. SDM*, 2011. doi: 10.1137/1.9781611972818.38. URL http://dx.doi.org/10.1137/1.9781611972818.38.

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.

C. Canudas De Wit, F. Morbidi, L. Leon Ojeda, A. Y. Kibangou, I. Bellicot, and P. Bellemain. Grenoble Traffic Lab: An experimental platform for advanced traffic monitoring and forecasting. *IEEE Control Systems*, 35(3):23–39, June 2015. URL https://hal.archives-ouvertes.fr/hal-01059126.

C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of xml documents with xpath expressions. *VLDB J.*, 11(4):354–379, 2002. URL http://dblp.uni-trier.de/db/journals/vldb/vldb11.html#ChanFGR02.

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.

C. Dousson and P. L. Maigat. Chronicle recognition improvement using temporal focusing and hierarchization. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 324–329, 2007. URL http://dli.iiit.ac.in/ijcai/IJCAI-2007/PDF/IJCAI07-050.pdf.

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, D. Keren, and A. Schuster. Communication-efficient distributed variance monitoring and outlier detection for multivariate time series. In *Proc. IPDPS*, 2014a.

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, 2014b. 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. Garofalakis, I. Sharfman, and A. Schuster. Prediction-based geometric monitoring over distributed data streams. In *Proc. SIGMOD*. ACM, 2012. URL http://dl.acm.org/citation.cfm?id=2213867.

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. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004. doi: 10.1145/1042046.1042051. URL http://doi.acm.org/10.1145/1042046.1042051.

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

C. Guestrin, P. Bodík, R. Thibaux, M. A. Paskin, and S. Madden. Distributed regression: an efficient framework for modeling sensor network data. In *Proc. IPSN*, 2004. doi: 10.1145/984622.984624. URL http://doi.acm.org/10.1145/984622.984624.

R. Gupta, K. Ramamritham, and M. K. Mohania. Ratio threshold queries over distributed data sources. *PVLDB*, 2013. URL http://www.vldb.org/pvldb/vol6/p565-gupta.pdf.

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.

F. Hayashi. *Econometrics*. Princeton University Press, 2000. ISBN 0691010188.

L. Huang, X. Nguyen, M. N. Garofalakis, J. M. Hellerstein, M. I. Jordan, A. D. Joseph, and N. Taft. Communication-efficient online detection of network-wide anomalies. In *INFOCOM*, 2007.

M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM TOCS*, 2005.

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, 2014a. doi: 10.1007/978-3-662-44848-9_40. URL http://dx.doi.org/10.1007/978-3-662-44848-9_40.

M. Kamp, M. Boley, D. Keren, A. Schuster, and I. Sharfman. Communication-efficient distributed online prediction by dynamic model synchronization. In *Proc. ECML PKDD*, 2014b. ISBN 978–3–662–44847–2.

S. R. Kashyap, J. Ramamirtham, R. Rastogi, and P. Shukla. Efficient constraint monitoring using adaptive thresholds. In *ICDE*, 2008.

R. Keralapura, G. Cormode, and J. Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *Proc. SIGMOD*, 2006.

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.

A. Knutson and T. Tao. Honeycombs and sums of Hermitian matrices. *Notices Amer. Math. Soc.*, 2001. URL http://www.ams.org/notices/200102/fea-knutson.pdf.

A. Lazerson, I. Sharfman, D. Keren, A. Schuster, M. N. Garofalakis, and V. Samoladas. Monitoring distributed streams using convex decompositions. *PVLDB*, 2015. URL http://www.vldb.org/pvldb/vol8/p545-lazerson.pdf.

M. Lichman. UCI machine learning repository, 2013. URL http://archive.ics.uci.edu/ml.

W. Lin, J. Cao, and X. Liu. E^3: Towards energy-efficient distributed least squares estimation in sensor networks. In *IWQoS 2014*, 2014. doi: 10.1109/IWQoS.2014.6914297. URL http://dx.doi.org/10.1109/IWQoS.2014.6914297.

C. G. Lopes and A. H. Sayed. Distributed adaptive incremental strategies: Formulation and performance analysis. In *Proc. ICASSP*, 2006. doi: 10.1109/ICASSP.2006.1660721. URL http://dx.doi.org/10.1109/ICASSP.2006.1660721.

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.

V. A. Marčenko and L. A. Pastur. Distribution of eigenvalues for some sets of random matrices. *Math. USSR Sb.*, 1967. ISSN 0025-5734. doi: 10.1070/SM1967v001n04ABEH001994. URL http://stacks.iop.org/0025-5734/1/i=4/a=A01?key=crossref.1d0b803ddab02373cb6b0690a61e734a.

G. Mateos and G. B. Giannakis. Distributed recursive least-squares: Stability and performance analysis. *IEEE Trans. Sig. Proc.*, 2012. doi: 10.1109/TSP.2012.2194290. URL http://dx.doi.org/10.1109/TSP.2012.2194290.

G. Mateos, J. A. Bazerque, and G. B. Giannakis. Distributed sparse linear regression. *IEEE Trans. Sig. Proc.*, 2010. doi: 10.1109/TSP.2010.2055862. URL http://dx.doi.org/10.1109/TSP.2010.2055862.

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.

S. Michel, P. Triantafillou, and G. Weikum. KLEE: a framework for distributed top-k query algorithms. In *Proc. VLDB*, 2005. ISBN 1-59593-154-6.

K. S. Miller. On the inverse of the sum of matrices. *Mathematics Magazine*, 1981. ISSN 0025570X. doi: 10.2307/2690437. URL http://www.jstor.org/stable/10.2307/2690437?origin=crossref.

F. Morbidi, L. Leon Ojeda, C. Canudas De Wit, and I. Bellicot. A new robust approach for highway traffic density estimation. In *ECC14*, 2014. URL https://hal.archives-ouvertes.fr/hal-00979684.

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.

M. A. Paskin, C. Guestrin, and J. McFadden. A robust architecture for distributed inference in sensor networks. In *Proc. IPSN*, 2005. doi: 10.1109/IPSN.2005.1440895. URL http://dx.doi.org/10.1109/IPSN.2005.1440895.

S. Roman. *Advanced Linear Algebra*, volume 135 of *Graduate Texts in Mathematics*. Springer, 1995.

S. Ronen, B. Gonçalves, K. Z. Hu, A. Vespignani, S. Pinker, and C. A. Hidalgo. Links that speak: The global language network and its association with global fame. *PNAS*, 2014. doi: 10.1073/pnas.1410931111. URL http://www.pnas.org/content/111/52/E5616.abstract.

M. Rudelson and R. Vershynin. Non-asymptotic theory of random matrices: extreme singular values. In *Proc. ICM*, 2010.

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.

A. Saudargienė. Structurization of the covariance matrix by process type and block-diagonal models in the classifier design. *Informatica*, 1999. URL http://iospress.metapress.com/index/NT17152864801166.pdf.

A. H. Sayed. Adaptive networks. *Proc. IEEE*, 2014. doi: 10.1109/JPROC.2014.2306253. URL http://dx.doi.org/10.1109/JPROC.2014.2306253.

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

S. Shah and K. Ramamritham. Handling non-linear polynomial queries over dynamic data. In *ICDE*, 2008.

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.

X. Song, C. Wang, J. Gao, and X. Hu. DLRDG: distributed linear regression-based hierarchical data gathering framework in wireless sensor network. *Neural Comput. Appl.*, 2013. doi: 10.1007/s00521-012-1248-z. URL http://dx.doi.org/10.1007/s00521-012-1248-z.

S. Y. Tu and A. H. Sayed. Diffusion strategies outperform consensus strategies for distributed estimation over adaptive networks. *IEEE Trans. Sig. Proc.*, 2012.

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.

R. Wolff, K. Bhaduri, and H. Kargupta. A generic local algorithm for mining data streams in large distributed systems. *IEEE Trans. Knowl. Data Eng.*, 2009. doi: 10.1109/TKDE.2008.169. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4604665.

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.

L. T. Yang and R. P. Brent. Parallel MCGLS and ICGLS methods for least squares problems on distributed memory architectures. *J. Supercomput.*, 2004. doi: 10.1023/B:SUPE.0000026847.75355.69. URL http://dx.doi.org/10.1023/B:SUPE.0000026847.75355.69.

H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD*, pages 217–228, 2014a.

Y. Zhang, D. M. Sow, D. S. Turaga, and M. van der Schaar. A fast online learning algorithm for distributed mining of BigData. *SIGMETRICS PER*, 2014b. doi: 10.1145/2627534.2627562. URL http://doi.acm.org/10.1145/2627534.2627562.

A. Ziyatdinov, J. Fonollosa, L. Fernández, A. Gutierrez-Gálvez, S. Marco, and A. Perera. Bioinspired early detection through gas flow modulation in chemo-sensory systems. *Sens. Actuators B Chem.*, 2015. ISSN 09254005. doi: 10.1016/j.snb.2014.09.001. URL http://linkinghub.elsevier.com/retrieve/pii/S0925400514010703.