Robust Complex Event Pattern Detection over Streams

by

Ming Li

A Dissertation
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy
in
Computer Science
by

March, 2010

APPROVED:

Prof. Murali Mani
Worcester Polytechnic Institute
Advisor

Prof. Elke A. Rundensteiner
Worcester Polytechnic Institute
Committee Member

Prof. Daniel J. Dougherty
Worcester Polytechnic Institute
Committee Member

Dr. Tao Lin
Amitive Inc.
External Committee Member

Prof. Michael Gennert
Worcester Polytechnic Institute
Head of Department

To my parents and grandparents.

Abstract

Event stream processing (ESP) has become increasingly important in modern applications, ranging from supply chain management for RFID tracking to real-time intrusion detection. In this dissertation, I focus on providing a robust ESP solution by meeting three major research challenges regarding ESP system robustness: *(1)* using a lightweight constraint-aware framework for event stream processing; *(2)* handling event streams with out-of-order data arrival and *(3)* handling event streams with interval-based temporal semantics. The following are the three corresponding research tasks completed by the dissertation:

Task I - Constraint-Aware Complex Event Pattern Detection over Streams. In this task, a framework for constraint-aware pattern detection over event streams is proposed. Given the constraint of the input streams, the proposed framework on the fly checks the query satisfiability / unsatisfiability using a lightweight reasoning mechanism. Based on the checking results, it adjusts the processing strategy dynamically by producing early feedbacks, releasing unnecessary system resources and terminating corresponding pattern monitor, thus effectively decreasing the resource consumption and expediting the system response on certain situations.

Task II - Complex Event Pattern Detection over Streams with Out-of-Order Data Arrival. In this task, a mechanism to address the problem of processing event queries specified over streams that may contain out-of-order data is proposed. Based on the analysis of the problems that state-of-the-art event stream processing technologies would experience when faced with out-of-order data arrival, a new solution of physical implementation strategies for the core stream algebra operators such as sequence scan, pattern construction and negation is provided.

Task III - Complex Event Pattern Detection over Streams with Interval-Based Temporal Semantics. In this task, an expressive language to represent the required temporal patterns among streaming interval events is introduced. Based on that, the corresponding temporal operator ISEQ and provide an efficient evaluation strategy for the proposed ISEQ operator is designed. For further improving the event processing performance, a mechanism to embed the "interval begin punctuation" into the interval stream is provided. Corresponding punctuation-aware query evaluation strategy is studied, which can greatly reduce the runtime memory and CPU footprint. A mechanism to push down the computation of interval event abstraction to the low level sensor network for increasing the computing leverage from the physical level devices is investigated.

Acknowledgments

There is a phrase I remember from some poem by Yoko Ono which says "draw a map to get lost". I think I am understanding this sentence a little bit better after spending quite some time in the past six years trying to draw out a little map, for a little little town in the United States of Database. But believe me, definitely I am not drawing it for getting lost... Well, I want to become a doctor of philosophy, not a philosopher :)

As to whether or not I did get lost with my little map, I guess only my PhD committee members are authorized to give an answer. But in any case, I would like to take this chance to express my gratitude to a couple people, who guiding me, mentoring me, helping me, giving money to me, feeding me, procrastinating with me, dating me, rooting for me and most importantly, believing in me, in the past six year while I was busily drawing.

I would like to express my gratitude to my advisor Prof. Murali Mani for everything he has done to make this dissertation possible. I deeply appreciate his great help and great patience in guiding my Ph.D. study at Worcester Polytechnic Institute.

My thanks go to the rest of my committee, Prof. Elke A. Rundensteiner, Prof. Daniel J. Dougherty and Dr. Tao Lin, who provided valuable feedback and suggestions to my PhD study, comprehensive-exam, dissertation proposal and dissertation drafts.

I am grateful for the help offered by other faculty and staff members from the Computer Science Department at WPI. Thanks a lot!

I thank all the lovely lovely DSRGers, who are fun, smart, energetic, creative, easygoing, hardworking, knowledgeable and all sharing the same belief that a week has seven days instead of only five. I am grateful to have the opportunity discussing all kinds of topics (research-related and gossip-related), enjoying all kinds of procrastination and sharing all kinds the food with such a wonderful group of data-oriented scholars.

My thanks also go to all the teachers who ever taught me during my kindergarten, primary school, middle school, high school, college and graduate school. OMG, I cannot believe I have attended that many that many that many different schools.

I am thankful for the financial supports I have received from my advisor, the department and NSF during my study and research in WPI. I would also like to thank my mentors, Dr. Bobbie Cochrane from IBM Software Group, Dr. Liangzhao Zeng from IBM T.J. Watson Research Center and Dr. Cirrus Shakeri from SAP Research Center for their help and advice throughout my internships during my Ph.D study. I thank my manager Nathan Abba from IBM, for the job offer at Silicon Valley Lab and for the great encouragement and supports. I also want to thank all my colleagues in the Master Data Management team at SVL, for helping me out with the MDM project at IBM (thus I could have more time to work on the dissertation) and cheering me up and encouraging me while I am feeling tired of the dissertation writing.

I would also like to thank a couple of “good old buddies” (called “Tie Ge Men” in Chinese) way back in my hometown city and in college. They are Guojun, Huicheng, Xiaoan, Dongheng, Jierong, Yingbin, Yuhua and Jiaxing from Zhuhai and Wenwei, Hongbin, Zhengchuan and all the 9502 folks from Fudan University. I feel really lucky for these precious friendships. I would never forget all the funs by fooling around with you guys, all the adventures taken with you guys, all the crazy and stupid stuff done together with you guys and all the supports and encouragement gotten from you guys!

I would like to thank my girlfriend Mo for her understanding and love during the past few years, especially during the time when I was far away in California. I thank my grandparents, my uncles and aunties and all my other relatives for their love and support. My parents receive my deepest gratitude. They have always believed in me and encouraged me. They don’t know too much about computer science (the statement still stays TRUE with the word “science” being removed), but they have been teaching me how important “knowledge” is since I was 1-year-old. I sincerely dedicate this dissertation to you, my dear parents.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Event Stream Processing

Recently the emergence of stream data processing had been extended to complex event processing on event streams. This research is generally called as *Event Stream Processing* (ESP). The motivation for these event processing systems comes from two directions. First, widespread deployment of cheap receptor devices such as wireless sensor networks and RFID technologies enables many new applications on the data streams collected from these receptors. Each data tuple from the sensors or RFID readers can be viewed as a primary event and been processed for monitoring and management purposes. The applications then can issue complex event patterns. Second, advanced applications require the existing publish/subscribe system supporting stateful filtering and propagation of the incoming messages. In such scenario, messages are viewed as business events. The routing of the messages then is determined by this message itself and its correlation with other previously messages. ESP enables applications such as algorithmic trading in financial services, RFID event processing applications, fraud detection, process monitoring, and location-based services in telecommunications. Below some applications based on the technique of complex event processing are listed:

Supply Chain Management [gar]. Business Activity Monitoring (BAM) has been described by Gartner [gar] as a technology that "allows business users real-time access to, and analysis of, important business indicators". One major BAM application is in supply chain management (SCM). ESP allows SCM to monitor, analyze, and act on the event flow regarding the produce procurement, order fulfillment and the transportation procedures.

Network Anomalies Detection [LTS$^+$08]. Assume a firewall server monitoring all the network packets between inside and outside machines. By analyzing the packet headers, the server can maintain the statistics of all the network flows. The statistics can include average bandwidth usage calculated from payload of each packets. A criteria can be set that when a series of flows each with 50% of the total network bandwidth, then an alert is set for potential network abuse.

Anti-Shoplifting [WDR06]. Assume a RFID tag is attached to every product in a retail store and RFID readers are installed at the shelves, checkout cashier and exits. Then a temporal sequence of RFID reader events: *TAKE_FROM_SHELF*, !*CHECKOUT*, *EXIT_FROM_STORE* means the occurrence of a shoplifting activity.

Events processing has been studied for more than a decade in the field of active databases, such as [GJS92][CKAK94]. See [PD99] for a survey on active database systems. Trigger-based active databases can respond automatically to events that are taking place either inside or outside the database system [WGB08][WBG08]. Traditional database systems are passive in the sense that commands are executed by the database in the form of queries. However, some applications expect more comprehensive facilities from databases for modeling part of the behavioral aspects of the application. This motivates the active databases that support moving the reactive behavior from the application into the DBMS. The benefit of using active databases lies in the centralized processing of database updates in a timely manner. Together with transaction techniques, active database systems can provide a stable, consistent and highly efficient framework for reaction-based applications.

To support reactive behavior, such application logic has to be encoded manually into the databases prior to the actual running. A common approach for active databases is to use rules that have up to three components: an event, a condition, and an action. Such a rule is known as an *Event-Condition-Action* (*ECA*) rule. The introduction of the ECA rules seem like an extension of the well-known trigger concept in DBMS. However, ECA rules can support widely different functionalities from using composed events, complex conditions and arbitrary actions. Active database systems in general include models of event detection, condition monitoring and action execution. Thus event processing ability is essential for such a trigger-based system.

Event detection and condition monitoring affect mainly the efficiency of the active databases. Several system implementations have been proposed either based on finite automata [GJS92], trees [CKAK94] or colored Petri nets [GD94]. However, the processing implementation is largely ad-hoc and suffers from high cost for arbitrary composition of events and conditions. This situation is even deteriorated when the event processing is mixed together with transaction management and the existence of multiple conflicting ECA rules. Although many efforts have been made respective to the active database system in the last decade, the performance of existing active database systems are still far away from the expectation, which largely limits their usability.

In the last two decades, the relational data model has gained popularity because of its solid mathematical foundation. However, the relational data model does not address the temporal dimension of data. Variation of data over time is treated in the same way as ordinary data. This is not satisfactory for applications that require temporal data-related operations. In fact, most applications require temporal data to a certain extent. Temporal databases provide query evaluation over persisted temporal data for such applications.

Compared with event processing applications based on active databases / temporal databases techniques, ESP applications have the following differences:

- Different from the active databases and temporal databases, where events are treated either as updates or persisted tuples of databases, ESP processes real-time event streams instead of persisted information in the databases. The business logic executed by the ESP is directly over the data generated from the physical device layer or business process layer.

- Active databases and temporal databases do not have a strict concept of window. They usually use transaction for expiration of target patterns. That is, all the events of database updates are valid only in one transaction or predefined several sequential transactions. However, Window constraints are usually applied to the ESP application for result filtering and state purge.

Traditional stream processing systems such as Aurora [ACC$^+$03], Stream [BW01], NiagraCQ [KNV03], TelegraphCQ [CCD$^+$03] and CAPE [RDS$^+$04] [WR09][WRGB06][LZR06][ZRH04] extend the relational query processing to stream data. Generally the select, project, join and aggregate operations are supported in these stream systems. Window-based constraint is also commonly used to processing blocking and stateful operations, such as grouping and join. Thus, the existing continuous query processing techniques in the traditional data stream systems can be extended to support ESP applications. However, in supporting ESP applications, the event-specific stream processing technology, which has an event-specific query evaluation mechanism, is shown to be superior to using the generic stream processing solutions because it is being specifically designed for handling event queries over streams [WDR06][GC$^+$07][GADI08] [SC$^+$09][MM09][BGHJ09][LRE09]. For example, SASE [WDR06][GC$^+$07] proposes an expressive language to support pattern queries on such sequential streams and proposes customized algebra operators for the efficient processing of such pattern queries with sliding windows. [ADGI08] extends [WDR06] to support Kleene closure over event streams. [ACT08] uses a plan-based technique to perform complex event detection across distributed sources. Cayuga [DGP$^+$07] designs an event-driven pub/sub system using automaton-based techniques for streaming pattern detection and [DCR$^+$08] provides optimization methods for efficient event processing.

1.2 Research Challenges

Event-specific stream processing is gaining more and more popularity in the industry [KF09][vAEE$^+$09][PV09][KJP09][Dem09][MRLD08][WAR08]. However the ESP research is still at an early stage. Some issues on system robustness have not yet been considered in the current research work on ESP. First, data stream applications are required to handle very large volume of real-time inputs and provide fast real-time system response continuously - thus lightweight runtime processing and minimized memory footprint play an important role in the robustness of event stream processing. Second, event streams are generated by different sources in different formats and they are sent through the ESP systems by different mechanisms in practice - thus a robust ESP engine needs to provide real-time support for complex event queries over event streams with flexible input semantics. For providing a robust ESP solution, we are facing the following three research challenges:

Challenge I - Lack of Mechanism in Lightweight Constraint-Aware Query Processing. Complex event processing over high speed streams may consume large amount of CPU and memory resources. For example, the ESP for financial transaction applications need to process thousands of incoming financial transactions per second in real-time. The efficiency of event processing in the ESP system is very critical for such application. This requires a high-performance query processing mechanism. Keeping large amount of primary events and performing intensive monitoring task which finally leads to no situation detection will be a big waste of system resources. In many practical cases business events are generated based on predefined business logic. Hence, in real-life event-based systems, constraints (such as occurrence and order constraints) often hold among events. These known constraints might help us to design a high-performance ESP mechanism which can terminate long running query processes at the earliest possible moment. Due to the real-time nature of ESP, such constraint-aware processing is required to be kept lightweight. An event processing mechanism which can efficiently exploit given event constraints to facilitate query processing over large volumes of event streams is still an open research question.

Challenge II - Lack of Mechanism in Handling Event Streams with Out-of-Order Data Arrival. For an event stream processing system if the order in which the events are received by the system is not the same as their timestamp order, we say the data arrival of the system is *out-of-order*. Most event systems [WDR06][ADGI08][ACT08][DGP+07][DCR+08] assume the event arrival is not out-of-order. By this assumption, the later arrival of an event implies that it has a larger timestamp than the other events which have already arrived earlier. However, out-of-order events are not uncommon in practice. Networking latencies and worst yet machine failures may all cause events to arrive out-of-order at the event stream processing engine. State-of-the-art event stream processing technology experiences significant challenges when faced with out-of-order data arrival including blocking, resource overflow, and incorrect result generation. We can illustrate that the existing technology would fail in such circumstances, either missing resulting matches or incorrectly producing incorrect matches. Clearly, for handling out-of-order data arrival, a more sophisticated mechanism is needed. Thus, an ESP mechanism which performs query evaluation over event streams with out-of-order data arrival remains a research challenge.

Challenge III - Lack of Mechanism in Handling Event Streams with Interval-Based Temporal Semantics. Consider monitoring applications such as intrusion detection, sensor-based activity tracking and network monitoring. Existing ESP engines have focused on detecting temporal patterns from instantaneous events, that is, events with no duration. However, such sequential patterns are inadequate to express the complex temporal relationships in domains such as medical, multimedia, meteorology and finance where the duration of events could play an important role. Due to the differences between the temporal patterns over interval events and point events, the query semantics and evaluation mechanisms used for pattern detection over point events is not sufficient for pattern detection over interval events. An expressive language to represent the required temporal patterns among streaming intervals is needed. Also, query evaluation mechanisms for such pattern queries need to be designed.

1.3 State of the Art

State of the Art for Challenge I. The constraint-aware query processing has been studied extensively in traditional databases, which does not meet the requirement of event stream processing application because they do not provide real-time solution for event processing [DCR+08]. Some work on XML stream processing engines [BCCN06][SRM05][KSSS04][WSL+06] [LMR08d][LMR08c][LMR08b][LMR08a] have looked at the schema-based optimization opportunity focusing on reducing CPU and memory footprint in XML data processing. However such techniques for handling of semi-structured data cannot be applied in ESP which is handling high volume streaming events. The focus of [WDR06][ADGI08] is on providing a customized algebra operators for handling event streams. In [ACT08] the authors mainly focus on providing handling pattern detection over event streams in a distributed environment. Thus constraint-aware processing mechanism is not within their consideration. [DCR+08] provides a constraint-aware ESP solution. However, it only considers a limited number of event constraint types instead of completely applying the whole given constraint knowledge. Even though a compile time precomputation mechanism is given to improve the runtime constraint inferencing process, this process still requires multiple state checking at runtime for every input event. Besides that, the abductive inference which is required at their compile time precomputation is NP-complete.

State of the Art for Challenge II. There has been some initial work of investigating the out-of-order problem for generic (homogenous-input) stream systems. One model, which is adopted for this dissertation, introduces the notion of K-Slack [BMM$^+$04]. Such solution is trivial in regular stream system as in fact the processing such as join proceeds as normal (with a K-delayed purging), and any tuple that arrives after K is simply discarded [HBR$^+$05]. A native approach [DGP$^+$07] on handling out-of-order event stream is using K-Slack as a priori bound on the input streams. It buffers incoming events in the input queue until ordering can be guaranteed. Compared with the proposed approach where each operator is order sensitive, such processing requires additional space and introduces more latency before allowing events to be evaluated. A second solution proposed to handle out-of-order data arrival is applying punctuations, namely, assertions inserted directly in the data stream confirming that for instance a certain value or time stamp will no longer appear in the future input streams [DMRH04][LMT$^+$05]. Such techniques, while interesting, require for some service to first be creating and appropriately inserting such assertions - hence here it is not considered further.

State of the Art for Challenge III. Previous research on pattern detection over event streams mainly focused on extracting temporal patterns from point-based event data [WDR06]. In [ACT08][DCR$^+$08][DGP$^+$07] the events are defined based on the interval model. However, only the "*before*" / "*after*" temporal relation is supported. The data mining community studied discovering patterns over interval events [KF00][PHL08][WC07]. [KF00] uses a hierarchical representation that extends Allens interval algebra [All83] for modeling temporal patterns over event intervals. However, this representation is lossy as the exact relationships among the events cannot be fully recovered. [WC07][PHL08] devise a lossless representation to overcome the drawbacks of [KF00]. Based on their proposed representation, they design corresponding mining algorithms for pattern discovery over event intervals. [PHL08] also examines how the discovered temporal patterns can be utilized in classification to differentiate closely related classes thus building an interval-based classifier. However, these works mainly focus on pattern discovery (mining) instead of pattern detection. Besides that, they do not consider streaming input with window constraints.

1.4 Dissertation Tasks

In my dissertation, I focus on the robustness of an ESP solution by meeting the three research challenges discussed in *Chapter 1.2*. The following are the three tasks:

Task I - Constraint-Aware Complex Event Pattern Detection over Streams. The goal of this task is to provide a light-weight constraint-aware pattern detection mechanism over event streams. It consists of the following subtasks:

- Providing a lightweight framework for event constraint reasoning.
- Providing the static and runtime query satisfiability / unsatisfiability checking algorithm based on automaton technique.
- Providing lightweight runtime optimization mechanisms through static precomputation of condition checking.

- Investigating an efficient execution strategy following the event-condition-action rule-based framework for constraint-aware pattern detection.
- Evaluating the proposed techniques.

Task II - Complex Event Pattern Detection over Streams with Out-of-Order Data Arrival. The goal of this task is to design a mechanism to address the problem of query processing over event streams that may contain out-of-order data. It consists of the following subtasks:

- Analyzing the problems which state-of-the-art event stream processing technology would experience when faced with out-of-order data arrival and studying the levels of correctness in out-of-order processing that target different priorities of applications.
- Designing a new solution of physical implementation strategies for the core stream algebra operators such as sequence scan, pattern construction and negation, including stack-based data structures and associated purge algorithms.
- Investigating optimization strategies for sequence scan, construction and negation as well as the corresponding state purge to minimize CPU cost and memory consumption.
- Evaluating the proposed techniques.

Task III - Complex Event Pattern Detection over Streams with Interval-Based Temporal Semantics. The goal of this task is to design an expressive language to represent the required temporal patterns among streaming interval events and corresponding query evaluation strategy. It consists of the following subtasks:

- Providing a case study of using interval events to optimize network event stream correlation.
- Introducing an expressive language to represent the required temporal patterns among streaming interval events, designing the corresponding temporal operator and providing an efficient evaluation strategy for the proposed operator.
- To further improve the event processing performance, providing a method to embed the "interval begin punctuation" (indicating the start of an upstream interval) into the interval stream. Studying the corresponding punctuation-aware query evaluation strategy.
- Providing a method to push down the computation of interval event abstraction to the low level sensor network for increasing the computing leverage from the physical level devices.
- Evaluating the proposed techniques.

Figure 1.1 shows the overall picture of the three dissertation tasks.

1.5 Dissertation Outline

The rest of this dissertation is organized as follows. *Chapter 2* gives the preliminary of this dissertation. The mechanism for lightweight constraint-

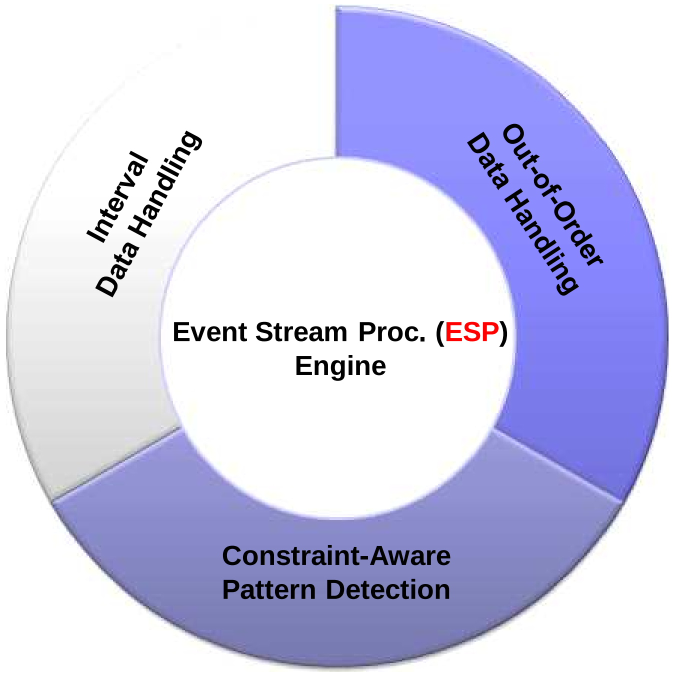

Interval Data Handling
Out-of-Order Data Handling
Event Stream Proc. (ESP) Engine
Constraint-Aware Pattern Detection

Figure 1.1: Dissertation Tasks

aware pattern detection over event streams is given in *Chapter 3* (**Task I**). *Chapter 4* discusses the proposed mechanism for query processing over event streams with out-of-order data arrival (**Task II**). *Chapter 5* proposes the framework for query evaluation over event streams with interval-based temporal semantics (**Task III**). Finally, *Chapter 6* discusses the solution integration and concludes this dissertation.

Chapter 2

Preliminary

2.1 Event Data Model

Event Instance. An *event instance* is an occurrence of interest in a system, which can be either primitive or composite as further introduced below.

Event Type. Similar event instances can be grouped into an *event type*. That is, each event type corresponds to a set of event instances. Event types describe a set of attributes that the class of event instances share. We use capitalized letters for event types such as E and we use lower-case letters such as e to represent event instances of event type E. An event type can be either a primitive event type or a composite event type[CKAK94]. *Primitive event types* are predefined in the application domain of interest. *Composite event types* are aggregated event types that are created by combining other primitive and/or composite event types.

Event Stream and Event Trace. An *event stream* is heterogeneously populated with event instances of different event types. In most event processing scenarios, it is assumed that the input to the system is a potentially infinite stream which contains all events that might be of interest [WDR06]. Such real-time input is referred to as an *event trace* (usually denoted as h), which evolves on the fly by receiving new instances as input. For an event trace h and an event type E, $E[h]$ denotes the set of all the event instances of E in h.

Temporal Aspects of Events. An event is associated with an unique timestamp, indicating the discrete ordering in the time domain. An event instance that happens instantaneously at a time point is called a *point event*. An event instance that occurs over a time interval is called an *interval event*. As a general representation for both the point and interval temporal semantics, for any event instance e, we use $e.ts$ and $e.te$ to denote the start and the end timestamp of the event instance e, respectively. The start and the end timestamps of a point event e are the same, which is simplified as $e.t$ (i.e., $e.ts = e.te = e.t$). For an event instance e, we use a pair of numbers as $e_{t1|t2}$ adjacent to it to represent its timestamp (denoting both the start and end time). For the point-based events, the representation is simplified as e_t, where the t adjacent to e denotes the time point when e happens. In this dissertation, we first assume events to be point-based (thus event e's timestamp will be represented as $e.t$) from *Chapter 2* to *4*. The handling of interval data and the solution integration for the support of intervals will be given in *Chapter 5* and *6*.

2.2 Event Query Model

Complex event pattern detection languages are studied in a number of existing works [ACT08][WDR06][DGP+07]. In this work I adopt the query language defined in [ACT08][WDR06] as follows to specify an *event pattern query*:

```
<Query>::= EVENT <pattern expression>
          [WHERE <equality conditions>]
          [WITHIN <window expression>]
```

The following I give a brief introduction to the semantics of the three clauses. For detail syntax and semantics of the query language, please refer to [ACT08][WDR06].

2.2.1 Semantics of the EVENT Clause

The **EVENT** clause specifies temporal and logical relationships among events. The semantics of the supported set of event composite operators, namely *SEQ*, *AND*, *OR*, and *NEGATION*, are provided below.

[**Sequence Operator**]. The sequence operator SEQ specifies a particular order in which the event instances of interest should occur and these event instances form a composite event instance. It takes a list of n ($n > 1$) event types as its parameters and outputs composite events $e = <e1\ e2\ ...\ em>$. The operator is defined as below:

$$SEQ(E1, E2, ..., Em)[h] = \{ < e1\ e2\ ...\ em > \mid (e1.t < e2.t ... < em.t) \wedge (< e1\ e2\ ...\ em > \in E1[h] \times ... \times Em[h]) \}. \tag{2.1}$$

Example 2.1. The following example illustrates the computation of *SEQ*(A, B) given the event trace h = "a_1, b_2, e_5, a_6, e_7". Remind that the small number adjacent to an event instance denotes the timestamp of the event. We have A[h] = { a_1, a_6 }, B[h] = { b_2 } and A[h] × B[h] = { <a_1 b_2> <a_6 b_2> }. The sequence result <a_1 b_2> satisfies the condition ($a_1.t < b_2.t$) (i.e., $1 < 2$). However, <a_6 b_2> is not a correct result for $a_6.t > b_2.t$ (i.e., 6 $\geq$ 2).

[**AND Operator**]. Conjunctive operator AND takes a list of n ($n > 1$) event types as its parameters. This operator specifies that all the sub-event types must occur for this event pattern. However, the sub-events can happen in any order. We can see that the SEQ operator is a restricted form of AND where events need to occur in order. The operator outputs composite events e = <*e1 e2 ... en*>. The operator is defined as below:

$$\begin{aligned}&AND(E1, E2, ..., Em)[h] = \\ &\quad \{< e1\ e2\ ...\ em > |(h_{ts} \leq MIN(ei.t_{i \in \{1,2,...,m\}}) \wedge \\ &\quad (MAX(ei.t_{i \in \{1,2,...,m\}}) \leq h_{te})) \wedge (< e1\ e2\ ...\ em > \\ &\quad \in E1[h] \times E2[h] ... \times Em[h])\}.\end{aligned} \tag{2.2}$$

[**OR Operator**]. Disjunctive operator OR takes a list of n ($n > 1$) event types as its parameters. It outputs a composite event when an instance of any of the expected event types occurs. The operator is defined as below:

$$OR(E1, E2, ..., Em)[h] = E1[h] \cup E2[h] \cup ... \cup Em[h]. \tag{2.3}$$

Example 2.2. For example, consider an event history h as { a_1, b_2, c_3, e_4, a_6, d_8 } and an event query as *OR*(B, C). *OR*(B, C)[h] = { b_2, c_3 }.

[**NEGATION Operator**]. Negation, denoted by "!", is applied to the events inside SEQ operators to express nonoccurrence of certain event patterns. For example, event pattern *SEQ*(A, !B, C) means finding all the matching of event pattern *SEQ*(A, C) that there is no B event instance between them. There are two special cases of negation which disallow events of certain event types appearing before and after a sequence. Detail description of these two cases can be found in [WDR06]. The following I give the definition of the negation operation which specifies that no event of a certain event type can appear between the two event sequences:

$$\begin{aligned}
&SEQ(E1, E2, ..., Ep, !Ek, Eq, ..., Em)[H] = \\
&\quad \{< e1\ e2\ ...\ em > | (< e1\ e2\ ...\ em > \in E1[h] \times E2[h] ... \times Em[h]) \wedge \\
&\quad (h_{ts} \leq e1.t) \wedge (em.t \leq h_{te}) \wedge (\nexists ek((ek \in Ek[h]) \wedge (ep.t < ek.t < eq.t)))\}.
\end{aligned} \tag{2.4}$$

Example 2.3. For example, consider an event history h as { a_1, b_2, c_2, b_3, e_5, a_6, e_7 }. We can see that *SEQ*(A, B)[h] ({ <a_1 b_2>, <a_1 b_3> }) = *SEQ*(A, C, B)[h] ({ <a_1 c_2 b_3> }) ∪ *SEQ*(A, !C, B)[h] ({ <a_1 b_2> }).

2.2.2 Semantics of the WHERE Clause

Most applications require real-time filtering, where users are interested in complex event patterns that impose additional constraints on the event instances [ACT08]. Such parameterized constraints between event attributes and specific values can be specified in the optional **WHERE** clause [WDR06]. One usual kind of equality condition checking is on the ID values (i.e., transactions IDs and RFIDs), such as EVENT *SEQ*(A, B) WHERE $A.id = B.id$. Such checking partitions the event history into sub-sequences. Each subsequence corresponds to one trace h. The query is then evaluated against each h. In the following discussion, we assume the event history is within an event trace. While customized predicate checking across multiple event types can be further accomplished[WDR06][BGAH07][ACT08], that is independent from the work in this dissertation. Thus we assume the predicate checking in an event query is only value-based comparison between constant values and specific attribute values of given event instances, such as EVENT $SEQ(A,\ B)$ WHERE $A.temperature$ = "37C". The *SELECT* operator introduced in [WDR06][ACT08] is the algebraic translation of the WHERE clause.

2.2.3 Semantics of the WITHIN Clause

Window-based processing is an essence in data stream processing systems. *Sliding windows* [BBMW02] are commonly used constraints to define the stateful operators in the traditional stream processing systems. In the traditional event query model such as [WDR06], the window constraint specification is given by a window parameter defined in the **WITHIN** clause. The time window argument specifies the maximum time duration between the occurrence of any two sub-events of a composite event instance. Hence, all the sub-events are separated by at most the time units defined by the window constraint. The *WIN* operator (sometimes referred to as the *WITHIN* operator) introduced in [WDR06] is the algebraic translation of the WITHIN clause.

2.3 Basic ESP System Architecture

Figure 2.1 shows the system architecture for a common ESP system. System architecture for each proposed solution given in *Chapter 3.4*, *4.6* and *5.5* will be based on this basic structure. The ESP system receives event through an *Input Adapter*, which connects to different kinds of data sources, such as system transaction datalogs, supply chain RFID readings, stock market data and e-commerce online transaction data. The ESP connects to two different output sockets, one is the *Result Monitor*, which consists within the *ESP Console*, the other is the *Output Adapter*, which relays output sequences to downstream receivers, such as different operational applications, spreadsheets, BI tools and BI dashboards. The ESP console also includes the *Query Register* for defining customized pattern monitor requirements. The *Plan Generator* parses and translates a given event query into an execution plan. The *Execution Engine*, which constructs results on the fly, is the key component of the ESP system. The definition and implementation of the query operators are contained by the *Operator Containers*, which includes the *Libraries* of the *Logical and Physical Operators*.

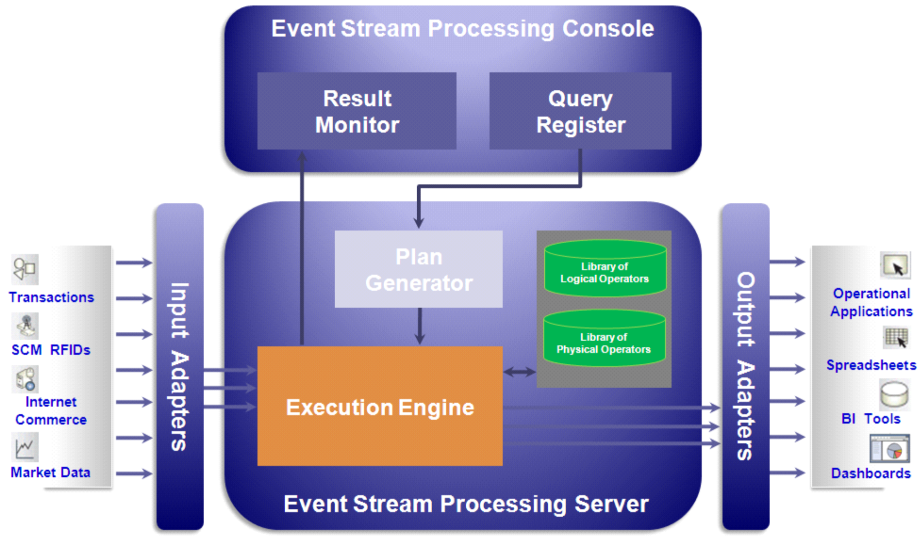


Figure 2.1: Basic ESP System Architecture

Chapter 3

Constraint-Aware Complex Event Pattern Detection over Streams

3.1 Introduction

Event stream processing (ESP) [WDR06][ACT08][Etz07][LLG$^+$09] technologies enable enterprise applications such as algorithmic trading, RFID data processing, fraud detection and location-based services in telecommunications. The key applications of the ESP technologies rely on the detection of certain event patterns (usually corresponding to the exceptional cases in the application domain). Alerts will be raised after the target pattern has been detected in the form of system notifications or triggers. Such functionality is sometimes referred to as the *situation alert*, which corresponds to many key tasks in enterprises computing.

In many practical cases business events are generated based on predefined business logic, shown by the following two scenarios:

Supply Chain Management. As we discussed earlier in *Chapter 1.1*, One major business activity monitoring (BAM) application is in supply chain management (SCM), which provides a flexible model to express business rules on top of a supply chain process. The business events corresponding to the stream-line logistics flow in SCM follow a predefined procedure.

Network Anomalies Detection. Assume a firewall server monitoring the network packets between inside and outside machines. The server can maintain the statistics of all the network traffic flows. Anomalies are detected from statistical data sent as event streams [ACT08], which are generated by workflow engines or simply customized programs following predefined schema.

In real-life event-based systems, constraints such as workflows often hold among the event data. For pattern detection over such event data streams, reasoning using the constraints enables us to *(1)* identify queries which are guaranteed to not lead to successful matches at the earliest, thereby helping us to terminate these long running pattern detection processes and release the corresponding CPU and buffer resources; *(2)* identify queries which can be guaranteed to surely lead to a future alert at the earliest (even though the matched result has not yet happened), thereby helping us to get prepared for upcoming situations. The above two are referred to as *detection of query* *unsatisfiability* and *detection of query satisfiability* separately.

Consider the following event query [WDR06][ACT08] in SCM, which monitors whether an item has passed several process steps of certain location in a certain order:

SEQ(SUPPLIER_WAREHOUSE, LABEL_CENTER, SHELTER)

Without given constraint knowledge of the input events, the earliest we can say that the expected pattern cannot be matched over the event trace is after the whole event trace has been completely received and still no match has been found. Similarly, the earliest a situation alert could be triggered is after a match of the expected pattern corresponding to the alert has been fully received. Assume we are given the event constraint as the product transportation workflow shown in *Figure 3.1.* By such semantics of the input stream, if the item's arrival at a *logistics center* is notified, we can guarantee that no match can be found for the expected pattern in a future, since no *shelter* could appear in the coming trace. Thus the pattern monitor can be terminated at this moment. Similarly, if the item's arrival at a *label center* following a *retail warehouse* is notified, we can guarantee that the current event trace can surely lead to a future match for the expected pattern, since a coming *shelter* following the *label center* is indicated by the workflow. Thus an early alert can be triggered for helping the corresponding party get prepared for upcoming situations.

I propose a framework for constraint-aware pattern detection over event streams and have implemented the proposed framework in a prototype system called E-Tec (constraint-aware query **E**ngine for pattern de**Tec**tion over

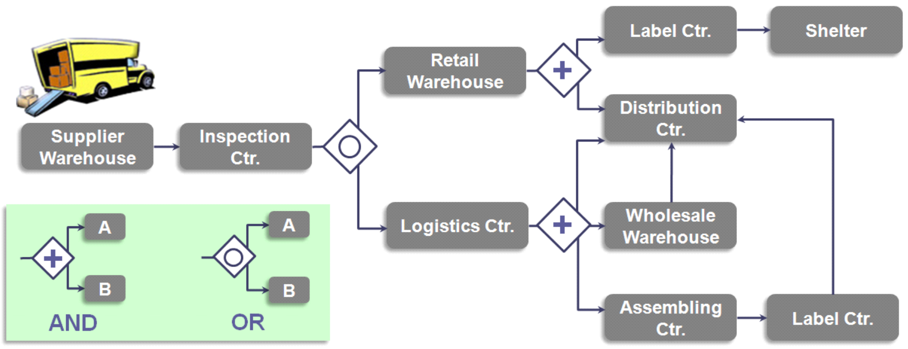


Figure 3.1: Example Workflow in SCM

event streams) [LMRL09b]. Given the constraint of the input event stream, E-Tec on the fly checks the query satisfiability / unsatisfiability using a lightweight reasoning framework. Based on such runtime constraint, E-Tec can adjust the processing strategy dynamically, by producing early feedbacks, releasing unnecessary resources (CPU and buffer) and terminating corresponding pattern monitor, thus effectively decreasing the resource consumption and expediting the system response on certain situation alerts. The proposed framework could be extended with window-based functionalities thus to support event pattern detection for queries with sliding windows. In this dissertation task, we will assume no window constraint on the pattern query. The contributions include:

- **Lightweight Constraint Checking.** Given the constraint of the input event stream at compile time, the query satisfiability / unsatisfiability is efficiently observed on the fly by E-Tec's *constraint engine* using the proposed checking algorithm. The process is made to be lightweight through decreasing the cost of runtime checking by applying the statically encoded information, which is precomputed using the proposed encoding algorithm. (*Chapter 3.2*)

- **Execution Strategy.** I propose a query execution strategy following the *Event-Condition-Action* (ECA) rule-based framework. Real-time streaming event data input serves as the *events*. The constraint engine described earlier uses the checking algorithm to determine whether a set of specific *conditions* are satisfied at runtime. Based on the checking results, corresponding *actions* are taken on the fly such as monitor termination, buffer releasing and early situation alerts. (*Chapter 3.3*)

- **System Integration and Experimental Studies.** The proposed constraint-aware pattern detection framework can be easily integrated with an automaton-based ESP engine by combing automaton applied for constraint checking with the automaton applied for pattern detection. A prototype system E-Tec is implemented following such design patter. Based on E-Tec, experimental studies are conducted to demonstrate that the proposed techniques bring significant performance gains in memory and CPU usage. (*Chapter 3.4*)

Recently the emergence of stream data processing had been extended to complex event processing on event streams [WDR06][ACT08][DCR$^+$08]. Wu [WDR06] proposes an expressive yet easy-to-understand language to support pattern detection over event streams, but constraint knowledge is not within the consideration of its query evaluation. In [ACT08], a plan-based technique is used to perform streaming complex event detection across distributed sources. Its focus is mainly on handling pattern detection over event streams in a distributed environment. A rule-based ESP solution is provided in [DCR$^+$08]. However, it only considers a limited number of rules instead of utilizing the complete input event constraint.

Roadmap. The rest of *Chapter 3* is organized as follows. *Chapter 3.2* provides the satisfiability / unsatisfiability checking algorithms. Execution strategy is studied in *Chapter 3.3*. Evaluations are presented in *Chapter 3.4*, followed by related work in *Chapter 3.5* and conclusions in *Chapter 3.6*.

3.2 Query Satisfiability and Unsatisfiability

3.2.1 Event Constraint

As discussed earlier, in many practical cases events are generated based on predefined constraint. In this work, we consider an event constraint C which can be expressed using a regular expression. For instance, C can be given as the event workflow of the input stream. $L(C)$ denotes the language defined by C. For any event trace h (which is assumed to be a finite stream), if h is the prefix of a sequence $k \in L(C)$, we call h being consistent with C. *Trace*(C) denotes the set which contains all the event traces that are consistent with C. Thus given a trace h, $h \in$ *Trace*(C) iff $\exists$ sequence k: $hk \in L(C)$. Obviously, $L(C) \subseteq Trace(C)$.

Example 3.1. Regular expression $A^+K^*B^+KC^+$ represents a given event constraint C_{exp1}, where consecutive A event instances, B event instances and C event instances are divided by a K event instances but the K's between the A's and B's is optional. Event trace h_1 = "a_1, k_2, b_4, b_7, k_8, c_9" and h_2 = "a_1, b_4, b_7, k_8, c_9" are both in $Trace(C_{exp1})$.

3.2.2 Satisfiability and Unsatisfiability Checking

An event trace is said to match event query Q if the evaluation of Q over h produces at least one matched pattern. For a pattern query Q, $L(Q)$ denotes the set which contains all the event traces that match Q.

Example 3.2. Consider event trace h_1, h_2 given in *Example 3.1* and event pattern query Q_{exp2} = EVENT $SEQ(A, K, K, C)$, which looks for event patterns with at least two K event instances appearing between an A instance and a B instance. Trace h_1 matches Q_{exp2} since in the trace there exist a complex event pattern instance $<a_1\ k_2\ k_8\ c_9>$ matching the target pattern $<a\ k\ k\ c>$. However, trace h_2 does not match Q_{exp2} since no instances of the target pattern could be found.

Given a pattern query Q, an event constraint C and a trace $h \in Trace(C)$, we want to determine whether a query match may exist for h while h keeps evolving at runtime. This problem regards to the checking of the *query satisfiability / unsatisfiability*, of which we give the definitions below.

Query Satisfiability. Given Q, C and $h \in Trace(C)$, Q is **satisfiable** iff $\forall\ k$: $hk \in L(C) \rightarrow hk \in L(Q)$. This is denoted as $Satisfiable(Q, C, h) = true$.

Query Unsatisfiability. Given Q, C and $h \in Trace(C)$, Q is **unsatisfiable** iff $\nexists\ k$: $hk \in L(C) \wedge hk \in L(Q)$. This is denoted as $Unsatisfiable(Q, C, h) = true$.

The key functionalities of event stream processing applications rely on the detection of certain event patterns (usually corresponding to the exceptional cases in the application domain). Alerts (such as system notification or triggers) are usually raised after the target pattern has been detected. Under the context of such *situation alerts*, checking the query satisfiability / unsatisfiability is equivalent to determining whether situation alerts will be raised in the future while a given event trace evolves on the fly. Given Q, C and $h \in \mathit{Trace}(C)$, Q either could be determined as satisfiable (i.e., $\mathit{Satisfiable}(Q, C, h)$ holds true) or unsatisfiable (i.e., $\mathit{Unsatisfiable}(Q, C, h)$ holds true), or could not yet be determined (i.e., both $\mathit{Satisfiable}(Q, C, h)$ and $\mathit{Unsatisfiable}(Q, C, h)$ are false, which means that whether a matched pattern may exist for Q while h evolves could not be decided yet).

Example 3.3. Consider Q_{exp2}, C_{exp1} given earlier and event trace h_3 = "a_1, k_2", h_4 = "a_1, b_4" and h_5 = "a_1, a_2". We have $\mathit{satisfiable}(Q_{exp2}, C_{exp1}, h_3)$ as true. This is because C_{exp1} guarantees one or more K instances will appear (i.e., the K's appearing before and after the consecutive B's) and then C events will appear after that. Thus, no matter how h_3 evolves, matched result(s) will surely appear in the future. For instance, a match is formed after h_3 evolving to h_1. Similarly, we knows that $\mathit{unsatisfiable}(Q_{exp2}, C_{exp1}, h_4)$ is true, since C_{exp1} indicates that only one K instance (i.e., the K appearing after the consecutive B's) will appear. We could also see that both $\mathit{satisfiable}(Q_{exp2}, C_{exp1}, h_5)$ and $\mathit{unsatisfiable}(Q_{exp2}, C_{exp1}, h_5)$ are false, since whether a match may exist could not yet be decided at the moment. How to determine satisfiability / unsatisfiability using algorithms for this example will be given later in *Example 3.5*.

A query could be statically determined as satisfiable / unsatisfiable before receiving any event input, i.e., the event trace h being an empty sequence. These two extreme cases are referred to as *static query satisfiability / unsatisfiability*.

Example 3.4. Consider C_{exp1} given earlier and Q_{exp4-a} = EVENT *SEQ*(A, K, C), Q_{exp4-b} = EVENT *SEQ*(A, K, D). Obviously we can guarantee the static query satisfiability of Q_{exp4-a} because C_{exp1} indicates the existence of instances such as $<a\ k\ c>$. Similarly, the static unsatisfiability of Q_{exp4-b} is guaranteed.

For an event constraint C, we let τ_C denote the minimized DFA for the language $L(C)$. Similarly, for a pattern query Q, we let τ_Q denote the minimized DFA for the language $L(Q)$. Construction of τ_C and τ_Q is described in [Koz03]. For a given DFA τ, We use $\mathring{s}\tau$ to represent τ's start state. The state transition function of τ used for processing a sequence input is denoted as $\hat{\delta}\tau$. $\hat{\delta}\tau(s,\ seq)$ denotes the state reached after finishing the transition of *seq*, beginning from a given state s in τ. $\{\hat{\delta}\tau(s,\ seq)\} = \emptyset$ if the transition falls out of τ. We use Ds to denote the derivative of a state s in a corresponding automaton, which is equivalent to the language accepted by the corresponding automaton starting from state s.

Two theorems are given below before showing the proposed algorithm on query satisfiability / unsatisfiability checking. Given query Q, constraint C and trace h, we have DFA τ_C, τ_Q defined earlier and we use $\tau_{\ddot{\cap}}$ to represent the FSA equivalent to $\tau_C \cap \tau_Q$, simply constructed as the cross product of τ_C and τ_Q. We let $\tau_\cap$ denote the DFA equivalent to the minimized DFA of $\tau_{\ddot{\cap}}$, however in $\tau_\cap$ during the minimization process we do not merge the automaton states in $\tau_{\ddot{\cap}}$ if they are mapped from different states from τ_C. Thus the states being merged during the minimization step are only the ones mapped from a same state at τ_C.

Below we explain the construction of $\tau_\cap$ in more detail. In the traditional algorithm for FSA minimization, we partition the set of states in FSA $\tau_{\ddot{\cap}}$ into a set of equivalence classes and we have one state in the minimized DFA corresponding to each equivalence class. Two states, x and y, in the FSA are said to be in the same equivalence class if $Dx = Dy$. Our specific minimization of $\tau_{\ddot{\cap}}$, denoted as $\tau_\cap$, is a variation of the traditional minimization algorithm. Note that each state in $\tau_{\ddot{\cap}}$ corresponds to a (s_C, s_Q), where s_C is a state in τ_C and s_Q is a state in τ_Q. Also, there is a state in $\tau_{\ddot{\cap}}$ corresponding to every such pair (s_C, s_Q). For a state corresponding to (s_C, s_Q) in $\tau_{\ddot{\cap}}$, we say that the mapping state in τ_C is s_C (similarly we say that the mapping state in τ_Q is s_Q). We partition the states in $\tau_{\ddot{\cap}}$ into equivalence classes as follows: two states, x and y in $\tau_{\ddot{\cap}}$ belong to the same equivalence class if $Dx = Dy$, and the mapping state in τ_C for x and y are the same. In other words, x corresponds to the pair ($s1_C$, $s1_Q$) and y corresponds to the pair ($s2_C$, $s2_Q$), then x and y belong to the same equivalence class if $Dx = Dy$ and $s1_C = s2_C$. Our algorithm also constructs one state corresponding to every equivalence class. Note that our minimization algorithm can result in more states in our minimized DFA than the traditional minimization algorithm. However, two states that are not merged in the traditional minimization algorithm will not be merged in our minimization algorithm as well. This implies the correctness of our minimization algorithm (i.e., the resulting FSA $\tau_\cap$ accepts the same language as the FSA that would result from the traditional minimization algorithm, which is the same as the language that would be accepted by $\tau_{\ddot{\cap}}$).

The reason for our specialized minimization is described later in *Chapter 3.2.3*, where we describe a light weight constraint checking mechanism, using the FSA resulting from our specialized minimization algorithm. For efficient algorithms of constructing and computing automaton derivatives, please refer to [Koz03].

Theorem 3.1. *Unsatisfiable*(Q, C, h) holds true iff $\{\hat{\delta}\tau_\cap(\mathring{s}\tau_\cap, h)\} = \emptyset$.

Proof.
$\Longleftarrow$: Given $\{\hat{\delta}\tau_\cap(\mathring{s}\tau_\cap, h)\} = \emptyset$, we know that h falls out of $\tau_\cap$. This implies $h \notin L(Q)$ since h is already in $L(C)$. Thus, there does not exist k: $hk{\in}L(C) \wedge hk{\in}L(Q)$. *Unsatisfiable(Q, C, t)* holds true by definition. □

$\Longrightarrow$: Given that *Unsatisfiable*(Q, C, h) holds true, we can not find a sequence k: $hk{\in}L(C) \wedge hk{\in}L(Q)$. Assume $\hat{\delta}\tau_\cap(\mathring{s}\tau_\cap, h) = s$, where s is an automaton state in $\tau_\cap$. Then there exist sequence $k' \in Ds$, $hk'{\in}L(C) \wedge hk'{\in}L(Q)$. By this we form a contradiction. □

Theorem 3.2. Assuming $\hat{\delta}\tau_\cap(\mathring{s}\tau_\cap, h) = p$ and $\hat{\delta}\tau_C(\mathring{s}\tau_C, h) = q$, *Satisfiable*$(Q, C, h)$ holds true iff Dp is equivalent to Dq.

Proof.
$\Longleftarrow$: The fact that Dp is equivalent to Dq implies that there does not exist

k: hk is in $L(C)$ but hk is not in $L(Q)$. *Satisfiable(Q, C, t)* holds true by definition. $\Box$

$\Longrightarrow$: Given that *Satisfiable*(Q, C, h) holds true, we know that for any k: $hk{\in}L(C) \rightarrow hk{\in}L(Q)$. Thus hk is in $L(C) \cap L(Q)$. Suppose Dp is not equivalent to Dq, there exists a sequence o: $k' \in Dq$ but $k' \notin Dp$ since $Dq \supseteq Dp$, which forms a contradiction. $\Box$

Based on the theorems, a query satisfiability / unsatisfiability checking algorithm shown in *Algorithm 1* is designed. *Line 6* to *13* give the static checking process. Q can be statically guaranteed as unsatisfiable if $\tau_\cap$ accepting only empty language (by *Theorem 3.1*) and Q can be statically guaranteed as satisfiable if $\tau_\cap$ is equivalent to τ_C (by *Theorem 3.2*). *Line 17* to *47* give the runtime checking process. Once an input event from the evolving trace *seq* triggers a state change in either $\tau_\cap$ or τ_C, derivative of the current state (precomputed in *Line 17*) of $\tau_\cap$ and τ_C will be compared. If they are equivalent, the satisfiability of Q can be guaranteed (by *Theorem 3.2*). If the transition falls out of $\tau_\cap$ (note that it could never fall out of τ_C because the input sequence is consistent with C), the unsatisfiability of Q can be guaranteed (by *Theorem 3.1*).

Example 3.5. Consider Q_{exp2} and C_{exp1} given earlier. *Figure 3.2* shows three automaton respectively: *(1)* $\tau_{C_{exp1}}$, *(2)* an equivalent NFA of $\tau_{Q_{exp2}}$ (instead of showing $\tau_{Q_{exp2}}$ for easier understanding) and *(3)* $\tau_\cap$ equivalent to $\tau_{C_{exp1}} \cap \tau_{Q_{exp2}}$. Let us first look at trace h_3 (given in *Example 3.3*). When the first event a_1 is processed, both $\tau_{C_{exp1}}$ and $\tau_\cap$ transit to state *s1* and the derivatives for these two states are $A^*K^*B^+KC^+$ and $A^*K^+B^+KC^+$

respectively, which are obviously not equivalent. When the second event k_2 is processed, both $\tau_{C_{exp1}}$ and $\tau_\cap$ transit to state *s2* and the derivatives turn to be equivalent as $K^*B^+KC^+$. Thus we can guarantee the satisfiability of Q_{exp2}. We then look at trace h_4 (given in *Example 3.3*) which is an example of query unsatisfiability. The transition falls out of $\tau_\cap$ when the second event b_4 is processed and unsatisfiability of Q_{exp2} can be guaranteed at this moment.

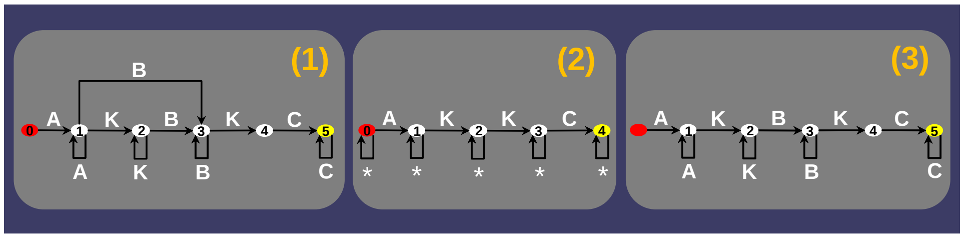


Figure 3.2: Example Automaton

3.2.3 Lightweight Constraint Checking

Checking the derivative equivalency between the states of $\tau_\cap$ and τ_C (polynomial time complexity) introduces runtime costs in *Algorithm 1*. It is conducted every time an input event instance triggers state transition(s) on either $\tau_\cap$ or τ_C, which could bring in big overhead for such runtime process. Besides that, *Algorithm 1* requires two automaton state lookups at runtime for each input event, i.e., the state lookups at $\tau_\cap$ and τ_C.

Below we introduce an optimized query satisfiability / unsatisfiability checking algorithm to decrease the runtime cost in *Algorithm 1*. Let us first

Algorithm 1 Query Satisfiability / Unsatisfiability Checking

```
1: **Procedure:** *SatUnsatChecking*
2: **Input:** (1) constraint $C$, (2) query $Q$, (3) real-time evolving sequence trace *seq* (must
   be consistent with $C$) as "$e1$, $e2$, $e3$ ..." received incrementally, with the End of Stream
   (EOS) message arriving at the very end if input termination is indicated
3: **Output:** static or runtime notification of query satisfiability / unsatisfiability
4:
5: ———————————————— **Static Checking** ————————————————
6: construct $\tau_C$, $\tau_\cap$ and precompute $D\mathring{s}\tau_\cap$ and $D\mathring{s}\tau_C$
7: if $\tau_\cap$'s accepted language $L(\tau_\cap) = \emptyset$ (i.e., $\tau_\cap$ being an empty automaton) then
8:     notify unsatisfiable and return
9: else
10:     if $\tau_C$'s accepted language $L(\tau_C)$ is equivalent to $L(\tau_\cap)$ (i.e., $D\mathring{s}\tau_\cap = D\mathring{s}\tau_C$) then
11:         notify satisfiable and return
12:     end if
13: end if
14: ————————————————————————————————————————
15:
16: ——————————————— **Runtime Checking** ———————————————
17: calculate the derivatives for all the states $\tau_C$ and $\tau_\cap$ except $\mathring{s}\tau_\cap$ and $\mathring{s}\tau_C$
18: var $p \leftarrow \mathring{s}\tau_\cap$
19: var $q \leftarrow \mathring{s}\tau_C$
20: var $p'$, $q'$
21: var *checkFlag* ← false
22: var $e \leftarrow$ *poll*(*seq*)
23: while $e \neq$ EOS do
24:     $p' \leftarrow \mathring{\delta}\tau_\cap(p, e)$
25:     $q' \leftarrow \mathring{\delta}\tau_C(q, e)$
26:     *checkFlag* ← false
27:     if $p' = null$ then
28:         notify unsatisfiable and return
29:     else
30:         if $p \neq p'$ then
31:             if $Dp'$ is equivalent to $Dq'$ then
32:                 notify satisfiable and return
33:             end if
34:             *checkFlag* ← true
35:             $p \leftarrow p'$
36:         end if
37:         if $q \neq q'$ then
38:             if !*checkFlag* then
39:                 if $Dp'$ is equivalent to $Dq'$ then
40:                     notify satisfiable and return
41:                 end if
42:             end if
43:             $q \leftarrow q'$
44:         end if
45:     end if
46:     $e \leftarrow$ *poll*(*seq*)
47: end while
48: ————————————————————————————————————————
```

look at a theorem given as follows, where $\tau_{\cap}$'s state set is denoted as $S\tau_{\cap}$ and τ_C's state set is denoted as $S\tau_C$.

Theorem 3.3. For any p in $S\tau_{\cap}$, there exists q in $S\tau_C$: for any sequence *seq*, $\hat{\delta}\tau_{\cap}(\mathring{s}\tau_{\cap}, seq) = p \rightarrow \hat{\delta}\tau_C(\mathring{s}\tau_C, seq) = q$.

Proof. The proof of this theorem is straightforward. By the automaton construction mechanism of $\tau_{\cap}$, we have guaranteed that for any state s in $\tau_{\cap}$, s maps to one and only one state in τ_C, which asserts the correctness of the given theorem. □

Remind that in the construction of $\tau_{\cap}$, the modified automaton minimization process skips merging the automaton states in $\tau_{\ddot{\cap}}$ (the cross product of τ_C and τ_Q) if they are mapped from different states from τ_C. From the above proof of *Theorem 3.3* we can see that the correctness of the theorem is based on such minimization mechanism, which guarantees each state in $\tau_{\cap}$ having one and only one mapping state in τ_C.

Example 3.6. Consider Q_{exp6} = EVENT $SEQ(OR(P|Q), A, B)$ and C_{exp6} = $PAB|Q{*}B|R{*}A$. *Figure 3.3* shows three automaton respectively: *(1)* $\tau_{C_{exp6}}$, *(2)* $\tau_{\cap}$ equivalent to $\tau_{C_{exp6}} \cap \tau_{Q_{exp6}}$ and *(3)* the minimized DFA of $\tau_{\cap}$, referred to as automaton $\tau_{\dot{\cap}}$. We use the automaton number plus the state label to distinguish each automaton state. For two different states s_a and s_b in $\tau_{C_{exp6}}$, if they map to state s'_a and s'_b respectively in $\tau_{\cap}$, we can guarantee that $s'_a \neq s'_b$. For example, we have state *(1)-0* maps to state *(2)-0*, state *(1)-1* maps to state *(2)-1*, ..., and state *(1)-4* maps to state *(2)-4*. Consider input event trace h_6 = "p_1", h_7 = "q_1" and h_8 = "q_1,

a_2". Trace h_6 reaches state *(2)-1* in $\tau_\cap$. By the precomputation, we already know that the derivative of state *(2)-1* is equivalent to the derivative of its mapping state (state *(1)-1*) in $\tau_{C_{exp6}}$. Thus we encode state *(2)-1* with some corresponding information. During real-time processing, only the $\tau_\cap$ automaton is run. After receiving trace h_6, we will reach state *(2)-1.* Based on its encoded information, we can then notify the query satisfiability . Similarly, query satisfiability can be notified after receiving trace h_8. For trace h_7, since the derivative of state *(2)-4* is not equivalent to its mapping state (state *(1)-4*), no information will be encoded for the state during the precomputation phase. So whether a matched pattern may exist for the query while the trace evolves could not be decided yet based on the input of h_7. Now we go back to look at $\tau_{\dot\cap}$ in *Figure 3.3*, which is the minimized DFA equivalent to the cross product of $\tau_{C_{exp6}}$ and $\tau_{Q_{exp6}}$. Without taking the distinguished minimization steps used in the construction of $\tau_\cap$, states in the cross product which are mapped from different states in $\tau_{C_{exp6}}$ could be collapsed together during the minimization. Thus, the property given in *Theorem 3.3* will no longer hold for $\tau_{\dot\cap}$. For example, states mapped from *(1)-1* and *(1)-4* in $\tau_{C_{exp6}}$ are combined into state *(3)-1* in $\tau_{\dot\cap}$. We cannot perform precomputation since *(3)-1* maps to multiple states in $\tau_{C_{exp6}}$.

The property given in *Theorem 3.3* guarantees the correctness of a run-time constraint checking mechanism given in the following *Algorithm 2.* Different from *Algorithm 1*, *Algorithm 2* achieves lightweightness in constraint checking by applying the *automaton encoding* before the runtime checking process. For each state p in $\tau_\cap$, its mapping state q in τ_C is found and

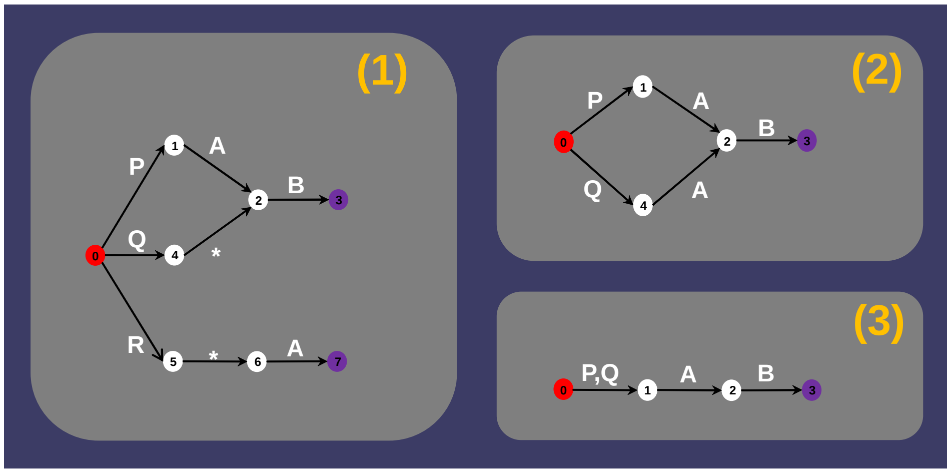


Figure 3.3: Automaton for Example Query with Lightweight Checking

derivative equivalency of p and q is checked. Then the corresponding checking result is encoded with p (*Line 9* in *Algorithm 2*). *Algorithm 3* depicts such automaton encoding process. The process has two components: the *traverser* and the *applier*. Each state in $\tau_\cap$ is associated with a variable *encoding* which is used to record the encoded information. By default the *encoding* value is set to be N/A for each state. The *traverser* traverses $\tau_\cap$ and directs the *applier* to each of its states. For a given state p, the *applier* calculates p's mapping state q in τ_C and performs the derivative comparison between p and q. If these two are equivalent, $p.encoding$ will be encoded as DER_EQUIVALENT. By using such encoded result on $\tau_\cap$, runtime cost in the runtime process is greatly decreased. Runtime checking of the derivative equivalency is completely replaced by a simple checking on the encoding value of the reached state in $\tau_\cap$ (*Line 20* in *Algorithm 2*). The query can be determined to be satisfiable while the encoding value

Algorithm 2 Lightweight Query Satisfiability / Unsatisfiability Checking

```
**Procedure:** *LightweightSatUnsatChecking*
**Input / Output:** same as *Algorithm 1* (*Line 2* to *3*)

——————————— **Static Checking** ———————————
Same as *Algorithm 1* (*Line 6* to *13*)
——————————————————————————————————

——————————— **Runtime Checking** ———————————
perform precomputation by running *Algorithm 3* as *AutomatonEncoding*($\tau_\cap$, $\tau_C$)
var $p \leftarrow \mathring{s}\tau_\cap$
var $p'$
var $e \leftarrow poll(seq)$
**while** $e \neq$ EOS **do**
    $p' \leftarrow \hat{\delta}\tau_\cap(p, e)$
    **if** $p' = null$ **then**
        notify **unsatisfiable** and **return**
    **else**
        **if** $p \neq p'$ **then**
            **if** $p.encoding$ = DER_EQUIVALENT **then**
                notify **satisfiable** and **return**
            **end if**
            $p \leftarrow p'$
        **end if**
    **end if**
    $e \leftarrow poll(seq)$
**end while**
——————————————————————————————————
```

is DER_EQUIVALENT. Besides that, running τ_C alongside with $\tau_\cap$ is no longer needed thus only one automaton look up at $\tau_\cap$ is required for each input event.

Example 3.7. Consider the same scenario as in *Example 3.5* but applying the lightweight constraint checking algorithm. The state *s2* to *s5* of $\tau_\cap$ are all encoded as DER_EQUIVALENT after applying *Algorithm 3.* For trace h_3, when the second event instance k_2 is processed, $\tau_\cap$ transits to state *s2*. Thus we are guaranteed the satisfiability of Q_{exp2} at this moment.

Algorithm 3 Automaton Encoding

```
1: Procedure: AutomatonEncoding
2: Input: (1) DFA $\tau_\cap$, (2) DFA $\tau_C$
3: Output: $\tau_\cap$ with encoded information on derivative equivalency checking
4:
5: calculate the derivatives for all the states $\tau_C$ and $\tau_\cap$ except $\mathring{s}\tau_\cap$ and $\mathring{s}\tau_C$
6: for all $p \in S\tau_\cap$ except $\mathring{s}\tau_\cap$ do
7:     find $p$'s mapping state $q$ in $\tau_C$
8:     if $Dp$ is equivalent to $Dq$ then
9:         $p.encoding \leftarrow$ DER_EQUIVALENT
10:    end if
11: end for
```

3.2.4 Handling Predicate-Based Filtering

As earlier discussion, most applications require real-time filtering, where users are interested in complex event patterns that impose additional constraints on the event instances. The proposed constraint-aware pattern detection framework supports predicate-based filtering on event streams using the same automaton-based mechanism introduced earlier. An example is shown as follows.

Example 3.8. Consider constraint $C_{exp8-a} = A^+B^+A^+C^+$, and query Q_{exp8} = EVENT *SEQ*(A, B, C) WHERE $A.id$ = "3". In order to fit into the automaton-based framework, we rewrite C_{exp8-a} into $C_{exp8-b} = (A[id{\neq}3]|A[id{=}3])^+B^+(A[id{\neq}3]|A[id{=}3])^+C^+$. DFA $\tau_{C_{exp8-b}}$ and $\tau_\cap$ (equivalent to $\tau_{C_{exp8-b}} \cap \tau_{Q_{exp8}}$) are given in *Figure 3.4 (1)* and *(2)* respectively. Take trace h_9 = "$a_1[id = 2]$, b_3" and h_{10} = "$a_1[id = 3]$, b_3" as example. For h_9, unsatisfiability of Q_{exp} can be guaranteed. For h_{10}, satisfiability of Q_{exp8} can be guaranteed instead.

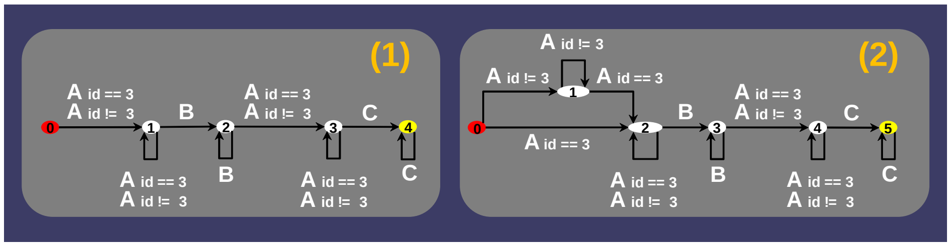


Figure 3.4: Automaton for Example Query with Predicate-Based Filtering

3.3 Query Execution

Pattern monitoring is a long running process for event pattern detection. In an execution strategy without considering constraint knowledge, the monitoring process could be stopped only when the event trace is terminated. Corresponding CPU and buffer resources could not be released earlier. During the monitoring process, situation alert will be raised while target event patterns has been detected. *Algorithm 4* given below sketches such *basic execution strategy*.

Algorithm 4 Basic Execution Strategy

```
1: Procedure: BasicExecution
2: Input: real-time evolving sequence seq as “e1, e2, e3 ...”, with the End of Stream (EOS)
   message arriving at the very end if input termination is indicated
3: Output: situation alerts and matched result sequences
4:
5: var $e \leftarrow poll(seq)$
6: while $e \neq$ EOS do
7:     process $e$:
8:     perform necessary data buffering and state purge, produce results and raise situation
       alerts if possible
9:     $e \leftarrow poll(seq)$
10: end while
11: terminate the pattern monitor for the current event trace
```

As earlier discussion, observation of the query satisfiability / unsatisfiability could be utilized in two aspects. First, it enables us to identify queries which are guaranteed to not lead to successful matches at the earliest, thereby helping us to terminate such long running pattern detection processes and release the corresponding CPU and buffer resources earlier. All the buffer taken for this trace can be released and no more CPU and memory footprint is required in the future on this trace. We call this process *early monitor termination.* Second, it enables us to identify queries which can be guaranteed to surely lead to a future alert at the earliest (even though the matched result has not yet happened), thereby helping us to get prepared for upcoming situations. We call this process *early situation alert.*

A *constraint-aware execution strategy* for complex event pattern detection over streams is thus proposed in *Algorithm 5*, by which the query satisfiability / unsatisfiability will be notified at the earliest possible moment during the execution to achieve both early monitor termination and early situation alert. The execution strategy follows the *Event Condition Action* (ECA) rule-based framework. It applies a constraint checker M using the checking algorithm (*Algorithm 1* or *2*) to notify the query satisfiability / unsatisfiability on the fly. Through the ECA framework, the real-time streaming event input serves as the **events**. The checking results from M serve as the **conditions** and the corresponding steps taken based on the checking result are seen as the **actions**.

To be specific, following benefits could be obtained through taking the corresponding actions of the early monitor termination and early situation alert under the proposed execution strategy:

- **Early Buffer Release.** By the early monitor termination, buffered elements can be released earlier.

- **Further Buffer Avoidance.** By the early monitor termination, no further event buffering is required for the current event trace.
- **Further Monitor Avoidance.** By the early monitor termination, no further pattern detection process is needed for the trace.
- **Taking Precaution Action for Upcoming Situations.** By the early situation alert, we can get prepared for upcoming situations at the earliest.

Example 3.9. Consider the same scenario as in *Example 3.5.* Let us first look at trace h_3. When the second event k_2 is processed, the constraint checker raises **satisfiable**. Thus, an early situation alert will be thrown at this moment for helping the corresponding parties to get prepared for upcoming situations at the earliest, even though the whole $<a\ k\ k\ c>$ pattern has not yet been formed. For trace h_4, the constraint checker raises **unsatisfiable** when the transition falls out of $\tau_\cap$ at the second event b_4. Thus a_1 (the first event in h_4) which was received and buffered earlier can be purged and no further buffering is required under this trace. Also, the pattern monitor can be terminated at this point in order to release corresponding CPU resources. Consider h_4 evolving to h_2 (given in *Example 3.1*). Pattern detection and buffering for extracting and keeping k_8 can be avoided through the early monitor termination.

Algorithm 5 Constraint-Aware Execution Strategy

```
1: Procedure: ConstraintAwareExecution
2: Input: (1) real-time evolving sequence seq as "e1, e2, e3 ...", with the End of Stream
   (EOS) message arriving at the very end if input termination is indicated, (2) procedure
   M for lightweight satisfiability / unsatisfiability monitor given in Algorithm 1 or 2
3: Output: situation alerts, matched result sequences and early situation alerts, with the
   early monitor termination functionality
4:
5: invoke M's static checking process
6: if M raises unsatisfiable then
7:     terminate the pattern monitor determination for the current event trace
8:     return
9: else
10:    if M raises satisfiable then
11:        raise early situation alert
12:    end if
13: end if
14: invoke M's runtime checking process
15: var $e \leftarrow poll(seq)$
16: while $e \neq$ EOS do
17:    pass e to M
18:    if M raises unsatisfiable then
19:        release buffer, perform early monitor determination for the current event trace
20:        pass EOS to M
21:        return
22:    else
23:        if M raises satisfiable then
24:            raise early situation alert
25:        end if
26:    end if
27:    process e:
28:    perform necessary data buffering and state purge, produce results and raise situation
       alerts if possible
29:    $e \leftarrow poll(seq)$
30: end while
31: terminate the pattern monitor for the current event trace
32: pass EOS to M
```

3.4 Performance Evaluation

3.4.1 System Implementation

E-Tec is implemented using Java 5. *Figure 3.5* shows the system architecture. Based on the basic system structure given in *Chapter 2.3*, the component of *Constraint Register* is plugged into the *ESP console* to provide the interface for constraint configuration. The *Execution Engine* and the corresponding components are equipped with the constraint handling ability: *(1)* the *Query Plan Generator* parses and translates a given event query into an execution plan, which includes a precomputed encoding; *(2)* the *Query Executor* takes in events from input streams and constructs results on the fly; *(3)* the *Constraint Engine* utilizes automaton-based technique to perform runtime constraint monitoring; *(4)* the *Execution Controller* receives feedbacks from the constraint engine and triggers the query executor to perform corresponding runtime actions.

The automaton-based model is commonly used by the state-of-the-art ESP engines. The proposed query satisfiability / unsatisfiability checking framework can be easily integrated with the automaton-based ESP engines by combing the monitoring automaton ($\tau_{\cap}$) with the automaton applied for pattern detection.

3.4.2 Experimental Setting

Experiments are run on two Pentium 4 3.0GHz machines, both with 1.98G of RAM. One machine sends the event stream to the second machine, i.e.,

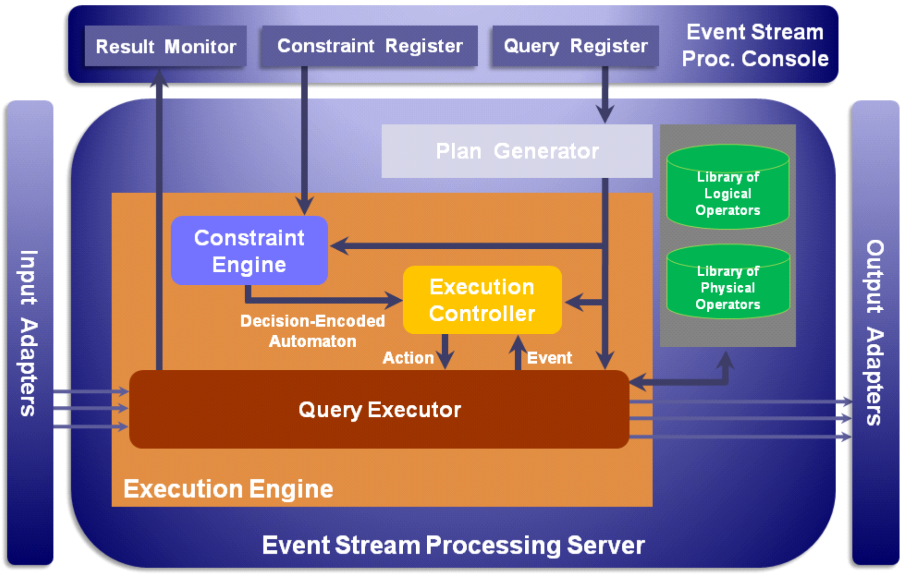


Figure 3.5: E-Tec System Architecture

the query engine. In *Chapter 3.4.3* and *Chapter 3.4.4* we will report the performance of the proposed constraint-aware techniques on a 5G data input based the supply chain data model given in [HG00], which contains multiple real life use cases on SCM. From its workflow, we can see that the data can be highly irregular, with 60% of the event types that can be optional or exclusive choices (used for controlling query selectivity). An on-line auction data which conforms to the schema used in XMark [SW02] can be an alternative data set for experiments.

Two sets of experiments are run. One is on event pattern queries with only pattern-based filtering, where the pattern-based selectivity is varied accordingly, which controls the percentage of patterns being filtered out through the query structure-related factors (Q_{exp2} as an example) from zero to 100% through changing the query complexity (state number of the query automaton in our case). The other set of experiments is on queries with only predicate-based filtering (Q_{exp8} as an example). In this test the pattern-based selectivity is 100%. However I also vary the predicate-based selectivity from zero to 100% through changing the predicate type and position.

3.4.3 Queries with Only Pattern-Based Filtering

Memory Consumption. The proposed constraint-based pattern detection technique should be able to minimize the amount of data that is buffered: with a smaller selectivity (less results being produced), more unnecessary data buffering could be avoided. The results shown in *Figure 3.6* provides the verification. X axis shows 6 groups of queries categorized by their pattern-based selectivities. Y axis shows the accumulative memory consumption for each query. We can see that the basic constraint checking (*Algorithm 1*) has the same buffer performance as the lightweight constraint checking (*Algorithm 2*) since they have the same effect on cutting memory consumption.

CPU Performance. *Figure 3.7* shows the query execution time. We can see that in most cases constraint-aware approaches outperform the naive approach without considering constraints: with a smaller selectivity, more unnecessary CPU computation could be avoided. However, when the selectivity is very high, constraint-aware approaches have poor performance because their overheads on runtime constraint checking become higher than the CPU saving through early monitor termination. Y axis here shows the execution time for each query. In the best case (i.e., the query for which

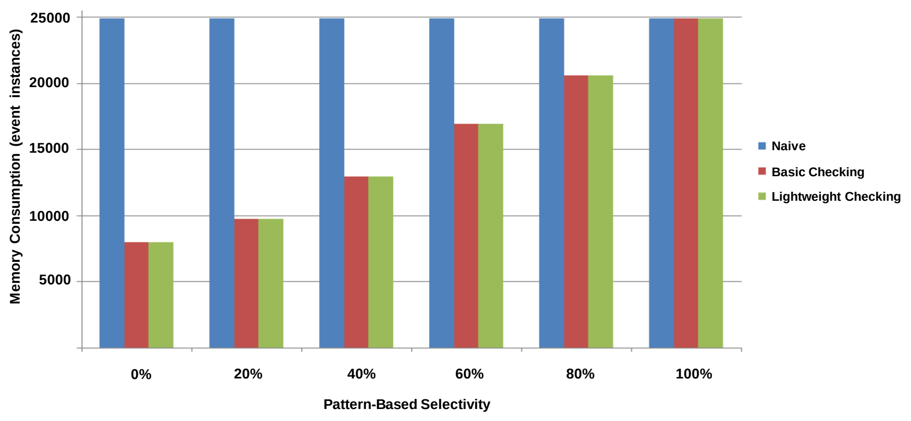


Figure 3.6: Results for Queries with Only Pattern-Based Filtering I

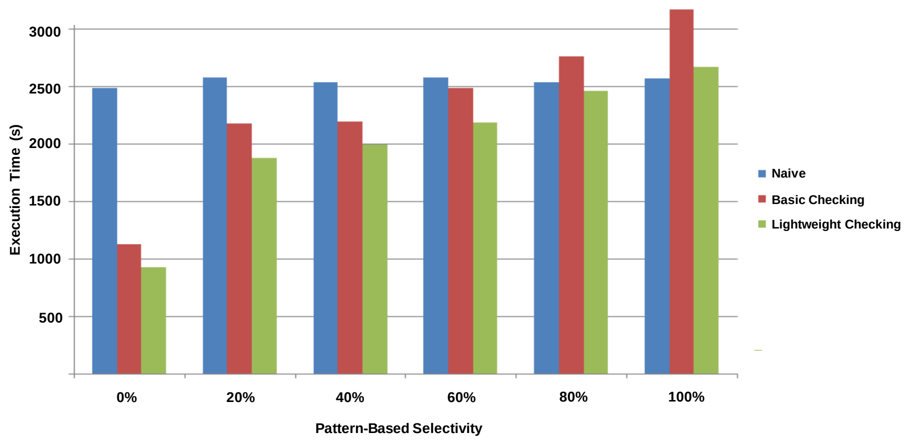


Figure 3.7: Results for Queries with Only Pattern-Based Filtering II

selectivity is 0%), plans optimized with constraint-based processing reduce the execution time of the original plan by 76%. We can also observe that the basic constraint checking does not perform as well as the lightweight constraint checking since the higher overhead from its runtime process.

Accumulative Query Determination Time. *Figure 3.8* shows the accumulative query determination time for each method, which is the accumulation of each input trace's execution time taken from the start of the trace till finally determining the result regarding the query satisfiability / unsatisfiability of the trace. For example, for a query which is determined as satisfiable at runtime before the trace ends, the determination time for it is the query execution time taken from starting the trace till a situation alert being raised. Constraint-aware approaches outperform the naive approach without considering constraints since the determination of the query being satisfiable / unsatisfiable comes before the trace being completely received and processed for most traces in the input. Y axis here shows such query determination time for each query. Plans optimized with constraint-based processing reduce the execution time of the original plan with similarly results (between 65% to 70%). Situation alerts raised by the system thus can lead to effective precaution action taking. We can also observe that the basic constraint checking does not perform as well as the lightweight constraint checking since its costly runtime process introduces higher overhead, similarly to the CPU performance results.

3.4.4 Queries with Only Predicate-Based Filtering

Experiments on memory and CPU consumption are also run for queries with only predicate-based filtering. Results with similar characteristics as in *Chapter 3.4.3* are reported, which are shown in *Figure 3.9*, *Figure 3.10* and *3.11*.

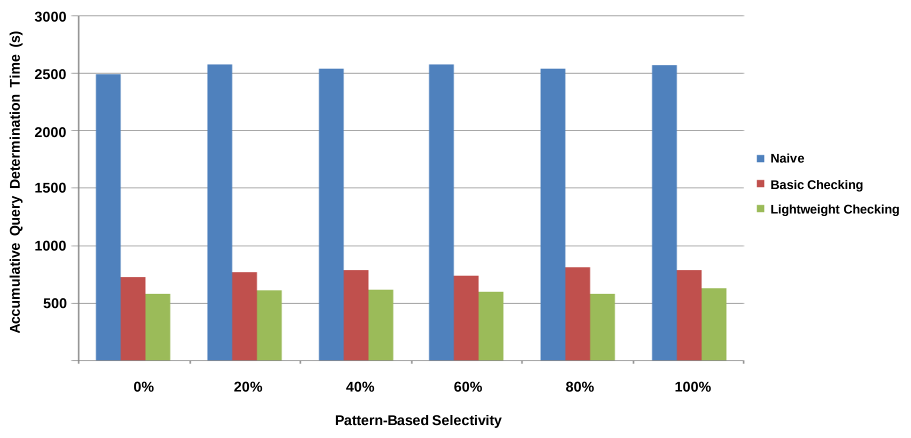


Figure 3.8: Results for Queries with Only Pattern-Based Filtering III

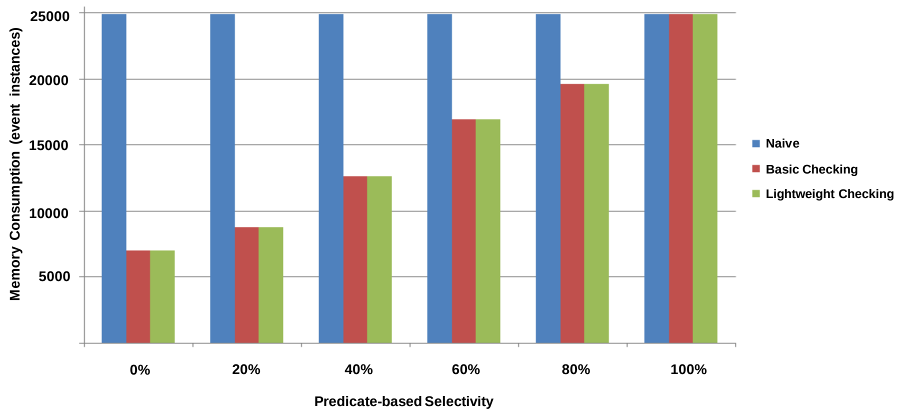


Figure 3.9: Results for Queries with Only Predicate-Based Filtering I

3.4.5 Conclusions of the Experimental Study

Above experimental results reveal that the proposed constraint-aware pattern detection framework is practical in two senses: *(1)* the technique can surely reduce the system memory consumption and *(2)* savings on CPU performance brought by the technique can be significant in most cases.

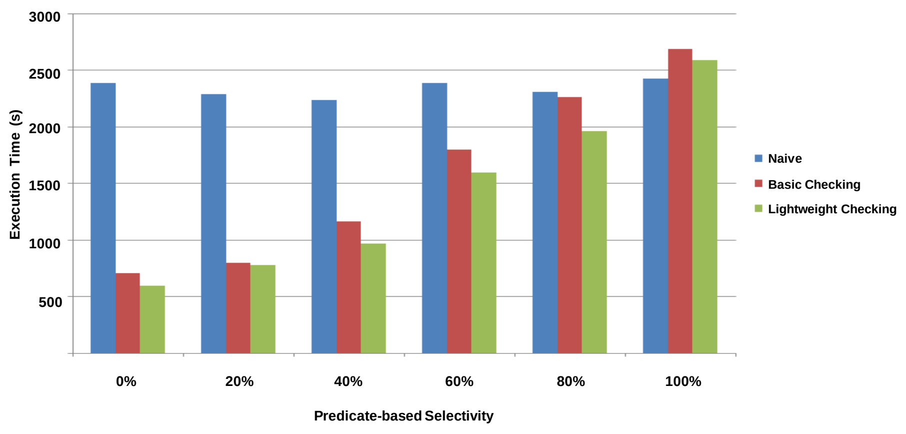


Figure 3.10: Results for Queries with Only Predicate-Based Filtering II

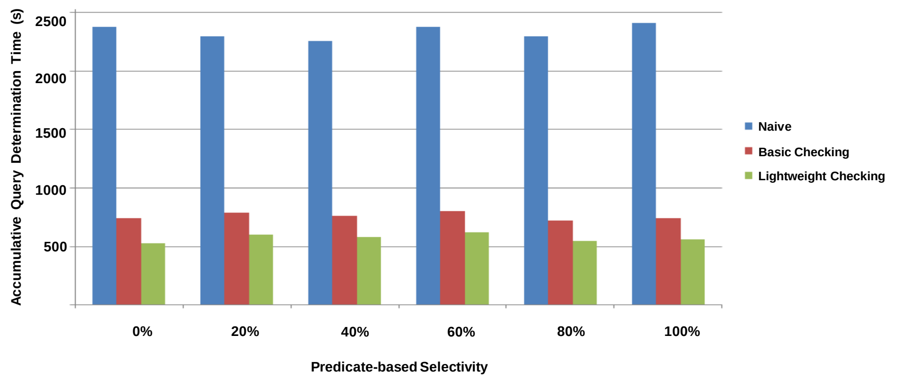


Figure 3.11: Results for Queries with Only Pattern-Based Filtering III

3.5 Related Work

The constraint-aware query processing has been studied extensively in traditional databases, which does not meet the requirement of event stream processing application because they do not provide real-time solution for event processing. XML stream processing work like [BCCN06][SRM05][KSSS04] [WSL$^+$06][LMR08d][LMR08c][LMR08b][LMR08a][WLRM06][WRML08] has looked at the schema-based optimization opportunity focusing on reducing

CPU and memory footprint in XML data processing. Such techniques for handling of semi-structured data cannot be applied in event stream processing which is handling high volume of real-time stream input in the format of heterogenous events. Event-specific ESP technology, which has an event-specific system design and evaluation mechanism, is shown to be superior to generic stream processing solutions [BW01][ACC$^+$03][CCD$^+$03] because it is being specifically designed for handling sequence queries over streaming event. An expressive yet easy-to-understand language is proposed in [WDR06] to support pattern queries on such sequential streams and proposes customized algebra operators for the efficient processing of such sequence queries with sliding windows. Constraint knowledge is not within the consideration of its query evaluation. A plan-based technique to perform streaming complex event detection across distributed sources is discussed in [ACT08]. Its focus is mainly on handling pattern detection over event streams in a distributed environment. In [SMMP09] and [MM09] CEP systems designed for query rewriting and distribution are proposed. These works do not consider constraints and they are following the traditional stream processing paradigm instead of the event-specific one for the purpose of distributed computing. A constraint-aware ESP solution is provided in [DCR$^+$08]. However, it only considers a limited number of event constraint types instead of completely utilizing the whole input constraint. Even though a compile time precomputation mechanism is given to improve the runtime constraint inferencing, this process still requires multiple state checking for every input event. Besides that, the abductive inference which is required at their compile time precomputation is NP-complete.

3.6 Conclusions

In many practical cases business events are generated based on predefined business logic. Hence, constraints often hold among event data. For pattern detection over event streams, reasoning using such known constraints enables us to identify the unsatisfiability and the satisfiability for a query at the earliest possible moment, thereby helping us to get prepared for upcoming situations at the earliest, thus helping us to effectively decrease the resource consumption and expedite the system response on certain situation alerts. In this dissertation task, I introduce a framework for constraint-aware pattern detection over event streams: *(1)* Given the constraint of the input event stream at compile time, the query satisfiability / unsatisfiability is efficiently monitored on the fly using my proposed lightweight runtime checking algorithm; *(2)* Following an ECA-based query execution strategy, I am able to adjust the processing strategy dynamically, by producing early feedbacks, releasing unnecessary resources and terminating corresponding pattern monitor; *(3)* I have implemented the proposed framework in the E-Tec prototype system and conducted experimental studies to illustrate that the proposed techniques bring significant performance improvement in both memory and CPU usage with little overhead.

Chapter 4

Complex Event Pattern Detection over Streams with Out-of-Order Data Arrival

4.1 Introduction

Event stream processing has raised increased interest in the communities of the database and distributed systems in the past few years [AE04][WDR06] [DGP$^+$07][SPL96]. A wide range and ever growing numbers of applications nowadays, including network monitoring, e-business, health-care, financial analysis and security supervision, rely on being able to process queries over data streams that take the form of time ordered series of events.

Let us consider a popular application for applying event sequence tracking techniques, namely, *anti-shoplifting*, which has been discussed earlier in

Chapter 1.1. Assume it is a bookstore which deploys the anti-shoplifting devices: RFID tags are attached to each book and RFID readers are placed at different locations throughout the store, such as book shelves, checkout counters and the store exit. If a book shelf and a store exit sensed the same book but none of the checkout counters sensed it in between the occurrence of the first two events, then we can conclude that this book is being shoplifted.

Event queries, such as those needed above to detect shoplifting, have been tackled in the literature. For instance, SASE [WDR06] proposes an expressive yet easy-to-understand language to support pattern queries on such sequential streams. It also proposes customized algebra operators for the efficient processing of such event pattern queries with sliding windows. This technology, being specifically designed for handling pattern queries over event streams, is shown to be superior to generic stream processing solutions [ACC$^+$03][BW01][KNV03][CCD$^+$03][RDS$^+$04].

For an event stream processing system if the order in which the events are received by the system is the same as their timestamp order, we say the data arrival of the system satisfies the *total order assumption.* Most systems [WDR06][ACT08], both event-based and stream-based ones, assume a total ordering among event arrivals. By this assumption, the later arrival of an event implies that it has a larger timestamp than the other events which have already arrived earlier. For example, the query evaluation approach of [WDR06] relies on such total ordering assumption for locating the expected event sequences.

However, out-of-order events are not uncommon in practice. For example, in a distributed computing environment, event sequences might arrive out of order at the processing engine due to network traffic and possible node failure. The existing technology would fail in such circumstances, either missing resulting matches or producing incorrect matches. Clearly, for handling out-of-order data arrival, a more sophisticated mechanism is needed. This is the problem I tackle in this dissertation task.

There has also been some initial work of investigating the out-of-order problem for generic (homogenous-input) stream systems. One model, which is adopted for this dissertation, introduces the notion of K-Slack [BMM$^+$04]. Such solution is trivial in regular stream system as in fact the processing such as join proceeds as normal (with a K-delayed purging), and any tuple that arrives after K is simply discarded [HBR$^+$05]. A native approach [DGP$^+$07] on handling out-of-order event stream is using K-Slack as a priori bound on the input streams. It buffers incoming events in the input queue until ordering can be guaranteed. Compared with the proposed approach where each operator is order sensitive, such process requires additional space and introduces more latency before allowing events being evaluated. A second solution proposed to handle out-of-order data arrival is applying punctuations [DMRH04][LMT$^+$05]. Such techniques, while interesting, require for some service to first be creating and appropriately inserting such assertions.

Contributions. I provide a solution framework for query evaluation over event streams with out-of-order data arrival in this task. The main contributions include:

- I analyze the problems that state-of-the-art event stream processing technology would experience when faced with out-of-order data arrival. (*Chapter 4.3*)

- I define different levels of correctness in out-of-order processing that target priorities of applications considering latency, output order, result correctness and result completeness. (*Chapter 4.4*)
- I provide new physical implementation strategies for the core stream algebra operators such as sequence scan, pattern construction, negation and the corresponding runtime purge. In particular, I introduce stack-based data structures as well as the associated sequence retrieval, event pattern construction, negation filtering and state purge mechanisms. (*Chapter 4.5*)
- Optimization for sequence scan, negation and state purge to minimize CPU cost and memory consumption are introduced. (*Chapter 4.5*)
- I conduct an experimental study that demonstrates the effectiveness of the proposed approach. (*Chapter 4.6*)

Roadmap. The rest of *Chapter 4* is organized as follows. In *Chapter 4.2* I give the overview of the SASE query algebra [WDR06]. Problems caused by the out-of-order data arrival are identified in *Chapter 4.3*. Different levels of "output correctness" is described in *Chapter 4.4*. In *Chapter 4.5*, I propose the solution of event stream processing with out-of-order data arrival. An experimental analysis is given in *Chapter 4.6*. Related work is discussed in *Chapter 4.7*. Conclusions for this dissertation task is given in *Chapter 4.8*.

4.2 Background

We assume here that the input event query has been translated into an algebraic query plan introduced in *Chapter 2*. The logical operators such as AND / OR (which detect patterns with logical relations) apparently would not be affected by out-of-order events. The SELECT operator (which performs value-based predicate checking) would be affected by out-of-order events only when it is associated with negation patterns. However that can be simply avoided by operator pushdown on the SELECT [WDR06]. Thus we only need to focus on the following operators: SEQ, NEGATION and WIN. Their corresponding physical implementations in SASE algebra are: SSC, NG and WD. SSC is formed by sequence scan (SS) and sequence construction (SC), and it contains functionalities pushed down from the window operator for the state purge operation. SS employs a NFA to detect matches to the event pattern specified in the query and SC constructs the expected event sequences based on events retrieved by SS. The WD operator checks whether events in the input event sequence occur within a sliding window. The NG operator handles the events in the queried event patterns that are preceded with the negation annotation ("!"), which is referred to as negative components. On the top of each SASE algebraic plan there is a TF (transform) operator, which handles the transformation of query results to composite event outputs.

Example 4.1. *Figure 4.1* shows an query plan for the query Q depicted in the same figure using the SASE algebra.

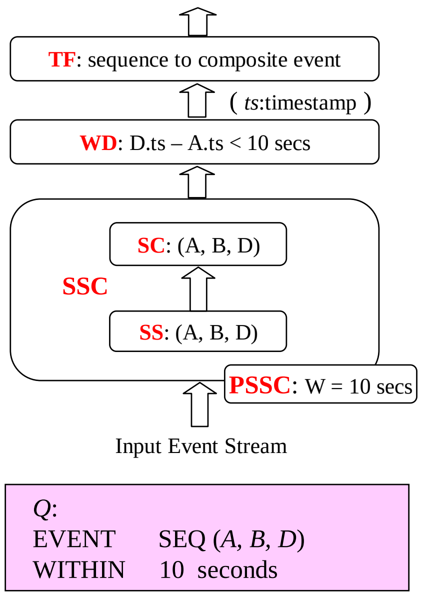


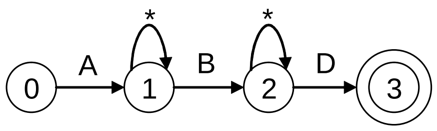
Figure 4.1: Event Query Plan

Sequence Scan and Construction (SSC). SSC as the bottom-most operator constructs a nondeterministic finite automaton. Let N denote the number of events in the query that are not involved in the negation query patterns. Then the number of states in the NFA equal to N + 1 (including the starting state). A data structure named *Active Instance Stack* (AIS) is proposed by [WDR06] for the execution of SSC. That is, instead of using a single stack for the NFA (*Figure 4.2(a)*), AIS associates a stack with each state of the NFA storing the events that triggered the NFA transition to this state. The events stored in each stack are called the active instances of this stack. In addition, for each active instance e in a stack, an extra field is created to record the *most recent instance in the stack of the previous state* (RIP).

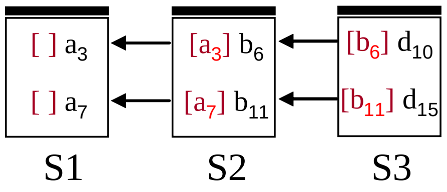
Figure 4.2(c) shows a partial input event stream. The events marked with an underscore are the ones being extracted during the sequence scan. All the retrieved events of type A, B and D are kept by AIS. *Figure 4.2(b)* shows the content of the three AIS stacks after the portion of stream S depicted in *Figure 4.2(c)* has been received. In each stack, the active instances are listed from top to bottom in the order of their arrival. Take the active instance b_{11} in stack S2 in *Figure 4.2(b)*. The most recent instance in stack S1 (holding event instances of type A) before b_{11} is a_7. The RIP field of b_{11} is thus set to a_7, as shown in the parenthesis preceding b_{11} in *Figure 4.2(b)*.

The sequence construction is initiated for each active instance of the accepting state, in our case, d_{10} and then d_{15}. With AIS, the construction is simply done by a depth first search in the DAG that is rooted at this instance and contains all the edges reachable from the root. Each root-to-leaf path in the DAG corresponds to one matched event sequence to be returned by this SSC operator. For example, the three event sequences created for the active instance d_{15} are $<a_3\ b_6\ d_{15}>$, $<a_3\ b_{11}\ d_{15}>$ and $<a_7\ b_{11}\ d_{15}>$. Thus, after receiving the events in the input stream S depicted in *Figure 4.2(c)*, the SSC operator should output four event sequences and then two of them will be removed by the WD operator. Totally there are two result sequences being produced, as shown in *Figure 4.2(d)*.

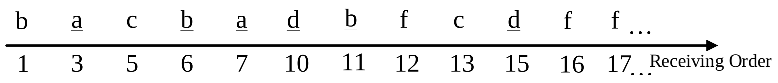
Purge at SSC (PSSC). State purge on SSC is conducted based on window constraints for removing outdated events from AIS. It considers the window constraint at the SSC operator, which can be seen as pushing down the windows down to the SSC. If the purge at SSC is conducted on a timely



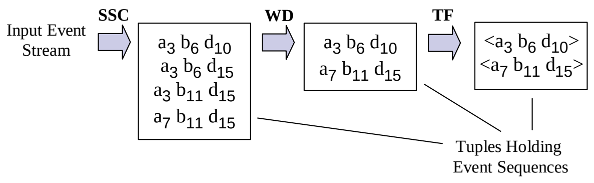
(a) Example Automaton



(b) SSC using Active Instance Stacks



(c) Input Event Stream

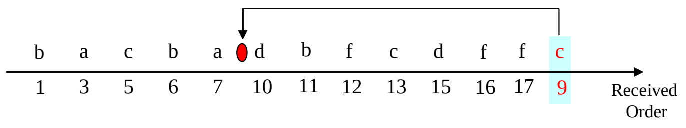


(d) Producing Result Tuples

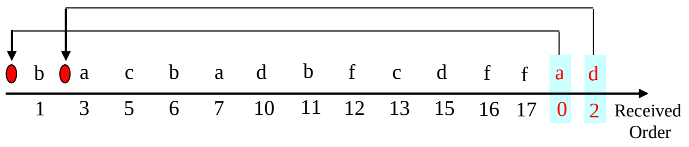
Figure 4.2: Query Evaluation of SASE

fashion, the window operator on top of the SSC can be removed. Event instances in AIS which fall out of the sliding window will no longer be able to contribute to the query result. PSSC dynamically prunes the event instances at AIS by removing such outdated events.

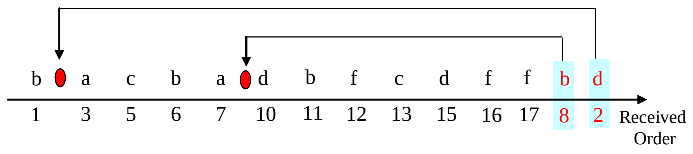
For example, when d_{15} is retrieved, a_3 can be removed from stack S1 because the distance between a_3 and d_{15} is already larger than the allowed



(a) Out-of-Order Event Arrival Example I



(b) Out-of-Order Event Arrival Example II



(c) Out-of-Order Event Arrival Example III

Figure 4.3: Out-of-Order Data Arrival Examples

window range $(15 - 3 > 10)$. Similarly, once f_{17} is received, b_6 can be safely pruned from S2 at AIS because it has slid out of the window $(17 - 6 > 10)$.

Negation (NG). The negation operator handles the negative components of a pattern sequence construction. Events under such negative components ignored during the SSC are collected in a buffer. We refer it as the negation buffer in this dissertation. In the above example, the C events from the input stream will be kept in the negation buffer. NG checks for each input event sequence whether there exist any C events in the negation buffer that arrives between the B and D events in the located event sequences. For example, when d_{15} is received, there are two C events kept in the negation buffer (namely, event instance c_5 and c_{13}). The second tuple $(a_7\ b_{11}\ d_{15})$

input to the NG operator will be removed from its output, because that there exist a C event (c_{13}) between b_{11} and d_{15}.

Purge at NG (PNG). The negation buffer is another in-memory data structure maintained by SASE. Similarly to AIS, windows constraint-based data purging needs to be conducted for removing outdated events from the negation buffer when memory resources are limited. In [WDR06] this process is seen as garbage collection on negative events. In this dissertation we model it as an operator, referred to as PNG (Purge at Negation). Event instances in the negation buffer which fallen out of the sliding window will no longer contribute to the query result. PNG dynamically prune the negation buffer by removing such outdated events. For example, when f_{16} is retrieved, c_5 can be removed from the buffer because that the distance between c_5 and f_{16} is already larger than the allowed window range ($16 - 5 > 10$).

4.3 Problems Caused by Out-of-Order Events

4.3.1 Out-of-Order Event Stream

SASE approach assumes a total ordering of all event arrivals, i.e., the order in which the events are received by the query system equal to their timestamp order. The query evaluation approach of SASE relies on this total order assumption for identifying event sequences. However, as mentioned in *Chapter 4.1*, if the input stream were to contain any out-of-order events, such handling approach becomes insufficient for event query evaluation.

Out-of-Order Event. For a newly arrived event e, supposed the events that we received before e are "*e1*, *e2*, *e3*, ..., *em*", if there exists any *ei* among "*e1*, *e2*, *e3*, ..., *em*" satisfying e.timestamp $<$ *ei*.timestamp, e is an *out-of-order event*.

In the example stream S shown in *Figure 4.3(a)*, the events are listed under their received order. We can see that event c_9 received after event f_{17} is an out-of-order event. The input event stream no longer satisfies the total order assumption. The out-of-order event c_9 should have arrived at the position indicated by a dot above the axis.

4.3.2 Problem for Sequence Scan and Construction (SSC)

Incomplete Event Retrieval. The current execution logic of NFA in SSC [WDR06] relies on the total ordering assumption. If this assumption no longer holds, some events which should have been kept might be discarded by the sequence scan. We refer to this as *incomplete event retrieval*.

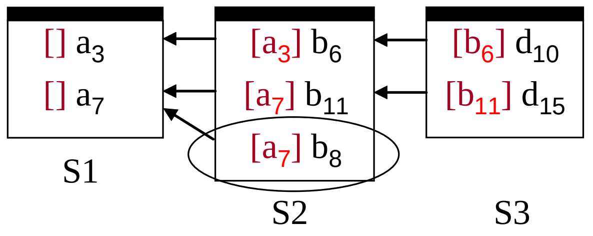
Example 4.2. Consider the example event stream in *Figure 4.3(b)*. Two out-of-order events, a_0 and d_2, came after f_{17}. The dots in the figure indicate the positions at which these two out-of-order events should have arrived under the event timestamp order. We can see that $<a_0\ b_1\ d_2>$ is an event sequence which should be constructed by the SSC. However, during the event retrieval of SSC by using NFA, when b_1 arrives, automaton state s_2 hasn't been activated yet. Hence, b_1 will simply be discarded. At the moment when the a_0 and d_2 are received, the event b_1 is gone. Thus the sequence $<a_0\ b_1\ d_2>$ is missed.

From the above example we observe that such incomplete event retrieval potentially causes some qualified event sequence being missed.

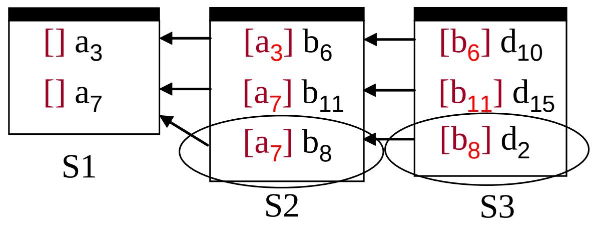
Event Misplacement. The retrieved events during the sequence scan will be placed in AIS for event sequence construction. Based on the total order assumption, newly arriving events are placed on top of the corresponding stack in AIS. For example, when a_7 is retrieved, it will be put on top of stack S1. With out-of-order event inputs, located events might be placed into the wrong spot in AIS during sequence scan by this simple "append" approach. We refer to this as *event misplacement*.

Example 4.3. Still consider the example event stream in *Figure 4.3(c)*. Assume the out-of-order event, b_8 and d_2 arrive after f_{17}. The dots above the axis show the position where b_8 should have arrived under the event timestamp order. If evaluating correctly, one candidate event sequence, $<a_7\ b_8\ d_{10}>$, should be produced after receiving b_8. However, by simply appending b_8 to stack S2, b_8 will be placed under b_{11}, with the RIP field set to a_7 (*Figure 4.4(a)*). The event sequence $<a_7\ b_8\ d_{10}>$ thus would never be constructed. Similarly, by simply appending d_2 to stack S3, d_2's RIP will be pointing to the newly appended b_8 (*Figure 4.4(b)*). Thus incorrect sequences such as $<a_3\ b_6\ d_2>$ and $<a_3\ b_{11}\ d_2>$ will be produced in the sequence construction.

From the above example we observe that such event misplacement potentially causes the SSC operator to miss event sequences and to produce incorrect event sequences.



(a) Incorrect AIS Update when b_8 Arrives



(b) Incorrect AIS Update when d_2 Arrives

Figure 4.4: Event Misplacement in AIS

4.3.3 Problem for Negation (NG)

To determine whether an event sequence satisfies the negation requirement, the execution of the negation operator utilizes the events kept in the negation buffer at the moment the new input tuple to the operator is scheduled to be consumed. Once a new event sequence is constructed by the SSC, it will be passed to the upper operators for further processing. For example, consider evaluating for the query *SEQ*(A, B !C, D) WITHIN W over the data given in *Figure 4.2(c)*. When d_{10} is met, event sequence $<a_3\ b_6\ d_{10}>$ will be produced and passed up to the Window (WD) operator and then the NG operator, as the given SASE query plan shown in *Figure 4.2(b)*. At this moment, there is only one event instance kept in the negation buffer c_5).

Negation event c_5 is not in the range of b_6 and d_{10}. Thus the event sequence $<a_3\ b_6\ d_{10}>$ does not get removed by the NG. It is passed up to the TF and then output. Under the total order assumption, this works correctly. However, with out-of-order events coming potentially in the future, such output event sequence is no longer guaranteed to be correct. Let's first look at the following example.

Example 4.4. Suppose that the out-of-order event c_9 comes right after f_{17}, as shown in *Figure 4.3(a)*. Obviously, the appearance of c_9 makes the output $<a_3\ b_6\ d_{10}>$ no longer a valid answer. It should not have been produced by the NG operator.

We call the event sequence $<a_3\ b_6\ d_{10}>$ in the above example spurious sequences because that at the moment it is being output by the NG, we cannot guarantee its correctness regarding to the functionality of the negation semantics. We refer such problem in NG as *producing spurious sequences.* Spurious sequences potentially will turn to an invalid output of the NG operator, if certain out-of-order events indeed arrive later. Intuitively we can see that in a sequence query with negation components, the NG operator can never produce any data event sequence which is guaranteed to be "correct", as shown by *Theorem 4.1.*

Theorem 4.1. In a sequence query with negation components, every event sequence output by the NG operator is a spurious event sequence unless the total order on the data arrival holds for the input stream.

Proof. Let us first assume the query is with only one negation pattern.

Suppose Et is a negation pattern in the event sequence query as: EVENT $SEQ(E1, E2, E3, ..., Ei, !Et, Ej, ..., Em)$ WITHIN W. Assume $<e1\ e2\ e3\ ...\ ei\ ej\ ...\ em>$ is any matching event sequence input to the NG operator. Suppose that at this moment, there is no Et events received with a timestamp within the range [ei.timestamp, ej.timestamp]. The above sequence $<e1\ e2\ ...\ em>$ will thus be put into the NG operator's output. However, any out-of-order event et can possibly be received after em with a timestamp ei.timestamp < et.timestamp < ej.timestamp. Thus, the $<e1\ e2\ ...\ em>$ actually is a spurious event sequence that erroneously had been sent out to upper operators. For the queries with more than one negation pattern, repeat the proof above for each of its negation patterns. □

4.3.4 Problem for Purge at SSC (PSSC)

A basic mechanism for window-based AIS checking is to compare the difference between the checked event and the latest event received by the system. According to the sliding window semantics, any matching event sequence $<e1\ e2\ ...\ em>$ for event pattern $SEQ(E1, E2, ..., Em)$ must satisfy the time-based constraint that (em.timestamp - $e1$.timestamp) < W. For any event instance ei kept in AIS, it can be purged from the stack once an event ek with (ek.timestamp - ei.timestamp) > W is received by the query engine. However, with out-of-order data arrivals, the above window-based AIS purge is no longer "safe".

Example 4.5. In *Figure 4.3(c)*, the out-of-order event b_8 comes after f_{17}. The out-of-order b_8 should be put together with a_3 and d_{10} to form a candidate event sequence output (<a_3 b_8 d_{10}>) during the sequence construction. However by the above AIS purging, a_3 would have already been removed.

Suppose the problems mentioned in *Chapter 4.3.2* and 4.3.3 are solved. Retrieving an out-of-order event might then trigger the construction of a new candidate event sequence, such as <a_3 b_6 d_8> in *Example 4.5.* We refer to such event sequences which consist of some out-of-order event as *out-of-order event sequence.* Out-of-order data arrival triggers the construction of out-of-order event sequences. We can see from the above example that PSSC purges some events from AIS which might be needed for forming such out-of-order event sequences in the future. We refer to this as *unauthorized AIS purge.* It prevents some out-of-order event sequences from being constructed by the SSC operator. For example, <a_3 b_6 d_8> can never be constructed due to the AIS purging on a_3 or b_6. Intuitively we can see that once out-of-order data arrival being possible, any data purge at AIS becomes "unsafe", as expressed by the theorem below.

Theorem 4.2. Any data purge of active instance stack (AIS) is unauthorized unless the total order on the data arrival holds for the input stream.

Proof. For any event instance kept by e_i in AIS, suppose that it is purged at some moment during the evaluation, and let's assume the event received right before e_i is purged is e_k. There can be out-of-order events e_{i-n}, e_{i-n+1}, ..., e_{i-1}, e_i, e_{i+1}, ..., e_{i+m-1}, e_{i+m} received after e_k ($i > n > 0$ and $m > 0$). By the notation we have e_{i-n}.timestamp < e_{i-n+1}.timestamp < ... < e_{i-1}.timetamp < e_i.timestamp < e_{i+1}.timestamp < ... < e_{i+m-1} <

e_{i+m}.timestamp. We can have e_{i+m}.timestamp - e_{i-n}.timestamp $<$ W by defining the incremental time unit from e_{i-n} to e_{i+m} to be small enough. Thus, e_i can be used to form a future potential out-of-order event sequence $<e_{i-n}\ e_{i-n+1}\ ...\ e_i\ ...\ e_{i+m-1}\ e_{i+m}>$. Hence the purge on e_i is an unauthorized AIS purge. $\square$

4.3.5 Problem for Purge at Negation (PNG)

Similarly to the AIS purge, the window constraint-based mechanism for purging from the negation buffer can be conducted simply by comparing the distance between the checked event and the latest event received at the system. As the previous example in *Chapter 4.2*, c_5 can be purged from the negation buffer once f_{16} is received. However, with out-of-order data arrival, such data purge on the negation buffer is no longer "safe", as shown in the following example.

Example 4.6. Assume the input event stream is as the one shown at *Figure 4.2(d)*, where out-of-order event b_4 comes after f_{17}. Suppose that the SSC can handle the out-of-order event arrival correctly, b_4 will lead to the construction of an out-of-order event sequence ($<a_3\ b_4\ d_{10}>$) by the SSC. This out-of-order event sequence will be passed up to the NG operator for further checking by applying the negation semantics. Obviously $<a_3\ b_4\ d_{10}>$ is not a qualified sequence because there is a negation event c_5 between b_6 and d_8. However, c_5 has been removed by the purge at this moment.

As the above example, PNG purges some events from the negation buffer which might be needed in the future for forming out-of-order sequence. We refer this as **unauthorized negation buffer purge.** Unauthorized negation buffer purge causes some unqualified out-of-order event sequences to be filtered out at the NG operator. For example, <a_3 b_6 d_8> will be mistakenly treated as a qualified sequence by the NG due to the c_5 having been removed. Similarly to the AIS purging, we can see that once out-of-order data arrival is possible, any data purge on the negation buffer becomes "unsafe" , as shown by the *Theorem 4.3* below. Its proof is similar to the proof of *Theorem 4.2*.

Theorem 4.3. Any data purge on the negation buffer is unauthorized unless the total order on the data arrival holds for the input stream.

4.3.6 Summary

Above we have discussed the SSC operator and its state purge function causing the *missing sequences* and *producing incorrect sequences*, as shown in *Figure 4.5*, corresponding to the SSC operator and the PSSC function described in *Chapter 4.2*. Given event query EVENT $SEQ(E1, E2, ..., Em)$ WITHIN W, the query plan is shown in *Figure 4.5*. We can see that the problems are all related to the in-memory data structures (AIS) at SSC. Assuming that precise query result is required, evaluation approach in *Chapter 4.2* is no longer sufficient once out-of-order data arrival is possible.

4.4 Levels of Correctness

Several aspects of "output correctness" are defined as below.

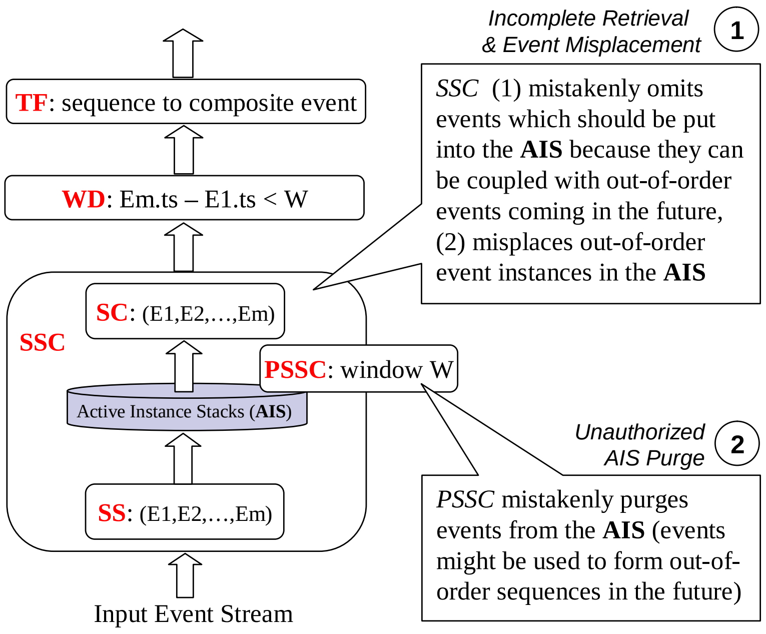

Incomplete Retrieval & Event Misplacement
1
SSC (1) mistakenly omits events which should be put into the AIS because they can be coupled with out-of-order events coming in the future, (2) misplaces out-of-order event instances in the AIS
TF: sequence to composite event
WD: Em.ts – E1.ts < W
SSC
SC: (E1,E2,…,Em)
PSSC: window W
Active Instance Stacks (AIS)
SS: (E1,E2,…,Em)
Input Event Stream
Unauthorized AIS Purge
2
PSSC mistakenly purges events from the AIS (events might be used to form out-of-order sequences in the future)

Figure 4.5: Problem Observation

Ordered. The *ordered* property holds if and only if for any sequence result T = $<e1\ e2\ ...\ em>$ from the system, we can guarantee that every future sequence result $T' = <e1'\ e2'\ ...\ em'>$ with $T' >_o T$. $T' >_o T$ holds **iff** $em.t \leqslant em'.t$ and if $\exists\ k\ (1 \leqslant k \leqslant m-1)\ ek.t > ek'.t$, then $\exists\ p\ (k < p \leqslant m)$, $ep.t < ep'.t$.

In-Time. Our algebra assumes the execution is driven by the arrival of new events. The *in-time* property holds if for any event sequence output T = $<e1\ e2\ ...\ em>$ from the system where we assume *ek* is the last event instance of T received by the engine, the output of T must be initiated by the arrival of *ek*.

Permanently Valid. The property of being *permanently valid* holds if and only if all output result sequences from the system satisfy the query semantics. That is, for any sequence result T = <*e1 e2 ... em*>, it should satisfy the sequence constraint as $e1.t \leqslant e2.t \leqslant e3.t \ldots \leqslant em.t$; the window constraint (if any) as $em.t - e1.t \leqslant$ *window size*; the predicate constraints (if any) and the restriction on the negation filtering. Satisfying the negation filtering constraint is defined as follows. Assume in the query there is a negation pattern *NE* between event type E and E' and E maps to *ep* and E' maps to *eq* in T. The negation filtering constraint is satisfied **iff** *(4-1)* no current received event *ne* of type *NE* such as $ep.t \leqslant ne.t \leqslant eq.t$ and *(4-2)* no future received event ne' of type *NE* such as $ep.t \leqslant ne'.t \leqslant eq.t$

Eventually Valid. We define a property a bit weaker than the "permanently valid" above. The validation towards satisfying the restriction on the negation filtering (defined in *(4-1)* and *(4-2)* above) is loosen up on *(4-2)* to the following: *(4-2)*$'$ if there is any future received *ne* of type *NE* satisfying $ep.t \leqslant ne.t \leqslant eq.t$, the invalidation of the previously output tuple T can be notified and the correction of such invalid output can be achieved. This output mechanism is also seen as being "valid", which is denoted as "eventually valid".

The "permanently valid" and "eventually valid" defined above are two different levels of satisfying "valid" result output.

Complete. If current received event instances <*e1 e2 ... em*> satisfy the query semantics (defined in the "permanently valid" above then the sequence result T = <*e1 e2 ... em*> will at some point of time be output by the system.

As summary, four different properties of output correctness are defined: *(a)* ordered, *(b)* in-time, *(c)* valid and *(d)* complete. The property of being "valid" can be categorized into two levels: *(c-1)* permanently valid and *(c-2)* eventually valid.

Based on such categorization, by combination we can define (2 * 2 * 3 * 2 =) 24 different categories of output correctness. Some of them can never be possible. For example, it is not possible that an execution strategy produces permanently correct unordered results with zero latency. The reason is that with out-of-order event arrivals, sequence results cannot be output immediately safely. Similarly, it is not possible that output tuples produced are eventually correct and at the same time keeping the order. The reason is that sequences sent by some later compensation computation can lead to out-of-order output. Several combinations as different levels of output correctness are now introduced as below.

- Full Correctness: the query evaluation satisfies the property of ordered, in-time, permanently valid and complete output.
- Delayed Correctness: the query evaluation satisfies the property of ordered, permanently valid and complete output.
- Delayed Unsorted Correctness: the query evaluation satisfies the property of permanently valid and complete output.
- Convergent Correctness: the query evaluation satisfies the property of ordered, in-time, eventually valid and complete output.

- Convergent Unsorted Correctness: the query evaluation satisfies the property of in-time, eventually valid and complete output.

Although “full correctness” is a nice output property, it is too strong a requirement and unnecessary in most practical scenarios (in fact, if events come out-of-order, “full correctness” cannot be achieved). In the applications while real-time valid output is required, “delayed correctness” or “delayed unsorted correctness” may be necessary, i.e., the receiver performs business action triggered by individual sequence results requires each result sent from the stream provider to be guaranteed valid. On the other hand, the application where real-time correctness is not important but there is a high requirement on system response time, “convergence unsorted correctness” or “convergent unsorted correctness” may be a more desired property, i.e., some online statistic analyzing tools requiring a large input rate for undertaking coarse granularity mining would prefer a fast data feed-in instead of a guaranteed valid one from the stream provider.

In *Chapter 4.5* I introduce a **slack-based approach** which satisfies the *delayed correctness.* A **conservative query evaluation approach** which satisfies the *delayed unsorted correctness* and an **aggressive query evaluation approach** which satisfies the *convergent unsorted correctness* are studied in [LLG$^+$09], which are not included in this dissertation.

4.5 Solution

4.5.1 Assumption on Unordered SSC Output

Construction of the out-of-order event sequence actually is delayed by its out-of-order event components. Suppose a_0 and d_2 in *Example 4.2* both were to arrive in order. Then the sequence $<a_0\ b_1\ d_2>$ would have been constructed before $<a_3\ b_6\ d_{10}>$. Assuming the execution of SSC produces output event sequences whenever new sequences are being formed, with out-of-order data arrival, the output order of the SSC can no longer be guaranteed.

If ordered output is needed from the SSC operator, additional semantic information such as K-Slack factor or punctuation is needed to "unblock" the on-hold candidate sequences from being output by the SSC operator. Since the input event stream to the query engine is unordered, it is reasonable to produce unordered output events to downstream. Thus in this dissertation, unordered sequence output at the SSC operator is permitted.

4.5.2 Solution for SSC

SSC operator consists of three major procedures: *(1)* event retrieval; *(2)* AIS construction and *(3)* event sequence production, with the first two affected by out-of-order data arrival as the previous discussion in *Chapter 4.2*. The following is the proposed mechanism for event retrieval and AIS construction.

Event Retrieval Mechanism. To avoid incomplete retrieval, all states of the NFA need to be set active before the retrieval over the event stream. Let's look at *Example 4.2*. With all the automaton states activated at the beginning, b_1 can be retrieved by the automaton even though no A events have appeared before it.

AIS Construction Mechanism. For avoiding event misplacement, we have to insert the retrieved events into the right position of AIS. In the case of total order, any new received event can be simply appended to the end of the sequence. We refer to this as the "*append semantics*". When events can arrive out of order, the "*sort semantics*" need to be applied: for each event instance that triggers a transition in NFA, instead of simply appending it to the stack, we search for a proper insertion place in the corresponding stack to guarantee that the event instances in the same stack are in chronological order from bottom to top. Also, the context RIP pointer of the inserted event e needs to be correctly set. Besides that, if e is not the rightmost event type in the sequence pattern of the query, RIP of the event instances in the right-adjacent stack might need to be updated as well. If the timestamp of e is in between of an event e' in the right-adjacent stack and the event pointed by the RIP fields of e' and e' will need to be reset to e.

Example 4.7. Similar to *Example 4.3*, let's again consider the event stream in *Figure 4.3(c)* with out-of-order event b_8 arriving after event f_{17}. Once b_8 is received, it is inserted between b_6 and b_{11} in stack S2. Event b_8 is not of a final state event type in the sequence pattern of the query. Thus we need to check the D event instances in stack S3 to see whether any of their RIP field needs to be reset. Since b_8 becomes the most recent event in stack S2 whose timestamp is smaller than the timestamp of d_{10}, the RIP field of d_{10} should be reset from the original b_6 to b_8.

Once a new event e is retrieved, it might trigger the construction of event sequences in SSC. By the total order assumption, only events from the rightmost event pattern in the query (D event type in *Example 4.1*) trigger the event sequence construction in SSC. However, with out-of-order data arrival, any located event might trigger sequence construction in SSC. If the event retrieval and AIS construction are correctly handled as above, the SSC operator needs to produce out-of-order event sequences whenever some new opportunity arises. For instance, two out-of-order event sequences - $<a_3\ b_8\ d_{10}>$ and $<a_3\ b_8\ d_{15}>$ - should be constructed by SSC after b_8 is inserted into the stack S2 in *Example 4.7.* The proposed process for the SSC operator which handles out-of-order data arrival is shown by the below *Algorithm 6* .

Algorithm 6 Out-of-Order Handling Incorporated SSC

```
1: Procedure: OutOfOrderSSC
2: Input:
3: (1) event Query EVENT SEQ(E1, E2, ..., Em) WITHIN W,
4: (2) AIS constructed from previously input events,
5: (3) newly received event $e$ (under event type $E$)
6: Output:
7: (1) updated AIS,
8: (2) matched result sequences triggered by the input event instance
9:
10: if event type $Ei$ is among $E1$, $E2$, ..., $Em$ then
11:     insert $e$ into stack $S_i$ (using "sort semantics")
12:     set $e$'s RIP
13:     check RIPs of the instances in $S_{i+1}$ and reset the ones being affected by $e$
14:     produce event sequences containing $e$ if any
15: end if
```

Optimization 4.1. *Line 11* and *12* in *Algorithm 6* add a newly located event into AIS by applying the “sort semantics” and then sets its RIP field.

Line 13 checks the RIP field of the event instance in the right-adjacent stack and resets the ones being affected by the newly located event. However, if the received event is "in-time", we will continue to follow the previous "append semantics": that is we simply put the event at the end of the corresponding stack and set its RIP as the most recent event in the left-adjacent stack. *Line 13* is no longer necessary for such in-time events. Besides that, sequence construction at *Line 14* of *Algorithm 6* would only be triggered when the received event type is at the rightmost in the query sequence (D events).

To avoid such overhead caused by treating every event as a "potential" out-of-order event, the SSC operator can maintain an "AIS-CLOCK" value, which equals to the largest timestamp of the events at AIS. *Algorithm 7* shows the optimized approach. Once a newly retrieved event is with a timestamp larger than the current AIS-CLOCK, AIS-CLOCK will be updated to this value. Such an event can be handled simply by the "append semantics" and corresponding steps for in-order events (*Line 7* to *11* in *Algorithm 7*). Whenever a newly retrieved event is with a timestamp smaller than the AIS-CLOCK, we instead apply "sort semantics" and conduct the corresponding out-of-order-specific steps (*Line 14* to *16* in *Algorithm 7*).

Example 4.8. Consider the sequence construction of out-of-order event instance a_1 under sequence query *SEQ*(A, B, D) over the runtime AIS state shown in *Figure 4.6*. Event instance a_1 needs to find matching event entry in S2 and S3 to produce out-of-order event sequences. The timestamp of b_3 is greater than a_1 and the timestamp of d_4 is greater than b_3. All the

Algorithm 7 Optimized SSC with AIS-CLOCK

```
1: Procedure: OptimizedOutOfOrderSSC_1
2: Input / Output:
3: same as Algorithm 6
4:
5: if e's event type $Ei$ is among $E1$, $E2$, ..., $Em$ then
6:     if e.timestamp < AIS-CLOCK then
7:         buffer e
8:         insert e into stack $S_i$ (using "sort semantics")
9:         set e's RIP
10:        check the RIP field of the instances in stack $S_{i+1}$
11:        & reset the ones being affected
12:        produce event sequences containing e if any
13:    else
14:        buffer e
15:        insert e into stack $S_i$ (using "append semantics")
16:        set e's RIP
17:        if $Ei = Em$ then
18:            produce event sequences containing e if any
19:        end if
20:    end if
21: end if
```

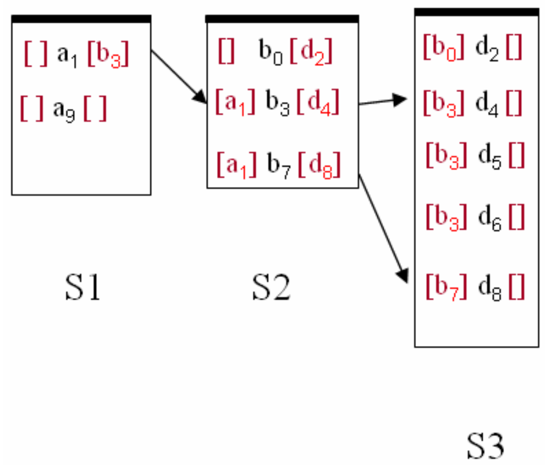


Figure 4.6: SSC using Active Instance Stacks with RIN

event instances after d_4 in S3 can be matched with b3 if event sequences involving b3 need to be constructed. Similarly, $b7$ needs to find matching event instance in S3 which timestamp is greater than 7. And the matching event entry of b_7 in S3 is d_8. As we can see, event instances need to find event entries in out-of-order event construction. And it is time-consuming for the system to find these entries when the query pattern is long and there are many event instances between adjacent event entries in the same stack. The following is an optimization technique for decreasing such cost.

Optimization 4.2. I propose the most recent instance in the stack of the next state (RIN). For an event instance e, instead of only one field as the RIP, such RIN field is added. Take the active instance a1 in the S1 stack in *Figure 4.6*. The most recent instance in the stack of the next state of a_1 is b_3, so the RIN field of a_1 is set to b_3. This RIN field tells that any instances in the S2 stack up to b_3 can be matched with a_1 if event sequences involving a_1 need to be created. When a new event e comes, the RIN filed of the event needs to be correctly set. Besides that, if e is not the leftmost event type in the sequence query, RIN of the event instances in the previous stack might need to be updated as well. If the timestamp of e is less than the original RIN value of an event in the previous stack but greater than the timestamp of the event in the previous stack, we then update the RIN field of the event to the timestamp of e. By using the RIP and RIN together in *Algorithm 8*, we can perform the out-of-order sequence construction in *Algorithm 7* more efficiently.

4.5.3 Solution for PSSC

When out-of-order data arrival is possible, based on *Theorem 4.2*, no event instance in AIS can be purged safely by the PSSC. To avoid errors, no data

Algorithm 8 Optimized SSC with AIS-CLOCK and RIN

```
Procedure: OptimizedOutOfOrderSSC_2
Input / Output:
same as Algorithm 6

if e's event type Ei is among E1, E2, ..., Em then
    if e.timestamp < AIS-CLOCK then
        buffer e
        insert e into stack $S_i$ (using "sort semantics")
        set e's RIP
        set e's RIN
        check the RIP values of the instances in stack $S_{i+1}$
        & reset the ones being affected
        check the RIN values of the instances in stack $S_{i-1}$
        & reset the ones being affected
        produce event sequences containing e if there any
    else
        buffer e
        insert e into stack $S_i$ (using "append semantics")
        set e's RIP
        set e's RIN to be null
        check the RIP values of the instances in stack $S_{i+1}$
        & reset the ones being affected
        check the RIN values of the instances in stack $S_{i-1}$
        & reset the ones being affected
        if $Ei = Em$ then
            produce event sequences containing e if any
        end if
    end if
end if
```

purge can ever be applied on AIS. That is not a realistic solution due to its unbounded memory requirement.

Thus, for "unblocking" the PSSC, we need additional semantic knowledge on the stream source to enable the safe data purge on AIS. K-Slack is a well-known approach [BMM+04][HBR+05][DGP+07] for processing unordered data streams. In real applications, the K-Slack assumption holds in many situations when predictions about network delay can be considered. Besides that, it is very suitable for producing approximate answers if that is acceptable. Thus, I propose a solution for data purging at SSC using the K-Slack semantics.

Here K-Slack is based on time units. It means that the out-of-ordering in event arrivals is within a range of K time units. That is, an event can be delayed for at most K time units. For example, in *Figure 4.3(a)*, the out-of-order event c_9 is received after f_{16}. Thus it is delayed for 7 ($16 - 9 = 7$) time units. If we set the K value as 5, the out-of-order data arrival case in *Figure 4.3(a)* would never arise.

Window purge using K-Slack compares the distance between the checked event and the latest event received at the system. A CLOCK value which equals to the largest timestamp seen so far for the received events is maintained. Each time the CLOCK value is updated, PSSC will be notified. According to the sliding window semantics, for any event instance e kept in AIS, it can be purged from the stack if (e.timestamp + W) < CLOCK. Thus, under the out-of-order assumption, the above condition on window purge will be (e.timestamp + W + K) < CLOCK. This is because after waiting for K time units, no out-of-order event with timestamp less than (e + W) can arrive. Thus e can no longer contribute to forming a new candidate sequence.

SSC passes the updated CLOCK values up to the PSSC whenever a new event with a larger timestamp is seen. Thus, before *Line 5* in *Algorithm 7*, we trigger PSSC by adding the following:

```
IF ei.timestamp > CLOCK
   CLOCK  = ei.timestamp and pass a CLOCK triggering to PSSC;
```

The below *Algorithm 9* depicts the basic approach for AIS purging incorporated into the out-of-order event handling by applying the K-Slack constraint. Each time the CLOCK is updated, PSSC gets triggered. Event instances in AIS will be purged when the previously introduced purge condition is satisfied.

Algorithm 9 Out-of-Order Handling Incorporated SSC State Purge

```
1: Procedure: OutOfOrderSSCPurge
2: Input:
3: (1) current AIS,
4: (2) CLOCK triggering from SSC
5: Output:
6: updated AIS
7:
8: On receiving a CLOCK triggering for event instance $e$ in AIS
9: if $e$.timestamp + W + K < CLOCK then
10:     purge $e$
11: end if
```

Example 4.9. Let's consider purge when evaluating event query *SEQ*(A, B, D) on the data in *Figure 4.2(c)*. Event instance a_3, b_6 and d_{10} are kept in AIS after d_{10} is received. Event d_{10}'s RIP points to event instance b_6 and b_6's RIP points to a_3. Suppose event f_{21} (which is not shown in the figure) is received after f_{16} and the window size W equals to 7. Assume K value equals to 2 for the K-Slack constraint. As more data is received, the CLOCK value increases and the order of those three event instances being purged from AIS is a_3 (due to $13 > 3 + 7 + 2$, when c_{13} is met), b_6 (when f_{16} is met) and then d_{10} (when f_{21} is met).

Holding the outdated event sequences in the AIS structure increases the workload of the SSC operator for event sequence construction. Take *Example 4.9* for instance. For data arrival under the total order assumption, when b_{15} is received, both a_3 and b_6 can be purged from AIS (due to $3 + 7 < 15$ and $6 + 7 < 15$). So, there are only three instances in AIS at this moment: a_7 in stack S1, b_{11} in S2 and d_{15} in S3. Thus, by receiving d_{15}, SSC operator produces one new event sequence output (<a_7 b_{11} d_{15}>). In the out-of-order scenario, SSC might produce more sequence output than in the in-order case. In *Example 4.9*, assume the K value of K-Slack constraint is 10. When d_{15} is met, event instances a_3, b_6 and d_{10} are still kept in AIS. Thus, by receiving d_{15}, the SSC operator produces three event sequences: <a_3 b_6 d_{15}>, <a_3 b_{11} d_{15}> and <a_6 b_{11} d_{15}>. The first two sequences actually should not be produced. This is because a_3, b_6 are both "outdated" event instances. They are held in AIS just for out-of-order event sequence construction once possible out-of-order events coming in the future. Thus, coupling the in-order event d_{15} with the outdated events a_3 and b_6 is not necessary. An event sequence produced by such construction can never be a result sequence because they would be removed later by window-based filtering (functionality of the WD operator). Thus, it also brings burden to the window-based filtering computation. Many of the outdated event instances may be kept in the AIS stacks if the K value is large. Thus the above overhead on sequence construction and AIS filtering should be considered. Below I propose a technique for decreasing such cost.

Optimization 4.3. We divide each stack in AIS into two parts: outdated event instances and up-to-date event instances. A divider is set for each stack: instances on or above it are outdated instances and instances below it are up-to-date ones. For a stack without outdated events, the set of outdated instances is empty. Besides applying the slack-based purge in *Algorithm 9*, the basic data purge introduced in *Chapter 4.2* is also applied. The divider for each stack will be set using such basic purge. While an in-order event triggers sequence construction in SSC (*Line 17* and *18* in *Algorithm 7*), only the events below the divider in each stack will be considered.

Again let's look at *Example 4.9* with a K value equal to 10 and window W equal to 7. When d_{15} is met, the divider of stack S1 is set to a_3 and the divider of stack S2 is set to b_5. Thus, only one new sequence (<a_7 b_{11} d_{15}>) will be constructed when the in-order event d_{15} is received. Construction of event sequences <a_3 b_{11} d_{15}> and <a_6 b_{11} d_{15}> is avoided by applying the AIS partition.

Optimization 4.4. For two event instances *ei* and *ej* in AIS *ej*'s RIP pointing to *ei.* Observe that if the condition of purging *ej* is satisfied ((*ej*.timestamp + W + K) < CLOCK), conditions to purge *ei* must also be satisfied. This gives an opportunity for lazy AIS purge: for each CLOCK update, only the instance in the last AIS stack will be checked for data purge. For any instance is purged from there, we can purge instances in other AIS stacks following the RIP path. Let's again look at *Example 4.9.* Performing the lazy PSSC purge, each time the CLOCK update triggers the PSSC, only the event instances in stack S3 (holding the D events) will be checked for possible purge. Purge of any D event instance from S3 will trigger the purge on stack S2 and then stack S1, following the RIP linking of the purged instances. In *Example 4.9*, d_{10} is purged when f_{21} is met. Event b_6 and a_3 are purged right after that because "d_{10} –> b_6 –> a_3" forms a chain through RIP linking.

4.5.4 Solution for NG and PNG

When out-of-order data arrival is possible, based on *Theorem 4.1*, we can see that a NG operator cannot produce any non-spurious data output. Thus, the NG is forever blocked. Using negative tuples is a possible approach to unblock the NG operator. We can let the NG operator output candidate event sequences even though they may be spurious answers. Whenever an out-of-data negation event arrives, we produce and send up negative tuples to correct the previous results. However, this approach has many drawbacks: *(1)* it requires the upper operators to have the ability to handle negative tuples; *(2)* it does not quite fit into the real-time applications of event stream processing; *(3)* unbound data holding is required for tracing back to the sequences that have already been output by the NG for producing the negative tuples. Thus in this dissertation we will not consider such approach and detail pros and cons of that approach is given in [LLG$^+$09].

Similar to the K-Slack in PSSC, for "unblocking" the negation operator and enable data purge at the operator state, slack-based approach could be applied for NG and PNG.

A conservative mechanism for NG using K-Slack is simply "postponing" the existing negation filtering K time units to capture all possible negation events, where a CLOCK value equal to the current largest timestamp from the received events is maintained.

Example 4.10. Let's first look back at *Example 4.4*. When d_{10} is seen, SSC produces $<a_3\ b_6\ d_{10}>$ as output to the NG operator. At this moment, the negation buffer holds only one event instance (c_5), It is with a timestamp not in the range of [6,10]. Thus, <a_3 b_6 d_{10}> should be output in the total order scenario. However, to avoid producing spurious sequences in the out-of-order scenario, NG cannot output this tuple in case potential out-of-order event such as c_9 in *Figure 4.3(a)* coming later. Suppose the input data is with the K-Slack constraints and K equals to 6. In such case, when event f_{16} is met, the CLOCK value is updated to 16. Potential out-of-order C events will be at least with a timestamp larger than 10 (10 = 16 - 6). So future out-of-order C events, if any, will not have timestamp within the range of [6,10]. Out-of-order event such as c_9 cannot be possible under K-Slack constraint after seeing event f_{16}. The NG operator can thus output the candidate event sequence <a_3 b_6 d_{10}> delayed for K time units. For simplification, we place only one negation component inside the query. Negation algorithm for queries with more than one negation patterns is similar. Every negation pattern needs to be taken care of in the negation filtering. The event sequence query is given as: EVENT *SEQ*(E1, E2, E3, E4, ..., Ei, !*NE*, Ej, ..., Em) WITHIN W, where *NE* is a negation pattern. While the SSC processing any new event during the retrieval phase from the input event stream, it will be put into the negation buffer if it is under the negation event type *NE*. Thus in *Algorithm 7*, we add the following lines after *Line 18*:

```
ELSE IF ei is NE type
      pass up ei to NG
```

Besides that, SSC passes up updated CLOCK value, whenever a new event with a larger timestamp is seen, to NG. Thus, before *Line 5* in *Algorithm 7*, add the following lines:

```
IF ei.timestamp > CLOCK
     CLOCK  = ei.timestamp
      pass up a CLOCK triggering to NG
```

If an event is the retrieved and triggers producing any new event sequence output from SSC, it will be passed up to the upper operators (WD or NG). We assume the computation of WD filtering has been pushed down to the SSC. Then the SSC's sequence output will feed to the NG directly. *Line 12* and *18* in *Algorithm 7* both change to the following:

```
try to produce event sequences containing ei
IF event sequence(s) being produced
       pass up the produced sequences to NG
```

The NG operator receives the above negation pattern event, updated CLOCK value and event sequence output from SSC. The algorithm for out-of-order incorporated negation operator is given below in *Algorithm 10.*

A set named "holding set" is applied to keep the spurious event sequences which cannot be output safely by the NG operators yet. Let's take *Example 4.10* to further illuminate *Algorithm 10.* When d_{10} is received at SSC, sequence $<a_3\ b_6\ d_{10}>$ is produced by SSC and passed to NG. It is put into the holding set. When f_{16} is reached, the CLOCK is updated to 16 and the NG is notified. The NG operator goes to check the sequences kept in the holding set (only $<a_3\ b_6\ d_{10}>$ there). It can be safely output from NG and then removed from the holding set at this moment (*Line 9* to *11*). For a sequence kept in the holding set, it will be removed from the set if during the holding any out-of-order negation event which timestamp is within the

Algorithm 10 Out-of-Order Incorporated Negation

```
1: Procedure: OutOfOrderNegation
2: Input:
3: (1) current negation buffer, (2) current holding set,
4: (3) negation pattern event / CLOCK triggering / event sequence
5: Output:
6: (1) updated negation buffer, (2) sequence output of negation,
7: (3) updated candidate set on receiving a negation event instance e
8:
9: On receiving newly received negation event e (under event type Ei) among the event
   sequence:
10: add e into the negation buffer
11: prune the holding set using e
12: On receiving an CLOCK triggering:
13: check each sequence in the holding set:
14: if any hold candidate hc: <e1 e2 e3 ... em> satisfies em.timestamp + K < CLOCK
    then
15:     output and then remove hc from the holding set
16: end if
17: On receiving sequence <e1′ e2′ ... em′>:
18: check the negation buffer
19: if no negation events e's timestamp is within the range of [e1′.timestamp, em′.timestamp]
    then
20:     put <e1′ e2′ ... em′> into the holding set
21: end if
```

range of [*ei*.timestamp, *ej*.timestamp] being added into the negation buffer. Suppose out-of-order event c_9 is received between d_{15} and f_{16}. Then the a_3 b_6 d_{10} will be filter out from the holding set by c_9.

It is a little different to handle sequence query such as *SEQ*($E1$, $E2$, $E3$, ..., Em, !*NE*), where *NE* is a negation pattern. For such query, output sequence <$e1$ $e2$ $e3$ ... em> from SSC will be put into the holding set of NG if no *NE* events in the negation buffer with a time stamp within the range of [*em*.timestamp, $e1$.timestamp + W] (W is the window size). When the CLOCK value satisfies CLOCK $e1$.timestamp > *em*.timestamp + K, this sequence can be safely output by the NG operator.

Optimization 4.5. The conservative mechanism on output sequences from the holding set “delays” output for K time units. This is already the earliest moment for the sequence output for queries such *SEQ*(A, B, !C, D). Now let's consider query *SEQ*(A, B, !C, D, E). For an event sequence kept in the holding set such as <a b d e>, we do not need to wait till the e.timestamp + K < CLOCK to safely output sequence <a b d e>. The sequence could be output earlier at the moment when d.timestamp + K < CLOCK. Generally, take a sequence query *SEQ*($E1$, $E2$, ..., Ei, *NE*, Ej, ..., Em) where *NE* is the last negation pattern in the query sequence. Instead of following the conservative mechanism to "delay K time units", an event sequence <$e1$ $e2$ ... ei ej ... em> kept in the negation buffer can be output from the holding set once ej.timestamp + K < CLOCK. For the long sequence with the negation pattern(s) appearing relatively early, this can shorten the holding on the event sequence output for the NG operator.

The basic approach and optimization technique of using K-Slack to handle the PNG is similar to the approach for PSSC. Thus the corresponding description is skipped in this dissertation.

4.6 Performance Evaluation

4.6.1 System Implementation

Figure 4.7 shows the system architecture for incorporating the proposed out-of-order handling into the basic ESP system structure given in *Chapter 2.3*. Out-of-order event-incorporated SEQ and Negation operator are added into the corresponding operator library containers.

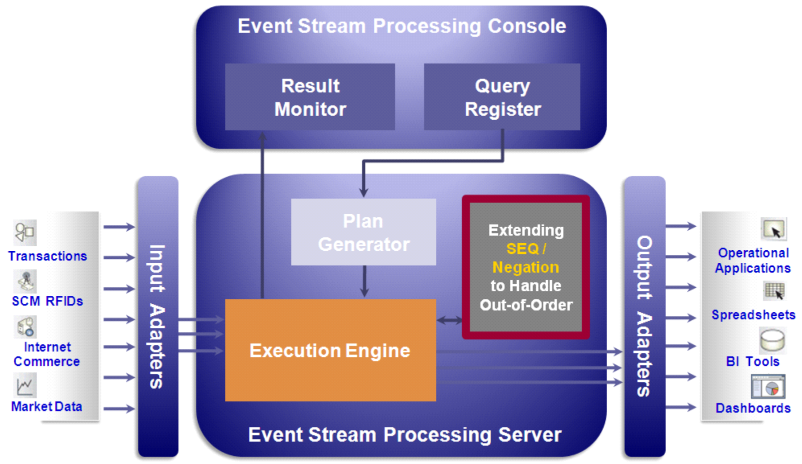


Figure 4.7: Out-of-Order Event-Incorporated ESP System Architecture

4.6.2 Experimental Setting

The proposed techniques have been implemented in a prototype system using Java 1.4. An event sequence generator is implemented for simulating sequences under different properties. Experiments are run on two Pentium4 3.0Ghz machines each with 512M RAM.

The percentage of the out-of-order events and the K-Slack factor are set in the generator. In the experiments, one machine generates and sends the event stream to the second machine, i.e., the query engine. In this dissertation I only provide the experimental results on the out-of-order incorporated SSC and PSSC operator. The results on negation-related operators can be found in [LLG$^+$09].

4.6.3 Sequence Scan and Construction

Figure 4.8 shows the CPU gain when applying the AIS-CLOCK technique. A sequence query of length 6 (i.e., $SEQ(A, B, C, D, E, F)$) is run on five different data sets, with the size ranging from 20000 to 100000. The percentage of out-of-order events is 90% in all datasets. Y axis shows the accumulated cost on runtime AIS construction (inserting events and resetting RIP) during the query evaluation. We observe that applying AIS-CLOCK can reduce the overall cost of AIS construction even though the percentage of in-order data is very small. For a decreased percentage of out-of-order data, the performance gain in CPU cost increases. Take the dataset with 80000 events as example. The gain of applying AIS-CLOCK jumps from 8% to 43% if the out-of-order percentage is decreased from 90% to 30%.

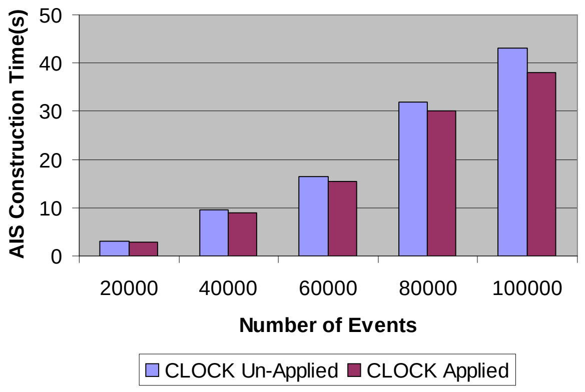


Figure 4.8: Results for Applying AIS-CLOCK

4.6.4 Purge at SSC

We now study the performance of applying AIS partition during the SSC purge. A sequence query of length 6 is run and the window size is set as 20.

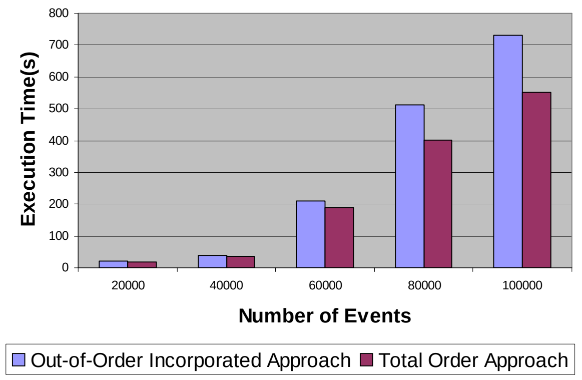


Figure 4.9: Technique Overhead

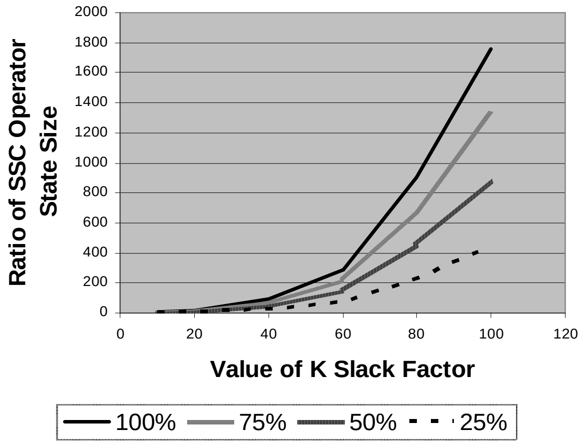


Figure 4.10: Results for Applying AIS Partition I

Performance gain on memory is shown in *Figure 4.10* and in CPU cost is shown in *Figure 4.11*. Through partitioning AIS, construction of outdated event sequences will be avoided for the in-order portion of the input stream. We observe that either a larger percentage of "in-order" events or a larger value slack factor result in more memory and CPU gain by applying AIS partition. Studying quantitatively, the ratio of intermediate buffer size of SSC is directly proportional to the value of $((K + W) / W)^S$, where K is the slack factor, W is the window size and S is the query sequence length.

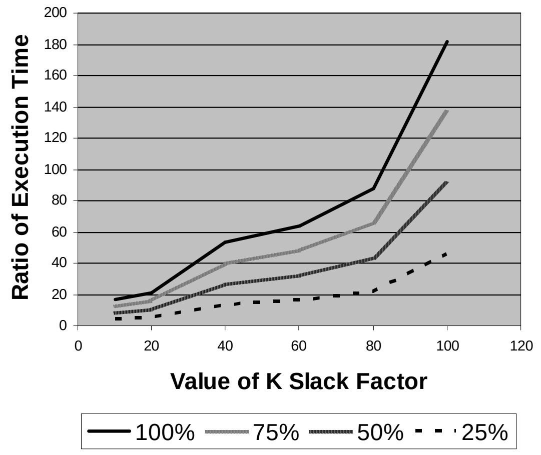


Figure 4.11: Results for Applying AIS Partition II

4.6.5 Overhead of Out-of-Order Handling

We now test the overhead of the proposed out-of-order event stream processing techniques. We utilize the same setting in *Chapter 4.6.3* but the out-of-order data percentage is set as 0%. In other words, all the input events are "in-order". Thus, evaluation based on the total order assumption can be applied in this scenario. The simple approach based on total order assumption and the out-of-order incorporated approach are compared in *Figure 4.9*. The performance difference (execution time denoted on the Y axis) is then the overhead of applying the out-of-order handling. Proposed techniques of AIS-CLOCK and AIS partition are both applied in the out-of-order incorporated approach. The overhead ranges from 5.1% to 24.6% in the five given datasets. The overhead increases while increasing the dataset size due to the cost on extra timestamp checking and AIS maintenance.

4.7 Related Work

Most stream query processing research over the past few years has assumed complete ordering of input data [BMM$^+$04][DMRH04][LMT$^+$05]. They tend to work with homogeneous streams, meaning, each stream contains only tuples of the same type. Thus the semantics of general stream processing which employs SQL-like queries composed of join, select, project, aggregation, is not that sensitive to the ordering of the data. Ordering is core for the event pattern detection we are targeting here.

However, there has been some initial work of investigating the out-of-order problem for generic (homogenous-input) stream systems. One model, which is adopted for this work, introduces the notion of K-Slack [BMM$^+$04]. Such solution is trivial in regular stream system as in fact the processing such as join proceeds as normal (with a K-delayed purging), and any tuple that arrives after K is simply discarded [HBR$^+$05]. A native approach [DGP$^+$07] on handling out-of-order event stream is using K-Slack as a priori bound on the input streams. It buffers incoming events in the input queue until ordering can be guaranteed. Compared with the proposed approach where each operator is order sensitive, such processing requires additional space and introduces more latency before allowing events to be evaluated.

A second solution proposed to handle out-of-order data arrival is applying punctuations, namely, assertions inserted directly in the data stream confirming that for instance a certain value or time stamp will no longer appear in the future input streams [DMRH04][LMT$^+$05]. Such techniques, while interesting, require for some service to first be creating and appropriately inserting such assertions - hence we do not consider it in this dissertation. Further research on this topic could be found in [LLG$^+$09][WLL$^+$09].

Lastly, we base the proposed solution on the SASE [WDR06] architecture which has been designed specifically for processing pattern queries over event streams. SASE proposes query language and algebra to support queries on sequential streams, which is adopted as the foundation of this dissertation task. However, [WDR06] does not support out-of-order data arrival.

4.8 Conclusions

In this dissertation task, I address the problem of processing event stream with out-of-order data arrival: *(1)* I analyze the problems state-of-the-art event stream processing technology would experience when faced with out-of-order data arrival and study the levels of correctness in out-of-order processing that target priorities of applications considering latency, output order, result correctness and result completeness; *(2)* I propose new implementation and optimization strategies for the core stream algebra operators such as sequence scan and construction as well as the associated state purge; *(3)* I conduct an experimental study that clearly demonstrates the effectiveness of the proposed approach over existing solutions.

Chapter 5

Complex Event Pattern Detection over Streams with Interval-Based Temporal Semantics

5.1 Introduction

Existing ESP engines have focused on detecting temporal patterns from instantaneous events, that is, events with no duration. Under such a model, an event instance can only be happening "*before*", "*after*" or "*at the same time as*" another event instance. However, such sequential patterns are inadequate to express the complex temporal relationships in domains such as medical, multimedia, meteorology and finance where the events' durations could play an important role.

In such real world applications domains, events have durations, two events can have overlapping portion thus the relations among two event instances is no longer sequential ("*before*", "*after*" or "*at the same time as*") like the point events.

Example 5.1. An ESP engine can be utilized to monitor the events generated by the warehouse of a supermarket. Based on the temperature values sent by the temperature readers, temperature fluctuations as the interval events are generated and sent to the ESP system. We assume three different interval events: *HIGH*, *MEDIUM* and *LOW*. Besides that, the duration of an item staying in the warehouse is extracted and sent to the ESP system as an interval event after the item leaves the warehouse, which is denoted as *STAY*. The event pattern of a *HIGH* event contains a *STAY* event means that the duration of an item staying in the warehouse is with a high temperature the whole time. We can use such pattern to indicate that the quality of this item might not be good.

As discussed in *Chapter 2.1*, event instances happen instantaneously at a time point are called events with point-based temporal semantics (*point events* in short) and event instances that occur over a time interval are called events with interval-based temporal semantics (*interval events* in short). For any event instance e, we use $e.ts$ and $e.te$ to denote the start and the end timestamp of the event instance e, which are called the *endpoints* of the event. The start and the end timestamps of an event instance with point-based semantics are the same, which is simplified as $e.t$ (i.e., $e.ts = e.te =$

$e.t$). Temporal patterns under the point-based temporal semantics and the interval-based temporal semantics have the following major difference:

Additional Temporal Relations between Events. For events with a point-based temporal semantics, the temporal relations between any two events e and e' can only be e "*before*" e' (e.t $< e'$.t), e "equal" e' (e.t $= e'$.t) and e "*after*" e' (e.t $> e'$.t). There are more temporal relations that can be defined between two interval events. According to the classification scheme proposed by [All83], there are 13 temporal relations between any two interval events: "*before*", "*after*", "*during*", "*contain*", "*meet*", "*met by*", "*overlap*", "*overlapped by*", "*start*", "*started by*", "*finish*", "*finished by*" and "*equal*". *Table 5.1* shows the detail of these temporal relations. Some of them are mirror relations. other. For example, a overlaps b can be represented as the relation of b is overlapped by a.

The relations between interval events can be expressed in terms of their endpoints (the start and end of an interval event). Under the classification in [All83], the relation between two interval events can be one from the above 13 relations if the order of all the endpoints of these two events are fixed. Furthermore, while the endpoint order are not fully fixed, relation between event instances can be much more flexible. Because of such difference between the point-based semantics and the interval-based semantics, the query language and evaluation mechanisms used for detecting temporal patterns over point events is not sufficient for pattern detection over interval events. An expressive language to represent the required temporal patterns among streaming interval events is needed. Also, a query evaluation mechanism for

Relation	**Temporal Algebra**
e before e'	$(e.\text{te} < e'.\text{ts})$
e after e'	$(e.\text{ts} > e'.\text{te})$
e during e'	$(e.\text{ts} > e'.\text{ts}) \wedge (e.\text{te} < e'.\text{te})$
e contain e'	$(e.\text{ts} < e'.\text{ts}) \wedge (e.\text{te} > e'.\text{te})$
e meet e'	$(e.\text{te} = e'.\text{ts})$
e met by e'	$(e.\text{ts} = e'.\text{te})$
e overlap e'	$(e.\text{ts} < e'.\text{ts}) \wedge (e.\text{te} > e'.\text{ts}) \wedge (e.\text{te} < e'.\text{te})$
e overlapped by e'	$(e.\text{ts} > e'.\text{ts}) \wedge (e.\text{ts} < e'.\text{te}) \wedge (e.\text{te} > e'.\text{te})$
e start e'	$(e.\text{ts} = e'.\text{ts}) \wedge (e.\text{te} < e'.\text{te})$
e started by e'	$(e.\text{ts} = e'.\text{ts}) \wedge (e.\text{te} > e'.\text{te})$
e finish e'	$(e.\text{ts} > e'.\text{ts}) \wedge (e.\text{te} = e'.\text{te})$
e finished by e'	$(e.\text{ts} < e'.\text{ts}) \wedge (e.\text{te} = e'.\text{te})$
e equal e'	$(e.\text{ts} = e'.\text{ts}) \wedge (e.\text{te} = e'.\text{te})$

Table 5.1: Temporal Relations between Two Intervals

such sequence queries needs to be designed.

Previous research on pattern detection over event streams mainly focused on extracting temporal patterns from point-based event data. For example, [WDR06] proposes *sequence scan and construction* for implementing the SEQ operator introduced in *Chapter 2.2.1*. However it handles the "*before*" / "*after*" temporal relation only on point-based events. Even though in [ACT08][DCR$^+$08][DGP$^+$07] the events are defined based on the interval model, only the "*before*" / "*after*" is supported. The data mining community studied discovering patterns over interval events [KF00] [PHL08][WC07]. [KF00] uses a hierarchical representation that extends Allen's interval algebra [All83] for modeling complex event patterns over intervals. However, this representation is lossy as the exact relationships among the events cannot be fully recovered. [WC07][PHL08] devises a lossless representation to overcome the drawbacks of [KF00]. Based on their proposed representation, they design corresponding mining algorithms for pattern discovery over event intervals. [PHL08] also examines how the discovered temporal patterns can be utilized in classification to differentiate closely related classes thus building an interval-based classifier. However, these works mainly focus on pattern discovering algorithms instead of pattern detection algorithms. Besides that, they do not consider streaming input with window constraints.

Contributions. In this dissertation task, I study query processing over event streams with interval-based temporal semantics. The contributions include:

- I provide a case study of using interval events to optimize network event stream correlation. (*Chapter 5.2*)
- I introduce an expressive language to represent the required temporal patterns among streaming interval events. I design the corresponding temporal operator ISEQ and provide an efficient evaluation strategy for the proposed ISEQ operator. (*Chapter 5.3*)
- For further improving the event processing performance, I provide a mechanism to embed the “interval begin punctuation”(indicating the start of an upstream interval) into the interval stream. Corresponding punctuation-aware query evaluation strategy is studied, which can greatly reduce the runtime memory and CPU footprint. (*Chapter 5.4*)

- I introduce a method to push down the computation of interval event abstraction to the low level sensor network for increasing the computing leverage from the physical level devices. (*Chapter 5.4*)

- I conduct experimental studies to demonstrate the efficiency of the proposed techniques in interval event stream handling. (*Chapter 5.5*)

Roadmap. The rest of *Chapter 5* is organized as follows. *Chapter 5.2* studies the use case of interval events in optimizing network event stream correlation. *Chapter 5.3* proposes an evaluation mechanism for detecting temporal pattern over interval event stream. Ideas of using punctuation and computation pushdown for optimizing the proposed interval stream processing framework are discussed in *Chapter 5.4*. Experimental results are presented in *Chapter 5.5*, followed by related works in *Chapter 5.6*. Conclusions for this dissertation task is given in *Chapter 5.7*.

5.2 Case Study: Intervals in Stream Correlation

5.2.1 Event Stream Correlation

Event Correlation [NRJ04][XN04][QL04][Cro04] is a technique for making sense of a large number of events and pinpointing the few events that are really important in that mass of information. It has been notably used in Telecommunications and Industrial Process Control since the 1970s, in Network Management and Systems Management since the 1980s, in Service Level Management and Event-Based Systems since the 1990s, and in Business Activity Monitoring since the early 2000s. Event Correlation is implemented by a piece of software known as the event correlator [SM04][MS07] [Luc07]. This tool is automatically fed with events originating from the source. Each event captures something special (from the event source standpoint) that happened in the domain of interest to the event correlator. An event may convey an alarm or report an incident (which explains why *event correlation* used to be called *alarm correlation*), but not necessarily. It may also report that a situation goes back to normal, or simply send some information. The severity of the event is an indication given by the event source to the event destination of the priority that this event should be given while being processed. The practice of event correlation is useful and necessary not only to reduce the number of alarms but also to do some processing of the likely causes to take some of the workload off of the network engineer.

Event correlation can be performed over streams besides event clouds [Bas07][MS07]. Over streams, event correlation deals with the task of processing multiple streams of event data with the goal of condensing the streams, identifying meaningful events within the streams and performing reasoning based on the streams. Together with other event stream processing (ESP) techniques, event correlation is utilized in the data stream applications such as algorithmic trading in financial services, RFID event processing applications, fraud detection, process monitoring, and location-based services in telecommunications. Obviously, ESP techniques can be applied in event stream correlation. On the other hand, as pointed out by [CA08], the typical CEP techniques on event clouds (such as using event detection graphs and a data flow architecture) is similar to the processing of data streams. Thus, we can anticipate that ESP techniques can be beneficial to major event correlation algorithms working over both general event clouds and event streams.

The research work in [XN04][NRJ04] applies an event correlation framework for streaming network management events. The proposed correlation approach is based on triggering events and common resources for event processing in the network security domain. One of the key concepts in such correlation framework driven by streaming events is data triggering, which produces the (low-level) events that trigger alerts. By grouping alert events that share "similar" triggering events, a set of alert events can be partitioned into different clusters such that the alerts in the same cluster may correspond to the same attack. The alert events in each cluster are consistent with relevant network and host configurations, which help analysts to partially identify the severity of alerts and clusters.

Upon receiving events, the event correlator discards those that it deems irrelevant. Next, it merges duplicate events and aggregates events that globally tell the same story. Finally, the event correlator performs Root Cause Analysis to identify, through dependency analysis, what events can be explained by a single one (the root cause). At this stage, the event correlator is left with at most a handful of events that need to be acted upon. Event Correlators also include problem-solving capabilities, in order to be able to trigger corrective actions or further investigations automatically. Event correlation is defined in many different ways, but in its barest essence, an event correlator attempts to do exactly as the name suggests: associate events with one another in useful ways. An event stream correlation can be decomposed into the following steps:

- Event Filtering.
- Event Aggregation.
- Root Cause Analysis.

Event Filtering consists in discarding events that are deemed to be irrelevant by the event correlator.

Event Aggregation consists in grouping events that match specific patterns. Based on rules defined on the temporal, spatial and other predicates of the events, this action reduces multiple occurrences of the similar event into a single event, likely with some kind of counter, or grouping corresponding events into a composite single event.

Root Cause Analysis is the last and most complex step of Event Correlation. It consists in analyzing dependencies between events, based on a model of the environment and dependency graphs, to detect whether some events can be explained by others.

A somewhat traditional approach to event correlation is that of rule-based analysis [Luc07][CJC06][Cro04]. In this approach, sets of rules are matched to events when they come in. Based on the results of each test, and the combination of events in the system, the rule-processing engine analyzes data until it reaches a final state. It will then report a diagnosis. For the results to be accurate, an excessive amount of expert knowledge is typically needed to input the correct rules and keep them updated in case of any changes or new data.

An approach that is radically different from the rule-based approach uses the *artificial intelligence* (AI). Event correlation frameworks have been proposed utilizing various combinations of the AI techniques, including Bayesian belief networks and expert systems. AI systems have an advantage in that, if well-programmed, they have the capability to be somewhat self-learning, helping to eliminate the continuous need for the expert knowledge of the previous systems. They also have the capability to sift through data at least as fast as the other systems to produce their results [FNS+99][BE04].

5.2.2 Using Intervals for Optimization

Many factors come into play in the process of event correlation. The two most important aspects of an event correlation system are the speed with which the event correlator response, and the accuracy of the data returned by a correlation. A system must have an appropriate combination of these two characteristics in order to be considered effective.

For the purpose of correlating events in the complex event processing, context consists of time, space, and semantic circumstances in which the events are all to be considered. Because many instances of relations can be determined through temporal information, temporal relations among events play an important role in a causal system. In many cases, there is a time period associated with possible correlations, which requires proper events occur during a particular time period for them to be correlated.

For a business application, an event is triggered when certain status with the event publisher has a meaningful change. An event interval is a period between two events triggered by the event publisher. Within an event interval, the status with the event publisher should remain the same. We can use event intervals to optimize the correlation operation for complex event processing to improve system accuracy and efficiency.

Let's take network management as an example. As the world relies on computers to do increasingly important and complicated tasks, the field of network management becomes ever more important. Whether ensuring that the e-commerce servers are constantly up and accessible or simply making sure that company executives can send and receive e-mail, the network engineers are heavily relied upon to ensure consistency. Whenever there is a problem, the engineers in charge of maintaining the network need to be able to quickly pinpoint the source of the problem, whether it is as simple as a stopped mail daemon or as complicated as a fiber cut between satellite offices. The specialization of event correlation aids in this endeavor, by attempting to consolidate the information received into a concise, clear package that can be quickly deciphered.

An event in network management is typically defined as a piece of information dealing with a happening in the network, and may also be referred to as an alarm, due to its nature usually being something causing problems. The network management system can be programmed to methodically obtain these events by polling devices, the devices can send events to the management system, or, as is commonly the case, a hybrid combination of the two is used. Examples of events are hardware or software failures, security violations, and performance issues.

Example 5.2. In network ABC, the SV-I server broke down (server failure). The root cause is traced by a link of events (such as a series of device/function failures and sensor readings) with particular patterns. By analysis, one possible root cause is determined to be network attacks. The event correlator try to find out the target attacks by associating the given server failure event with consecutive high temperature events from the temperature sensor, which can represented as *SEQ*(*NETWORK_ATTACK*, consecutive *TEMPERATURE_READING* with *value* > *120F*). We assume that the high temperature events can be furtherly correlated with either fan failure events and server overload events, which are in the link of tracing back to network attacks (*Figure 5.1*). We can see that the fan failure actually represents an event interval of consecutive temperature readings. Within an interval (either the fan failure or server overload), the status with the event publisher(the temperature reader) remains the same, which keeps producing high temperature readings. Thus, we can transform the correlation rule by using the interval events of fan failure and server overload by two patterns, BEFORE(*NETWORK_ATTACK*, *FAN_FAILURE*) and BEFORE(*NETWORK _ATTACK*, *SERVER_OVERHEAT*), to trace the target network attack events. With up-to-date CEP systems, the rules using the interval events can be executed more efficiently and more accurately.

5.3 Interval Event Stream Processing

5.3.1 Interval Event Query Model

An interval event e can be represented as two separated point-based events using its two endpoints, namely start endpoint (e^-) and termination endpoint (e^+) [Tom96][RB06]. We assume a data model in which an interval

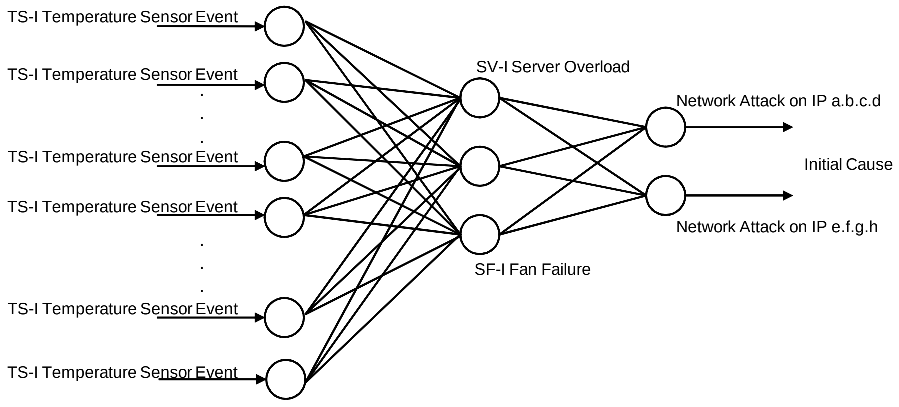

TS-I Temperature Sensor Event
TS-I Temperature Sensor Event
TS-I Temperature Sensor Event
TS-I Temperature Sensor Event
TS-I Temperature Sensor Event
TS-I Temperature Sensor Event
SV-I Server Overload
SF-I Fan Failure
Network Attack on IP a.b.c.d
Network Attack on IP e.f.g.h
Initial Cause

Figure 5.1: Network Event Correlation

event is an atomic unit semantically. Thus an interval event is composed fully after it ends and it arrives at the ESP system after it is completed. We will have further discussion in *Chapter 5.4* for the data model in which an interval event is composed by two atomic point events.

In the following discussion we assume that the timestamps of the events are globally ordered, reflecting the ordered semantics of the physical events. In case of disordered arrival of the events at the ESP engine, the mechanism introduced in [LLD^{+}07] can be applied after some further adjustments and it will not affect the correctness of the basic approach introduced in this chapter. Further discussion will be given in *Chapter 6.1*. Each event is assigned a timestamp from a discrete ordered time domain. We assume that such timestamps are assigned by a separate mechanism before events enter the event processing system and that they reflect the true order of the occurrences of these events. For an ordered interval-based event stream, the event receiving order at the ESP system is the same as the order of the end time of the event instances.

For fully supporting event processing over interval streams, the query algebra and evaluation corresponding to detecting temporal relations among events need to be adjusted. Similar to the discussion in *Chapter 4*, only SEQ, NEGATION and WIN need to be adjusted for handling interval streams. The logical operators such as AND / OR (which detect patterns with logical relations) apparently do not need to be adjusted for interval handling. The SELECT operator (which performs value-based predicate checking) needs special adjustment only when it is associated with negation patterns. However that can be simply avoided by operator pushdown on the SELECT [WDR06]. The handling of negation over interval events is not covered in this dissertation so we will be focusing only on the SEQ and WIN operator.

The SEQ operator [WDR06][ACT08] handles only the "*before*" / "*after*" temporal relation (treating a point event as an interval with the same start and end time timestamps). Because of the transitive property of the "*before*" / "*after*" relation, this basic two-arguments operator can be extended to handle sequence with three or more event as $SEQ(E1, E2, E3, ...)$, which indicates that an $E1$ event is followed by an $E2$ and the $E2$ event is followed by an $E3$ event, and so on. For example, $SEQ(A, B, C)$ detects a sequential event patterns $<a, b, c>$ where a is an event instance of type A, b is an event instance of type B, c is an event instance of type C, a *before* b and also *b before c*.

As we have pointed out in *Chapter 5.1*, additional temporal relations can be defined between two interval events. The reason is that two non-equal interval events can have overlapping portion instead of a simple sequential relation. The relations between two intervals can be expressed in terms of relations between their endpoints, i.e., the start and end of the interval. Under interval-based temporal semantics, relation between two event instances can be very flexible.

We consider a *temporal relation* to be a relation among two or more interval events, which can be divided into two different categories, namely, *closed endpoint relation* and *open endpoint relation*.

A temporal relation where the order of all endpoints of the events are fixed is called a closed endpoint relation, which falls into the categories given by [All83]'s classification (discussed earlier in *Chapter 5.1*). We refer to these temporal relations as *Allen-based relations*.

A temporal relation where the order of all endpoints of the events are not fully fixed is called an open endpoint relation. Real world applications might have customized requirement on interval pattern detection where open endpoint relations are needed to be defined. For example, we can define temporal relation R as "intervals of type $E1$ starts before intervals of type $E2$". The temporal algebra of such pattern is as $e1.ts < e2.ts$, where $e1$ is an instance of $E1$ and $e2$ is an instance of $E2$. We can see that such temporal relation is the disjunction of several closed endpoint relations. For example, temporal relation R defined above is equivalent to the disjunction of several closed endpoint relations as $(E1\ \textit{before}\ E2) \vee (E1\ \textit{meets}\ E2) \vee$

($E1$ *overlaps* $E2$) $\vee$ ($E1$ *finished by* $E2$) $\vee$ ($E1$ *contains* $E2$).

To express temporal relation between two intervals (referred to as *primitive temporal relation*), the simple SEQ operator becomes insufficient because it only considers "*before*" / "*after*" as temporal relations over point-based events. One approach to define a primitive temporal relation is simply using the 13 Allen-based relations and their disjunction using the syntax Rel[*list of Allen-based relations*]($E1$, $E2$), where *Rel* is a temporal operator for interval. The temporal semantics of *Rel* is defined by a list of Allen-based relations. For example, Rel[*overlap*](A, B) represents the *overlap* relation in Allen's model. Rel[*before,meet,overlap,finished by,contain*](A, B) represents an open endpoint temporal relation which is the disjunction of five different Allen-based relations.

While the expressiveness of such relation representation is no longer sufficient if it is extended to represent temporal relation among three or more intervals (referred to as *composite temporal relation.* as Rel[*list of Allen-based relations*]($E1$, $E2$, $E3$, ..., Em). One reason is that some temporal relation might not satisfy the transitive property (such as *overlap*). Take Rel[*overlap*](A, B ,C) as an example. Given three interval event instances a of type A, b of type B and c of type C, "a *overlap* b and b *overlap* c" cannot infer "a *overlap* c" because that *overlap* relation does not have the transitive property. This representation cannot express the pattern such as "A *overlaps* B, B *overlaps* C and A *overlaps* C". Another reason is that a composite relation might contain more than one temporal relations, such as a composite relation R defined as "A *contains* B and B *overlaps* C".

A *hierarchical representation* [KF00] is proposed to encode composite relations. Similarly, we can extend our previously defined operator syntax to represent a composite relation with multiple temporal relations as $\mathrm{Rel}_n(...\mathrm{Rel}_2(\mathrm{Rel}_1[\textit{list of Allen-based relations}](E1, E2), E3),..., Em)$. It can express composite relation such as "*A contains B*, which as a composite event, *overlaps C*". However, such representation is still not expressive enough as it lacks the ability to represent pair-wise relations among events. It still cannot express relation such as "*A contains B* and *B overlaps C*".

We introduce the endpoint-based encoding mechanism to represent temporal relations among event intervals, inspired by [NB94][NB95], where an endpoint sequence representation for intervals is studied. The basic idea is to express a relation using the conjunction of *temporal restriction*, which restricts the temporal relation to <, <=, =, >, >= between two interval endpoints. Such conjunction representation is called a *temporal restriction list* (*TList* in short). TList is with the syntax as "TList ::= TList∧TList | $I_1{}^*{<}I_2{}^*$ | $I_1{}^*{<=}I_2{}^*$" | $I_1{}^*{=}I_2{}^*$", where $I_1{}^*$ and $I_2{}^*$ define two endpoints. Although that the same expressibility can be achieved using the 13 Allen-based relations and their disjunction to define the primitive relations for intervals among the composite pattern, the above endpoint-based language has the advantages of simplicity in use and it is closer to the business rules of the temporal-based real world applications [NB94][NB95].

Example 5.3. For example, the event relation "A starts earlier than B" is simply represented as "$A.ts < B.ts$" and the event relation "A overlaps B, B overlaps C" is represented as "$(A.ts < B.ts) \wedge (B.ts < A.te) \wedge (A.te < B.te) \wedge (B.ts < C.ts) \wedge (C.ts < B.te) \wedge (B.te < C.te)$".

Please note that if there exists any conflicts (such as *ep1* $>$ *ep2* $\wedge$ *ep1* $<$ *ep2*, where *ep1* and *ep2* are two event endpoints) in the TList, it becomes invalid. We assume a validating process thus all the TLists in this dissertation are considered valid.

Our proposed encoding mechanism is with the syntax as "EVENT *ISEQ* [TList](E1, E2, E3, ..., Em; W)", where ISEQ is the temporal operator with the following semantics:

$$\begin{aligned}
&ISEQ[TList](E1, E2, ..., Em; W)[H] \ = \\
&\quad \{< e1\ e2\ ...\ em > \mid (TList(e1, e2, ..., em)) \wedge \\
&\quad (< e1\ e2\ ...\ em > \in E1[H] \times E2[H] ... \times Em[H]) \wedge \\
&\quad (max(ei.end)_{i \in \{1,2,...,m\}} - min(ej.start)_{j \in \{1,2,...,m\}} < W)\}.
\end{aligned} \tag{5.1}$$

In the ISEQ operator given above, $\{E1, E2, ..., Em\}$ is the set of event types defined in ISEQ and the TList defines the endpoint relation among the instances. An occurrence number will be attached to distinguish multiple occurrences of the same event type. Given E and E' defined in ISEQ, maximum four different temporal restrictions could be defined: *(1)* $E.ts$ Rel_1 $E'.ts$, *(2)* $E.ts$ Rel_1 $E'.te$, *(3)* $E.te$ Rel_1 $E'.ts$ and *(4)* $E.te$ Rel_1 $E'.te$. Rel_1 to Rel_4 are among possible point-based temporal relations $<$, $<=$, $=$, $>$ and $>=$ ('$>$' is the mirror relation of '$<$' and '$>=$' is the mirror relation of '$<=$'). For any given E in ISEQ, $E.ts <= E.te$ is always a required temporal restriction. A reasoning framework on the endpoint-based temporal representation is studied in [NB94][NB95]. It introduces an algorithm with exponential complexity which can be used to infer the temporal relation between two endpoints based on a given set of temporal restrictions. Queries defined by the ISEQ using the TList can also be expressed by the disjunctions of the Allen's algebra operators [Tom96]. Consider *Example 5.3* given earlier on temporal relation R as "intervals of type A starts before intervals of type B". The TList for R is "$A^- < B^-$", which implies "$(A\ \textit{before}\ B) \vee (A\ \textit{meets}\ B) \vee (A\ \textit{overlaps}\ B) \vee (A\ \textit{finished by}\ B) \vee (A\ \textit{contains}\ B)$".

In a traditional point-based event query algebra, the window constraint specification is expressed as the time window parameter, used for restricting the duration length among events in the temporal pattern. In [WDR06], the window expression gives the time window argument W, which specifies the maximum time duration between the occurrences of the first and last events in the event temporal pattern. We adopt the operator pushdown approach in [WDR06] to handle the window-based filtering in ISEQ, which uses the window size W to control the maximum span of the result composite, defined as $\max(ei.end)_{i\in\{1,2,...,m\}}$ - $\min(ej.start)_{j\in\{1,2,...,m\}}$.

5.3.2 ISEQ Operator

The physical implementation of ISEQ has three core operations listed below:

Event Buffering. A newly received event instance is buffered in the operator state of ISEQ if it is necessary. Given a newly received event interval e of type E which is among the set of expected events $\{E1, E2,...., Em\}$, e needs to be buffered into a stack structure referred to as the *instance stack* if and only if it is possible to form result tuples using e together with some other received interval or future coming intervals. So, if E is with a given or inferred temporal restriction as $E.te > E'.te$ and no event instance of E' is currently buffered, the condition that requires an instance to be buffered is not satisfied thus e can be discarded directly without buffering (referred to as *on-the-fly dropping*). For other cases, e is added to the corresponding stack for buffering unless its interval length is larger than the window size (in such case the even will be discarded without consideration).

Result Construction. The result construction is performed on the fly triggered by newly arrived tuples to ISEQ. Given a newly received and buffered event interval e of type E among expected event types, new results could possibly be constructed if and only if e might be contained by a result composite event consisting of currently received instances. So, if E is not with a given or inferred temporal restriction as $E.te < ep$, $E.te = ep$ or $E.te <= ep$, where ep is another interval endpoint, e could then possibly contribute in forming new result sequences consisting of the current buffered intervals. So the result construction condition is satisfied thus the construction process triggered by e can be called. The process uses a multi-join algorithm based on the attribute constraints on the interval endpoints defined by TList is applied to construct possible composite events. In the join process, the values of event endpoints (both the start endpoint and termination endpoint) are used if the endpoints are associated with some temporal restrictions or with the window constraint.

Operator State Purge. Window constraints can be utilized in ISEQ to avoid unnecessary event buffering. It provides opportunity to dynamically purge events from the ISEQ operator when the event has fallen out of the sliding window. The latter is important in stream processing where runtime data structures need to be pruned to avoid memory depletion. Memory footprint is reduced due to such pruning. In addition, if the checking overhead is kept to be small, CPU footprint can also be reduced because of the saving on buffering-related operators and result construction Furthermore, similar to pushing down the window constraint into SEQ operator in [WDR06], if the purge at ISEQ is conducted on a timely fashion, the checking on window constraint could be skipped thus the corresponding computation for window-based filtering is avoided in the result construction phase. Given a buffered event interval e of type E among expected event types, e can be safely purged from the buffer if and only if it is no longer contributing in forming new results. So, if the termination endpoint associated with E is in given or inferred "$>$" temporal restrictions with all the endpoints in the pattern except itself, the purge condition is satisfied and the event instance e can be purged from the buffer once the result construction process triggered by e (if any) is completely finished. A window constraint-based purge named *cascading purge* could be performed: if E is with a given or inferred temporal restriction as $E.te > E'.te$ and the stack for E' events is empty, all the E events can be safely removed. The process can go on following the chain of such temporal restrictions on the interval termination endpoints. While a fine-grained duration constraint [NB94][NB95] defined in ISEQ, it can be utilized to furtherly avoid unnecessary event buffering. The basic idea is dynamically checking the window constraint while a new interval instance e of type E is received. For a buffered interval ei of Ei with a duration restriction as $Ej.te - Ei.ts(te) <$ W, if $e.te$ - $ei.ts(te) >$ W, ei can be purged from the operator state of ISEQ. The correctness of this window-based purging mechanism is shown as follows. By the arrival of e, we can know any interval e' coming in the future satisfies $e.te < e'.te$. Thus, any future Ej instance ej will satisfy $ej.te - ei.ts(te) >$ W. So, e is guaranteed to no longer contribute in forming new query results. In this dissertation we will not consider data purge on such duration constraint since we only consider the window semantics defined in *Chapter 5.3.1*.

An optimization can be brought into this process. Remember that we assume the input interval stream is ordered and the event receiving order at the ESP system is the same as the order of the end time of the event instances. Such order semantics of the input intervals can be utilized to reduce the join computation in the result construction of ISEQ. This is similar to the idea of using a runtime stack nondeterministic finite automaton (NFA) for pattern retrieval on point-based events [WDR06]. The optimization is for avoiding the multi-join on the longest path of termination endpoints linked through temporal restrictions. Let N denote the length of the path. Then the number of states in the NFA equals N+1 (including the starting state). A data structure named AIS as discussed earlier in *Chapter 4* associates a stack with each state of the NFA storing the events that trigger the NFA transition to this state. For each instance ei in the stack, an extra field named RIP records the nearest instance in terms of time sequence in the stack of the previous state to facilitate sequence result construction. When

Algorithm 11 Basic ISEQ Operations

```
1: Procedure: ISEQOperation
2: Input:
3: (1) event Query EVENT $ISEQ[TList](E1, E2, ..., Em; W)$,
4: (2) newly received event $e$ (under event type $E$)
5: Output:
6: matched result sequences triggered by the input event instance
7:
8: compute the inferred temporal restrictions
9: form the DAG $G$ representing the temporal restrictions
10: compute the indexing scheme for AIS-based approach
11: if CLOCK updates then
12:     perform window-based purge
13:     perform corresponding cascading purge
14: end if
15: checkState = true
16: if $E$ is among $E1, E2, ..., Em$ then
17:     if $e.te$ - $e.ts < W$ then
18:         if $E$ is with a given or inferred temporal restriction as $E.te > E'.te$ then
19:             if no event instance of $E'$ is currently buffered then
20:                 checkState = false
21:             end if
22:         end if
23:         if CheckState then
24:             buffer $e$ into the corresponding AIS stack if indexing is applied on $E$ for the
                AIS-based approach and into the corresponding instance stack otherwise
25:             if $E$ is not with any temporal restriction as $E.te < ep$, $E.te = ep$ or $E.te <=$
                $ep$ in $G$, where $ep$ is a vertex in $G$ and $ep \neq E.te$ then
26:                 produce event sequences containing $e$ (if any) by corresponding join algorithm
27:             end if
28:             if $G$ covers all the endpoints in the pattern and $E$ is with a temporal restriction
                as $E.te > ep$ for any $ep \in G$ and $ep \neq E.te$ then
                    purge $e$
29:             end if
30:         end if
31:     end if
32: end if
```

the newly inserted event is an instance of the final stack then AIS computes sequence results. With the AIS states, the construction is simply done by a depth first search in the AIS stacks that is rooted at this instance and contains all the virtual edges reachable from this root. Each root-to-leaf path in the AIS stacks corresponds to the complete or a portion of a matched event sequence, which will be constructed by the rest of the multi-join process after the AIS-based construction. With such AIS data structure, a more sophisticated cascading purge named *cascading AIS purge* could be performed: once an event instance is purged from the AIS stack, events whose RIP field pointing to this event can also be purged.

Algorithm 11 depicts the key ISEQ operations described above. Upon the arrival of a new event interval, buffering decision is made and possible result sequences are produced at the earliest moment. Window-based and cascading purge are performed triggering by the CLOCK updates *Line 11*. The CLOCK value is introduced earlier in *Chapter 4*, which equals to the largest end time timestamp seen from the received intervals. The given and inferred temporal restrictions are managed as a DAG structure [Koz03], with the edges marked as either ">", ">=" or "=". Corresponding construction supports for applying the AIS data structure is given in *Line 10* and *24*. In addition to that, specific AIS-incorporated computation (*Line 13* and *26*) are plugged in for utilizing the indexing structure.

Example 5.4. Consider event pattern query $Q = ISEQ[A^- < B^+ < C^+ < D^+](A, B, C, D)$ and interval event trace $S =$ "$b_{3|6}$, $d_{6|10}$, $b_{9|11}$, $c_{4|12}$, $a_{7|14}$, $d_{9|15}$, $a_{8|16}$" (shown in *Figure 5.2*). Remind that by the notation given in

Chapter 2, for an interval event instance e, we use a pair of numbers as $e_{t1|t2}$ adjacent to it to represent its timestamp (denoting both the start and end time). Interval instance $d_{6|10}$ will be discarded upon arrival through the on-the-fly dropping since no C events are currently buffered and between C and D there is a temporal restriction defined as $C^+ < D^+$. While $d_{9|15}$ arrives, the result construction is triggered to produce a result sequence $<a_{7|14}$ $b_{9|11}$ $c_{4|12}$ $d_{9|15}>$. While $a_{8|16}$ arrives, the construction process is triggered again, producing another result sequence $<a_{8|16}$ $b_{9|11}$ $c_{4|12}$ $d_{9|15}>$. Assuming that the window size W equals to 30 and we furtherly receive interval $e_{20|35}$, $b_{3|6}$ and $c_{4|12}$ can then be safely purged from the operator state.

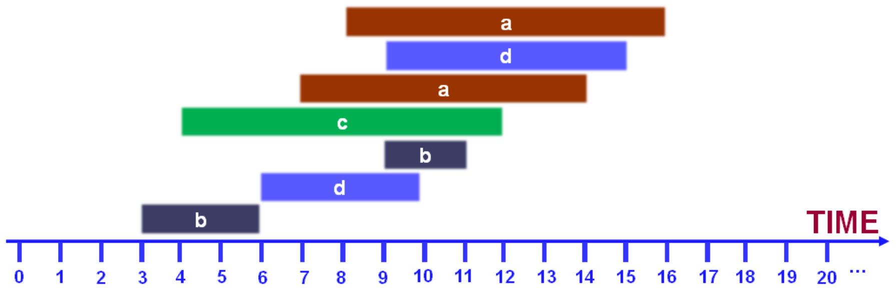


Figure 5.2: Example Interval Event Input

5.3.3 Query Evaluation Strategy

Algorithm 12 sketches my proposed execution strategy for a long running process of interval event pattern detection. The monitoring process is stopped when the event trace is terminated. Corresponding CPU and buffer resources could not be released earlier. During the monitoring process, each received event triggers data buffering, result construction and operator state purge following *Algorithm 11* given in *Chapter 5.3.2*.

Algorithm 12 Execution Strategy for Interval Event Stream Processing

```
 1: Procedure: IntervalProcessingExecutionStrategy
 2: Input: real-time evolving interval sequence trace seq as "e1, e2, e3 ..." by the order of
    their termination endpoint, with the End of Stream (EOS) message arriving at the very
    end if input termination is indicated
 3: Output: matched result sequences
 4:
 5: var $e \leftarrow poll(seq)$
 6: while $e \neq$ EOS do
 7:     process $e$:
 8:     perform necessary data buffering and state purge, construct new results if possible
        based on Algorithm 11
 9:     $e \leftarrow poll(seq)$
10: end while
11: terminate the pattern monitor for the current event trace
```

5.4 Towards Efficient Interval Processing

5.4.1 Using Punctuations

In many ESP applications, event intervals are actually extracted from the raw primitive point-based events (such as the RFID sensor readings) by business intelligence (referred to as BI) middlewares [vAEE$^+$09][PV09][Luc07] and then passed to the downstream ESP systems. Consider the previous example given in *Chapter 5.1* where an ESP system is used to monitor the events generated by warehouses of a supermarket. Based on the temperature values sent by the temperature readers, temperature fluctuations, *HIGH*, *MEDIUM* and *LOW*, as the interval events are generated and sent to the ESP system. In real world applications, such temperature fluctuation intervals are actually extracted by the middleware systems which receives the actual readings from the temperature sensors. Let's assume the *HIGH* temperature is above 100F, the *MEDIUM* temperature is within the range of (50F, 100] and the *LOW* temperature is 50F or lower. We also assume the sensor reads temperature every two seconds and reports the following reading: 01:00:00PM - 55F; 01:00:02PM - 70F; 01:00:04PM - 95F; 01:00:06PM - 80F; 01:00:08PM - 110F; 01:00:10PM - 120F; 01:00:12PM - 90F; 01:00:14PM - 60F; 01:00:16PM - 30F... The interval streams generated will be: "*MEDIUM*(01:00:00PM, 01:00:08PM), *HIGH*(01:00:08PM, 01:00:12PM), *MEDIUM*(01:00:12PM, 01:00:16PM)...". Assume we are having another two interval events, *WET* and *DRY*, to represent the humidity of the environment generated under a similar sensor layer as the one used for the temperature readings. By such context, a practical event query can be looking for the pattern of *HIGH* overlaps *DRY*. Such corner changes which trigger new intervals are called *critical state changes*. These critical state changes (as the begin and end of an interval) are captured by the BI middlewares and then composed into interval events and passed to the downstream ESP systems once the interval is fully formed. Thus, under the above application structure, the information of the "interval start" is actually known to the BI middleware at real-time. A mechanism to improve the efficiency of interval stream processing is to embed a punctuation named *interval begin punctuation* into the interval event stream. The proposed punctuation is defined as follows:

Interval Begin Punctuation(IBP). IBP indicates the initialization of an interval instance. At the moment an interval event e starts, its corresponding IBP will be created and sent. It has associated a metadata schema as ibp_e = $<e.id, e.ts>$, where $e.id$ is the ID value of e, assigned automatically by the EPS. The ID value is unique among the events in the stream. Given an

IBP *p*, its timestamp *p.t* equals to *e.ts*.

In the discussion in *Chapter 5.3.1*, a data model in which an interval event is an atomic unit semantically is assumed. Thus an interval event is composed fully after it ends and it arrives the ESP system after it is completed. Applying IBP does not require the change of this model. However, the IBP information can be used for effective interval event processing. The interval event sender (i.e., the BI middlewares as shown in our earlier example) should have the mechanism to encode an unique ID to the event intervals. The ESP system receives interval event streams mixed together with IBPs. Remember that we assume order for the input interval stream. Under such model which interleaves IBPs with event interval instances, the order of receiving events and IBPs at the ESP system is the same as the order of their end time timestamp. Note that since IBPs are point-based data, the time stamp of an IBP equals to its end time timestamp.

An IBP-aware interval event processing approach can greatly reduce the runtime memory and CPU footprint for temporal pattern detection over interval-based event streams. The key operations of an IBP-incorporated ISEQ operator is given as below.

Event Buffering. The event buffering conditions in the basic ISEQ stays. However, the IBP information is also hold in the AIS for the events being indexed. We will have further discussion on this in the result construction. With the IBP information being available, additional on-the-fly event dropping becomes possible, as follows. Given a newly received IBP of E interval e, which is among the set of expected events $\{E1,\ E2, ..., Em\}$, if $E.ts$ is among the indexed start endpoints (referred to as the IBP of E being indexed) and *(1)* the AIS stack pointed by the AIS stack of ibp_e is empty, or *(2)* E is with a given or inferred temporal restriction as $E.ts > E'.te$ and no events of E' is currently buffered, the received IBP can be dropped without buffering. Given a newly received interval instance e of type E which is among the set of expected events $\{E1, E2, ..., Em\}$, if *(1)* the IBP of E is required to be indexed and no IBP entry corresponding to e is currently buffered, or *(2)* E is with a given or inferred temporal restriction as $E.te > E'.ts$, the IBP of E' is required to be indexed and no IBPs of E' is currently buffered, or *(3)* E is with a given or inferred temporal restriction as $E.te > E'.te$ and no events of E' is currently buffered, the condition that requires the e instance to be buffered is not satisfied thus e can be discarded directly without buffering.

Result Construction. As discussed earlier, the result construction is performed on the fly triggered by newly arrived tuples to ISEQ. Given a newly received and buffered event interval e of type E among expected event types, new results could possibly be constructed if and only if e might be contained by a result composite event consisting of currently received instances. The conditions for result construction triggering in the basic ISEQ stays for the IBP-incorporated ISEQ. So, if E is not with a given or inferred temporal restriction as $E.te < ep$, $E.te = ep$ or $E.te <= ep$, where ep is another interval endpoint, e could then possibly contribute in forming new result sequences consisting of the current buffered intervals. So the result construction condition is satisfied thus the construction process triggered by e can be called.

For the part without AIS indexing, the process uses a multi-join algorithm based on the attribute constraints on the interval endpoints defined by TList is applied to construct possible composite events. In the join process, the value of event endpoints (both the start and termination endpoints) are used if the endpoint is associated with some temporal restriction or with the window constraint. The AIS stack is brought into the multi-join process for the indexed temporal restrictions to avoid the joins on a path of event endpoints (both the start and termination endpoints) linked through temporal restrictions using not only the interval termination but also the IBPs. This is different than the AIS-based approach in the basic ISEQ, where the IBPs are not available. The path with the most join avoidance will be selected, which is the longest path in the DAG formed by the temporal restriction, and it is not counted as one join if an edge is formed by one single event type. For event types with only indexed termination endpoints, the AIS structure remains the same as the basic ISEQ operator. For event types with only indexed IBPs, a corresponding AIS stack at first holds the IBPs and later is filled with the corresponding full instances. The RIN pointers introduced in *Chapter 4.5* can be applied to the stacks consisting of the IBP entries. If the path includes both the start and termination endpoints of an event type, two different AIS stacks will be applied and they both link to a shared structure (referred to as the *full edge stack*) holding the event instance. The construction on the indexed path remains as a simple depth first search in the AIS stacks that is rooted at this instance and contains all the virtual edges reachable from this root. Each root-to-leaf path in the AIS stacks corresponds to the complete or a portion of a matched event sequence, which will be constructed by the rest of the multi-join process after the AIS-based construction.

Operator State Purge. The conditions for operator state purging in the basic ISEQ stays for the IBP-incorporated ISEQ. So, if the termination endpoint associated with E is in given or inferred ">" temporal restrictions with all the endpoints in the pattern except the ones from E itself, the purge condition is satisfied and the event instance e can be purged from the buffer once the result construction process triggered by e (if any) is completely finished. However, more purging opportunities become possible with the IBP being available: the window-based purge and the corresponding cascading purge can be simply extended to cover the IBPs kept in the AIS stacks. The benefits of doing so is that it can lead to further on-the-fly dropping since there could be fewer IBPs kept in the indexes after the purge.

Algorithm 13 depicts the corresponding operations given above. We can see that upon the arrival of a new event interval and an event IBP, buffering decision is made and possible result sequences are produced at the earliest moment. Similar to the basic ISEQ, upon the arrival of new event intervals, corresponding construction and operator state purge are triggered to performed. The query execution strategy based on ISEQ given in *Algorithm 12* stays the same for the IBP-incorporated ISEQ.

Example 5.5. Again consider the scenario given in *Example 5.4*. Interval event $b_{3|6}$ can be discarded without buffering, since we can guarantee that no future arrival of A could have a start time smaller than $b_{3|6}$'s end time,

Algorithm 13 IBP-Incorporated ISEQ Operations

```
1: Procedure: ISEQPlusOperation
2: Input:
3: (1) event Query EVENT ISEQ[TList](E1, E2, ..., Em; W),
4: (2) newly received event IBP $ibp_e$ / event instance $e$ (under event type $E$)
5: Output: matched result sequences triggered by the input event instance
6:
7: same as Line 8 to 14 in Algorithm 11
8: if $E$ is among $E1$, $E2$, ..., $Em$ then
9:    on the arrival of $ibp_e$:
10:    checkState = true
11:    if the IBP of $E$ is required to be indexed then
12:        if the AIS stack pointed by the AIS stack of $ibp_e$ is empty then
13:            checkState = false
14:        end if
15:        if $E$ is with a given or inferred temporal restriction as $E.ts > E'.te$ && checkState
           && no events of $E'$ is currently buffered then
16:            checkState=false
17:        end if
18:    end if
19:    if checkState then
20:        buffer $ibp_e$ into the corresponding AIS stack by the append semantics
21:    end if
22:    on the arrival of $e$ instance:
23:    checkState = true
24:    startFlag, endFlag = false
25:    if $e.te$ - $e.ts$ < W then
26:        if the IBP of $E$ is required to be indexed then
27:            startFlag = true
28:            if no IBP entry corresponding to $e$ is currently buffered then
29:                checkState = false
30:            end if
31:        end if
32:        if the full instance of $E$ is required to be indexed then
33:            endFlag = true
34:        end if
35:        if checkState && $E$ is with a given or inferred temporal restriction as $E.te > E'.ts$
           && the IBP of $E'$ is indexed and no IBP of $E'$ is currently buffered then
36:            checkState = false
37:        end if
38:        if checkState && $E$ is with a given or inferred temporal restriction as $E.te > E'.te$
           && no event instance of $E'$ is currently buffered then
39:            checkState = false
40:        end if
41:        if checkState then
42:            if startFlag && !endFlag then
43:                insert $e$ into the corresponding AIS entry based on the event ID
44:            else
45:                if !startFlag && endFlag then
46:                    buffer $e$ into the corresponding AIS stack by the append semantics
47:                else
48:                    if startFlag && endFlag then
49:                        buffer $e$ into the full edge stack and buffer $e$'s reference into the corre-
                          sponding AIS stack by the append semantics, update existing AIS
50:                    else
51:                        buffer $e$ into the corresponding instance stack
52:                    end if
53:                end if
54:            end if
55:            if $E$ is not with any temporal restriction as $E.te < ep$, $E.te = ep$ or $E.te <=$
               $ep$ in $G$, where $ep$ is a vertex in $G$ and $ep \neq E.te$ then
56:                produce event sequences containing $e$ (if any)
57:            end if
58:            if $G$ covers all the endpoints in the pattern and $E$ is with a temporal restriction
               as $E.te > ep$ for any $ep \in G$ and $ep \neq E.te$ then
                   purge $e$
59:            end if
60:        end if
61:    end if
62: end if
```

using the fact that no IBPs of A is met before the arrival of $b_{3|6}$. Similarly, interval $b_{9|11}$ is required to be buffered, indicated by the IBP of $a_{7|14}$.

Adapting the Point-Based ESP Solution. As mentioned earlier in *Chapter 5.3.1*, for a model where an interval event is an atomic unit semantically, an interval event is composed fully after it ends and it arrives the ESP system after it is completed. However, for a model where an interval event can be broken down to two atomic point events without losing semantic information, an interval logic can be converted to a point-based logic, which simply uses point-based events to represent the critical changes (the start and end of an interval). Again consider *Example 5.1* given previously regarding the ESP application in supermarket warehouses event monitoring. If an interval event simply carry semantic information as the time period of the temperature / humidity condition, we can replace the given business logic purely using point-based events showing the critical changes happening in the raw reading. The pros for this approach is that a point-based event processing mechanism can be easily adapted thus an interval event query can be evaluated simply using the point-based query processing mechanism. However, the business logic based on intervals is no longer kept under such a framework thus it is not straightforward to representation of the BI rules for the event reasoning and processing. Actually, the *Algorithm 13* given above, which follows the interval-based data model, already provided a point-based query processing framework, where the IBPs serve as a part of the critical state change events. Any further optimization on top of a point-based ESP solution can be easily consolidated into this framework.

5.4.2 Pushing Down the Interval Event Abstraction

As mentioned earlier for the IBP-based solution, in many ESP applications, event intervals are actually extracted from the raw primitive point-based events (such as the RFID sensor readings) by BI middlewares and then passed to the downstream ESP systems. Following such application structure, the low level physical devices (i.e., the sensor network) with enough computing power would actually be able to capture these critical state changes. Such mechanism of pushing down the computation of interval event abstraction to the low level sensor network can greatly improve the efficiency and scalability for ESP applications with intense computing ability on the physical level devices. This is because that the computation happens much closer to the information source thus the cost of data transportation is avoided.

5.5 Performance Evaluation

5.5.1 System Implementation

Figure 5.3 shows the system architecture for incorporating the proposed out-of-order handling into the basic ESP system structure given in *Chapter 2.3.* The proposed ISEQ operator is added into the corresponding operator library containers. While the input is point-based events (seen as interval events each with the same start and end time timestamp) and the AIS indexing is applied, the ISEQ operator behaves exactly the same as the SEQ operator. Thus, it can be treated as an extended SEQ operator.

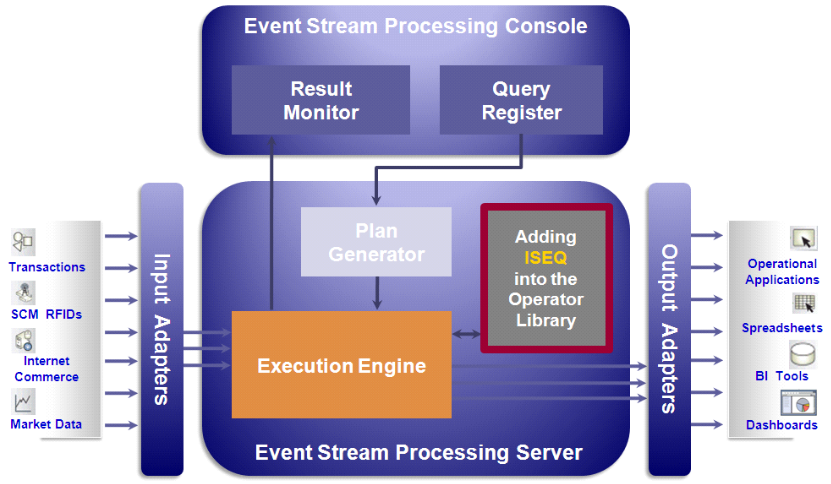


Figure 5.3: Interval Event-Incorporated ESP System Architecture

5.5.2 Experimental Setting

Experiments are run on two Pentium 4 3.0GHz machines, both with 1.98G of RAM. One machine sends the event stream to the second machine. From *Chapter 5.5.3* to *5.5.6* we are going to study the performance of the proposed interval event stream processing techniques on a 2G generated data input, which contains 20 different event types with uniform distribution.

Totally four sets of experiments are run to test the effects of various factors: *(1)* the indexing percentage that controls the indexable endpoints and endpoints types (either start or termination); *(2)* the query length that controls the number of interval patterns in the ISEQ operator; *(3)* the average interval length that controls the average span of the interval events with the normal distribution and *(4)* the event density that controls the number of events within one sliding window with the normal distribution. The applied queries are with the template as "EVENT *ISEQ*[TList](A, B, ... ; W)", where the TList defines the endpoint temporal restrictions among the event patterns. Performances of *(1)* the basic ISEQ without AIS indexing (referred to as *naive ISEQ*) approach, *(2)* the basic ISEQ with AIS indexing (referred to as *basic indexing*) approach and *(3)* the IBP-incorporated ISEQ (referred to as *IBP-incorporated*) approach are measured respectively. Experimental results are given from *Chapter 5.5.3* to *5.5.6* below.

5.5.3 Experiments with Varying Query Types

This set of experiments varies the percentage of indexable endpoints as well as the indexable endpoints types in the given query. The indexable endpoints will contribute to the AIS construction for the basic ISEQ with AIS indexing approach and the IBP-incorporated ISEQ approach. Ten different combinations are covered by the experiments, which is shown in *Table 5.2*. The query length is fixed as 10. The average interval length is fixed as W/10 (W is the sliding window size, which is fixed as 30 seconds for all queries) with the event density as 200 events per window. Results are shown in *Figure 5.4* and *5.5*. The property of the input event data such as the average interval length and event density greatly affects the performance, which will be studied later in *Chapter 5.5.5* and *5.5.6*.

Memory Consumption (*Figure 5.4*). X axis here shows the ten groups of queries categorized by indexing scheme discussed earlier (*Table 5.2*) and Y axis shows the accumulative memory consumption for each query. With the cascading AIS purge, the basic indexing approach and the IBP-incorporated

Name	**Description**
100t-0s	100% termination endpoint indexing
90t-10s	90% termination endpoint and 10% start endpoint indexing
80t-20s	80% termination endpoint and 20% start endpoint indexing
70t-30s	70% termination endpoint and 30% start endpoint indexing
60t-40s	60% termination endpoint and 40% start endpoint indexing
50t-50s	50% termination endpoint and 50% start endpoint indexing
40t-60s	40% termination endpoint and 60% start endpoint indexing
30t-70s	30% termination endpoint and 70% start endpoint indexing
20t-80s	20% termination endpoint and 80% start endpoint indexing
10t-90s	10% termination endpoint and 90% start endpoint indexing
0t-100s	100% start endpoint indexing

Table 5.2: Profiles for Different Query Types

approach both have less memory footprint than the naive ISEQ approach except the case with no termination endpoint indexing for the basic indexing approach. However it only shows a slight gain (less than 5% for the case with the most gain) under the given setting. With a smaller window, which can be achieved by increasing the average interval length or decreasing the event density, more memory footprint can be avoided. This will be furtherly discussed in *Chapter 5.5.5* and *5.5.6*. Addition to that, for the basic indexing approach, the gain on memory consumption is affected by the percentage of indexable termination endpoints in the query.

CPU Performance (*Figure 5.5***).** X axis still shows the ten different indexing scheme and Y axis shows the execution time for each query. We can see that the IBP-incorporated approach in all cases outperform the naive ISEQ approach. This is because that it has indexing support for all the query categories due to the IBP utilization. In most cases the basic indexing approach outperforms the naive ISEQ approach: with a higher percentage of the termination indexing, more CPU computation could be avoided in terms of result sequences construction using the costly multi-join algorithm. For example, in the best case (i.e., the query with 100% indexable termination endpoint patterns), execution with the basic indexing approach reduce the execution time of the plan with naive ISEQ by 60%. However, while the percentage of indexable termination endpoints is not high in the given query, the basic indexing approach has poor performance because the overheads on index construction and maintenance. The overhead ranges from 3% to 12% in the query categories of *20t-80s*, *10t-90s* and *0t-100s*. The overhead increases while decreasing the portion of indexable termination endpoints in the query. We can also observe that the basic indexing approach does not perform as well as the IBP-incorporated approach. This is due to the cost avoidance using the IBP information in the IBP-incorporated approach is not applicable for the basic indexing approach.

5.5.4 Experiments with Varying Query Length

This set of experiments studies how varying the relative query length affects the interval stream processing cost. The query length is varied from 2 to 18. For example, among them a sequence query with length 6 (i.e., $ISEQ[A^- < B^- < C^- < D^+ < E^+ < F^+](A, B, C, D, E, F)$) is run. The *50t-50s* indexing profile is applied to all the queries in this set of experiments. The

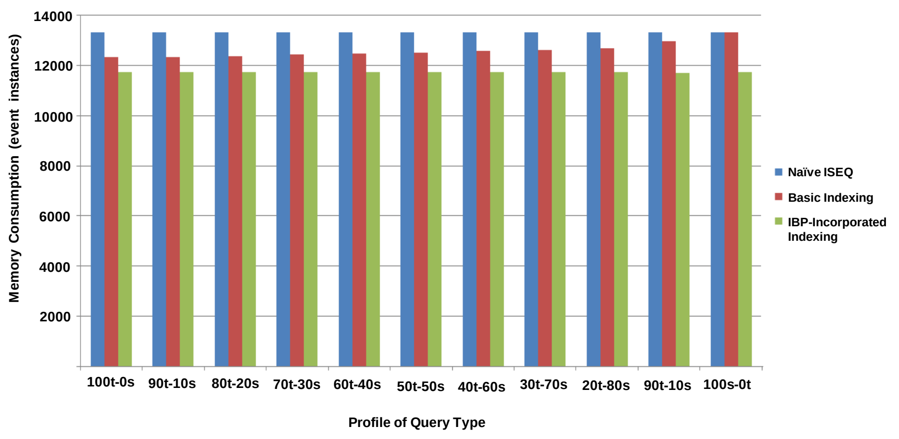


Figure 5.4: Results for Varying Query Types I

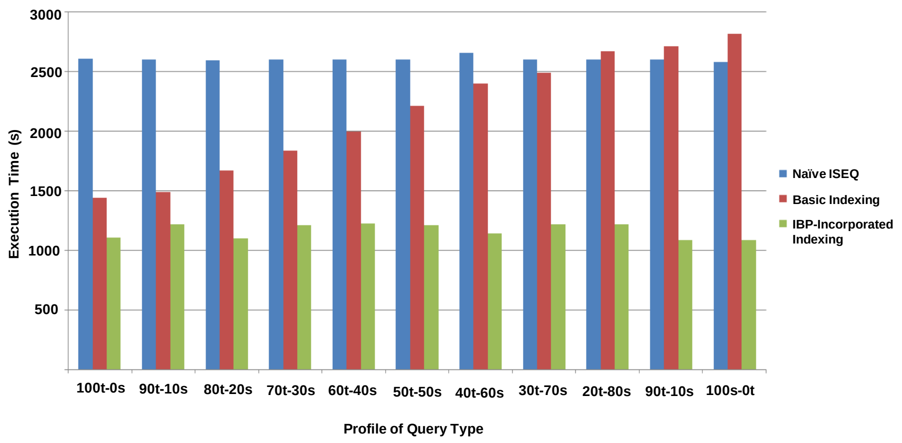


Figure 5.5: Results for Varying Query Types II

average interval length is fixed as W/10 with the event density as 200 events per window, which stays the same as *Chapter 5.5.3*. Experimental results are shown in *Figure 5.6* and *5.7* and the result analysis is given as follows.

Memory Consumption (*Figure 5.6*). X axis here represents the query length and Y axis shows the accumulative memory consumption for each query. We can see that the ratio of memory consumption saving (the slight saving on memory footprint discussed earlier in *Chapter 5.5.3*) stays relatively steady for the index-applied approaches while the query length increases, since the event intervals among different types are with uniform distribution.

CPU Performance (*Figure 5.7*). X axis still represents the query length and Y axis shows the execution time for each query. A query with a longer length requires much more CPU resources for the result construction than the naive ISEQ approach. Thus we can see that the ratio of CPU gain increases sharply for the index-applied approaches while the query length increases. Similar observation can be found in the comparison between the two index-applied approaches. The ratio of the IBP-incorporated approach's CPU gain over the basic indexing approach increases steadily while the query length increases, from 45% to 66%.

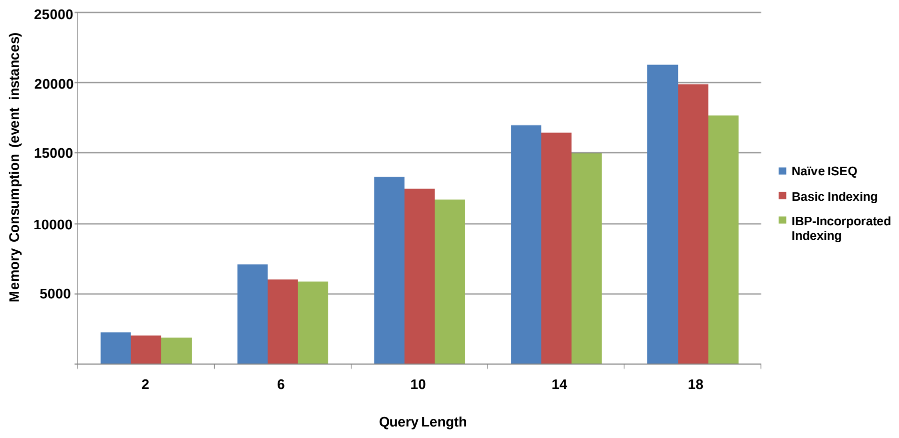


Figure 5.6: Results for Varying Query Length I

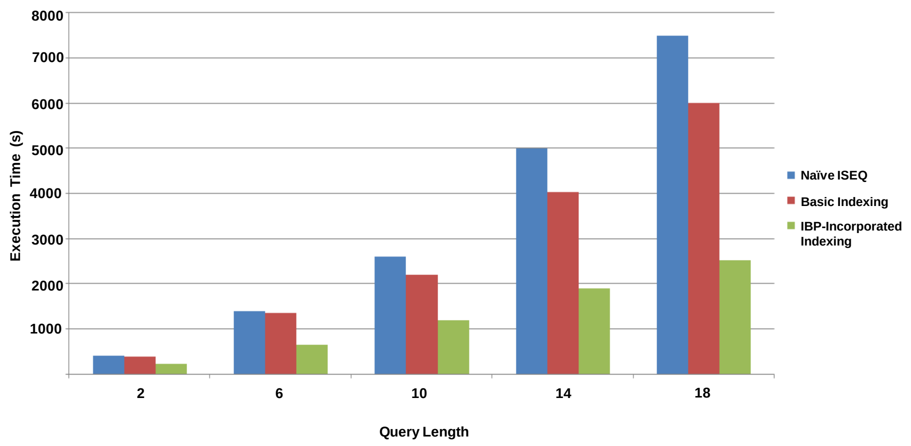


Figure 5.7: Results for Varying Query Length II

5.5.5 Experiments with Varying Average Interval Length

Since the input event stream is infinite, consistent performance over time can only be achieved by actively maintaining the data structures incrementally based on the given window constraint of the query [ADGI08]. Thus the event interval size and event density both affect the cost of buffer consumption and the result construction since they both affect the amounts of active instances kept in the operator state. We next study the effect of interval size by varying it from W/100 to W/5. Similar to the earlier settings, the *50t-50s* indexing profile is applied to all the queries in this set of experiments. The event density is set to 200 events per window and the query length is set to 10. Experimental results are shown in *Figure 5.8* and *5.9* and the result analysis is given as follows.

Memory Consumption (*Figure 5.8*). X axis here represents the interval length and Y axis shows the accumulative memory consumption for each query. We can see that with larger intervals (thus relatively smaller sliding window size in terms of holding how many complete event intervals), more memory footprint can be avoided for the IBP-incorporated approach. The ratio of the memory consumption gain scales with the average interval length. This is because that more intervals can be discarded directly through the on-the-fly dropping and more cascading AIS purge can be applied while intervals become easier to fall out of the sliding window. Similar observation can be found while comparing the basic indexing approach and the naive ISEQ approach.

CPU Performance (*Figure 5.9*). X axis still represents the interval length and Y axis shows the execution time for each query. Similar to the observation on the memory consumption, we can see that with larger intervals, more CPU cost can be avoided for both index-applied approaches, with a gain ratio in proportion to the interval length.

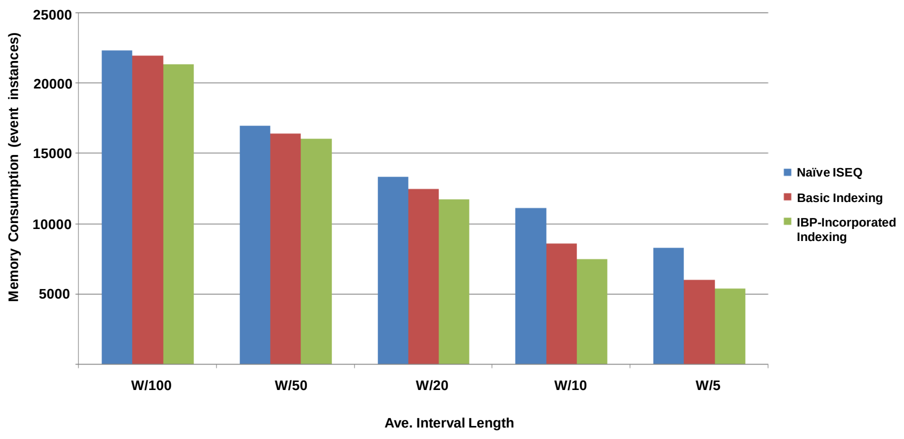


Figure 5.8: Results for Varying Average Interval Length I

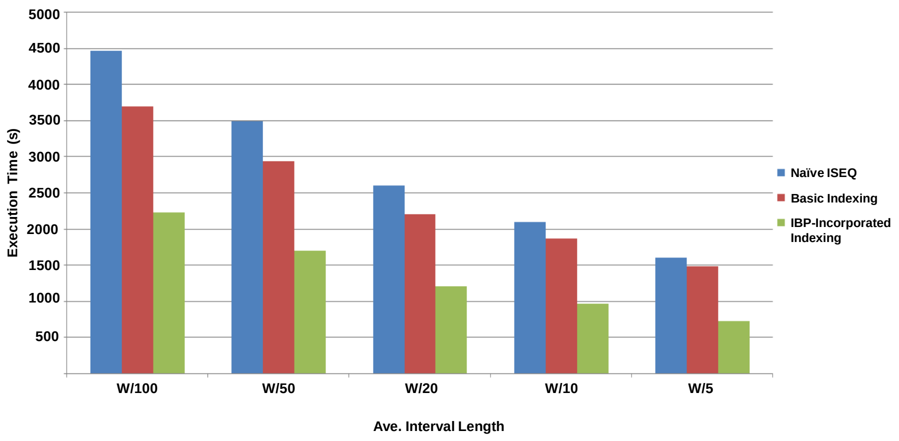


Figure 5.9: Results for Varying Average Interval Length II

5.5.6 Experiments with Varying Event Density

As the discussion in *Chapter 5.5.5*, the event interval size and event density both affect the cost of buffer consumption and the result construction. In this set of experiments we study the effect of event density by varying it from 50 events per window to 800 events per window. Note that for intervals we consider the event center (the middle point of the interval) as its representation. Similar to the earlier settings, the *50t-50s* indexing profile is applied to all the queries in this set of experiments. The average interval length is given as W/20 and the query length is set to 10. Experimental results are shown in *Figure 5.10* and *5.11* and the result analysis is given as follows.

Memory Consumption (*Figure 5.10***).** X axis here represents the interval length and Y axis shows the accumulative memory consumption for each query. We can see that with more sparse input (thus relatively smaller sliding window size in terms of covering how many event interval centers), more memory footprint can be avoided for the IBP-incorporated approach. The ratio of the memory consumption gain is in inverse proportion to the event density. This is because that with a more sparser input data set, less data will be hold by the operator since the state purge. Similar observation can be found while comparing the basic indexing approach and the naive ISEQ approach.

CPU Performance (*Figure 5.11*). X axis still represents the interval length and Y axis shows the execution time for each query. Similar to the observation on the memory consumption, we can see that with more sparse input, more CPU cost can be avoided for both index-applied approaches. However, the ratio is no longer just in inverse proportion to the event density when the input becomes very dense. We can see that the CPU cost increases sharply for the naive ISEQ approach comparing with the index-applied approaches while the query density jumps from 200 to 400 and from 400 to 800. Similar observation can be found for the comparison between the two index-applied approaches. This is because that with larger operator state, the result construction for the patterns without indexing becomes more and more inefficient.

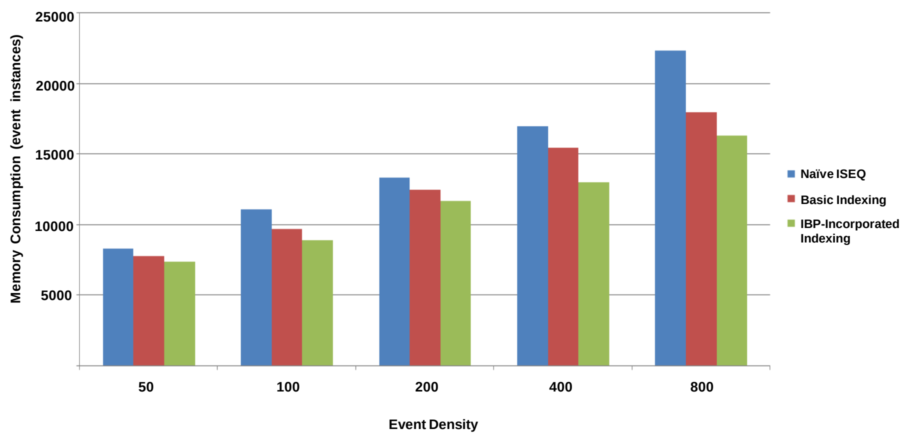


Figure 5.10: Results for Varying Event Density I

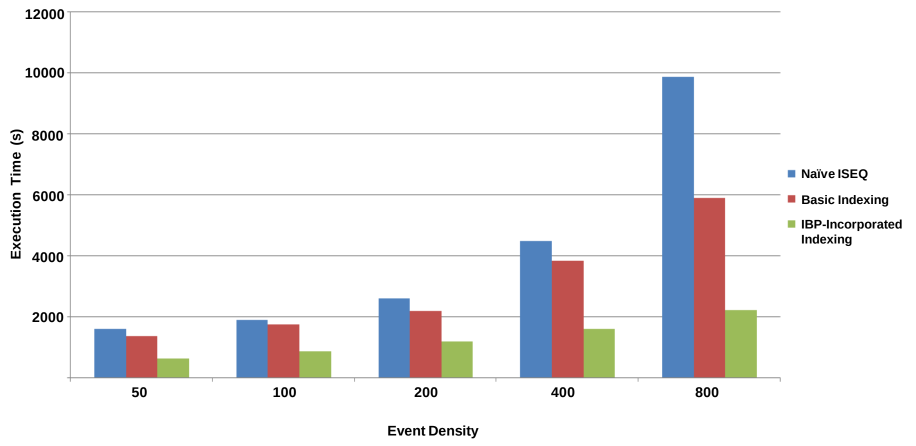


Figure 5.11: Results for Varying Event Density II

5.5.7 Conclusions of the Experimental Study

Above experimental results reveal that the proposed interval event stream processing framework is practical in three senses: *(1)* interval streams are handled correctly by the proposed framework; *(2)* the index-applied approaches outperform the naive ISEQ approach in most cases and *(3)* the IBP-incorporated outperforms the basic indexing approach.

5.6 Related Work

In [WDR06], the authors propose an expressive yet easy-to-understand language to support pattern queries on such sequential streams and propose customized algebra operators for the efficient processing of such pattern queries with sliding windows. [ACT08] uses a plan-based technique to perform streaming complex event detection across distributed sources. These researches on event pattern detection over event streams mainly focused on extracting temporal patterns from point-based event data [WDR06]. Even though in [ACT08][DCR$^+$08][DGP$^+$07] the events are defined based on the interval model. However, only the "*before*" / "*after*" temporal relation is supported, which simplifies the interval-based temporal model to the point-based temporal model by overlooking the patterns where events can have some overlapped portion.

The data mining community studied discovering patterns over interval events [KF00][PHL08][WC07]. [KF00] uses a hierarchical representation that extends Allen's interval algebra [All83] for modeling complex event patterns over intervals. However, this representation is lossy as the exact relationships among the events cannot be fully recovered. [WC07][PHL08] devise a lossless representation to overcome the drawbacks of [KF00]. Based on their proposed representation, they propose corresponding mining algorithms for pattern discovering over event intervals. [WC07] proposes the TPrefixSpan algorithm to mine the new temporal patterns from interval-based events. The completeness and accuracy of the results are also proven. Their experimental results show that the efficiency and scalability of the TPrefixSpan algorithm are satisfactory. An efficient algorithm called IEMiner is designed by [PHL08] to discover frequent temporal patterns from interval-based events. The algorithm employs two optimization techniques to reduce the search space and remove unpromising candidates. [PHL08] also examines how the discovered temporal patterns can be utilized in classification to differentiate closely related classes thus building an interval-based classifier called IEClassifier. Even though we adapted the idea of lossless representation of event patterns in these works, we cannot adapt their algorithms because they mainly focus on pattern discovering algorithms instead of pattern detection algorithms. Besides that, they do not consider streaming input with window constraints.

5.7 Conclusions

ESP is emerging as a key capability for many monitoring applications such as intrusion detection, sensor-based activity tracking and network monitoring. Existing ESP engines have focused on detecting temporal patterns from instantaneous events, that is, events with no duration. However, such sequential patterns are inadequate to express the complex temporal relationships in domains such as medical, multimedia, meteorology and finance where the durations of events could play an important role. Due to the differences between the temporal patterns over interval events and point events, the query semantics and evaluation mechanisms used for pattern detection over point events is not sufficient for pattern detection over interval events. An expressive language to represent the required temporal patterns among streaming interval events and corresponding evaluation mechanism for such event temporal queries is needed. In this dissertation task, I provide a framework to support interval event stream processing: *(1)* I introduce an expressive language to represent the required temporal patterns among streaming interval events; *(2)* I design the corresponding temporal operator ISEQ and provide an efficient evaluation strategy for the proposed ISEQ operator; *(3)* For further improving the event processing performance, I provide a mechanism to embed the "interval begin punctuation"(indicating the start of an upstream interval) into the interval stream and based on that I also discuss an approach to convert the interval-based event query into a simple point-based event query thus providing a possible adaptation for the point-based ESP systems; *(4)* I study a method to push down the computation of interval event abstraction to the low level sensor network for increasing the computing leverage from the physical level devices; *(5)* I conduct experimental studies to demonstrate the effectiveness of my proposed approach on query processing over interval event streams.

Chapter 6

Solution Integration and Dissertation Conclusions

6.1 Solution Integration

The research on event-specific stream processing technology is still at an early stage. Some issues on system robustness have not yet been considered in the current research work on ESP. First, data stream applications are required to handle very large volume of real-time inputs and provide fast real-time system response continuously, thus a lightweight runtime processing and minimized memory footprint play an important role in the robustness of event stream processing. Second, event streams are generated by different sources in different formats and they are sent through the ESP systems by different mechanisms in practice – thus a robust ESP engine needs to provide real-time support for complex event query over event streams with flexible input semantics. As discussed in *Chapter 6.1*, these research challenges regarding the robustness of the ESP systems are categorized into the following three: *(1)* lack of mechanism in lightweight constraint-aware query processing; *(2)* lack of mechanism in handling event streams with out-of-order data arrival; *(3)* lack of mechanism in handling event streams with interval-based temporal semantics.

This dissertation focuses on providing a robust ESP solution by meeting the above three research challenges. As mentioned in the earlier chapters (*Chapter 3*, *4* and *5*), the proposed techniques for the research challenges are have the limitation as not incorporating with each other. For example, in the proposed constraint-aware complex event pattern detection framework, assumptions of events being point-based and in-order are made. Similarly, in the proposed out-of-order processing framework, I assume only point-based events. For supporting the cases of more complex scenarios, these proposed techniques are required to be integrated together. The following I study the integration approach for the proposed techniques. It is categorized into four cases as follows and part of the information is shown in *Figure 6.1*:

Event Processing for Out-of-Order Interval Streams. Solution for out-of-order (referred to as *OOO*) handling introduced in *Chapter 4* applies to the design of an OOO-enabled ISEQ operator, which requires design revises on the basic ISEQ in terms of event buffering, result construction and operator state purge. Such handling revises are similar to the revises made to the basic SEQ for building the OOO-enabled SEQ. However, while with an approach where intervals are converted into points, the proposed out-of-order handling technique could be applied directly.

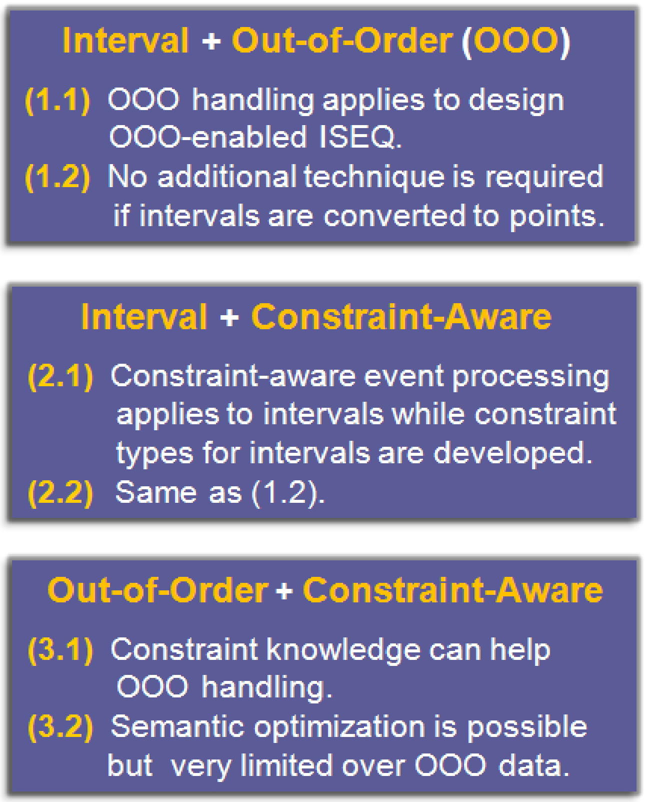
Constraint-Aware Event Processing for Interval Streams. The proposed constraint-aware event pattern detection framework in *Chapter 3* can work with interval events correctly while the query is limited to the temporal semantics applied in the constraint. Thus, if the constraint language is extended to support the interval relations, the proposed constraint-aware framework could be easily adapted to handle interval events. Similar as above, while with an approach where intervals are converted into points, the proposed out-of-order handling technique could be applied directly.

Constraint-Aware Event Processing for Out-of-Order Point-Based Streams. The constraint-aware knowledge could help an out-of-order data-incorporated event engine to determine whether possible out-of-order input on a certain event pattern could be seen from the downstream. On the other hand, since the out-of-order property of the input stream, the information given in the constraint (such as a workflow) could majorly become invalid. Thus the semantic-based optimization discussed in *Chapter 3* can only be used in very limited cases where constraints can still be implied from the out-of-order data input.

Constraint-Aware Event Proc. for Out-of-Order Interval Streams. This case requires applying all integration methods in the categories above.

6.2 Dissertation Conclusions

In this dissertation, I focus on providing a robust ESP solution by meeting the three research challenges concluded in *Chapter 6.1*. The dissertation



Interval + Out-of-Order (OOO)

(1.1) OOO handling applies to design OOO-enabled ISEQ.

(1.2) No additional technique is required if intervals are converted to points.

Interval + Constraint-Aware

(2.1) Constraint-aware event processing applies to intervals while constraint types for intervals are developed.

(2.2) Same as (1.2).

Out-of-Order + Constraint-Aware

(3.1) Constraint knowledge can help OOO handling.

(3.2) Semantic optimization is possible but very limited over OOO data.

Figure 6.1: Integration of the Proposed Techniques

research has lead to several publications (shown in *Table 6.1*) thus far with additional manuscripts currently in preparation at international conferences / workshops / journals that are the premier venue in the study of event processing and semantic computing (*ICDE*, *SIGMOD*, *DASFAA*, *ICDCS Workshops*, *ICSC* and *DEBS*), including 4 full research papers (one still in preparation for submission), 2 short research papers, 2 demonstration

Dissertation Task	**Publication**
(I). Constraint-Aware Complex Event Pattern Detection over Streams	[LMRL10][LMRL09b] [LMRL09a][LMRLonb]
(II). Complex Event Pattern Detection over Streams w/ Out-of-Order Data Arrival	[LLD+07][LLG+09] [WLL+09]
(III). Complex Event Pattern Detection over Streams w/ Interval-Based Temporal Semantics	[LMR+09][LMRLona]

Table 6.1: Publications Lead by the Dissertation Research

proposals and one journal paper (still in preparation for submission).

The dissertation research is completed by finishing three dissertation tasks, which are concluded as below:

Task I - Constraint-Aware Complex Event Pattern Detection over Streams. ESP has become increasingly important for modern enterprises to react quickly to critical business situations. In many practical cases, constraints (such as business workflows) often hold among business events. For query processing over event streams, reasoning using such known constraints enables us to *(1)* notify the unsatisfiability for a query at the earliest, thereby helping us to terminate the long running pattern detection processes that are guaranteed to not lead to successful matches; *(2)* identify the satisfiability for a query at the earliest possible moment, thereby helping us to get prepared for upcoming situations at the earliest. How to completely and efficiently exploit the given semantic knowledge on the input event streams to detect event patterns over large volumes of business transaction streams is still an open question. In this task, I propose a framework for constraint-aware pattern detection over event streams. Given the constraint of the input streams, the proposed framework on the fly checks the query satisfiability / unsatisfiability using a lightweight reasoning mechanism. Based on the constraint specified in the input stream, we are able to adjust the processing strategy dynamically, by producing early feedbacks, releasing unnecessary system resources and terminating corresponding pattern monitor, thus effectively decreasing the resource consumption and expediting the system response on certain situations. I have implemented the proposed constraint-aware pattern detection mechanism in a prototype system called E-Tec (constraint-aware query Engine for pattern deTection over event streams). Experimental studies are conducted to illustrate the significant performance improvement achieved by applying the proposed framework with little overhead.

Task II - Complex Event Pattern Detection over Streams with Out-of-Order Data Arrival. A key aspect of event processing is to extract patterns from event streams to make informed decisions in real-time. However, network latencies and machine failures may cause events to arrive out-of-order at the ESP system. State-of-the-art event stream processing technology experiences significant challenges when faced with out-of-order data arrival including output blocking, huge system latencies, memory resource overflow, and incorrect result generation. In this task, I propose a mechanism to address the problem of processing event queries specified over streams that may contain out-of-order data. I first analyze the problems that the state-of-the-art event stream processing technologies would experience when faced with out-of-order data arrival and study the levels of correctness in out-of-order processing that target priorities of applications considering latency, output order, result correctness and result completeness. I then provide a new solution of physical implementation strategies for the core stream algebra operators such as sequence scan, pattern construction and negation, including stack-based data structures and associated purge algorithms. Optimization for sequence scan and construction as well as state purging to minimize CPU cost and memory consumption are also introduced. Experimental studies are conducted to demonstrate the effectiveness of the proposed approach on query processing over event streams with out-of-order data arrival.

Task III - Complex Event Pattern Detection over Streams with Interval-Based Temporal Semantics. Existing ESP engines have focused on detecting temporal patterns from instantaneous events, that is, events with no duration. However, such sequential patterns are inadequate to express the complex temporal relationships in domains such as medical, multimedia, meteorology and finance where the durations of events could play an important role. Due to the differences between the temporal patterns over interval events and point events, the query semantics and evaluation mechanisms used for pattern detection over point events is not sufficient for pattern detection over interval events. An expressive language to represent the required temporal patterns among streaming interval events and corresponding evaluation mechanism for such event temporal queries is needed. In this dissertation task, I introduce an expressive language to represent the required temporal patterns among streaming interval events. I design the corresponding temporal operator ISEQ and provide an efficient evaluation strategy for the proposed ISEQ operator. For further improving the event processing performance, I provide a mechanism to embed the "interval begin punctuation"(indicating the start of an upstream interval) into the interval stream. Corresponding punctuation-aware query evaluation strategy is investigated, which can greatly reduce the runtime memory and CPU footprint. I then study a mechanism to push down the computation of interval event abstraction to the low level sensor network for increasing the computing leverage from the physical level devices. Experimental studies have been conducted to demonstrate the effectiveness of the proposed approach.

Bibliography

[ACC$^+$03] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, August 2003.

[ACT08] Mert Akdere, Ugur Cetintemel, and Nesime Tatbul. Plan-based complex event detection across distributed sources. *PVLDB*, 1(1):66–77, 2008.

[ADGI08] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2008.

[AE04] Asaf Adi and Opher Etzion. Amit - the situation manager. *VLDB J.*, 13(2):177–203, 2004.

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[Bas07] Tim Bass. Mythbusters: event stream processing versus complex event processing. In *DEBS*, page 1, 2007.

[BBMW02] B. Babcock, S. Babu, R. Motwani, and J. Widom. Models and issues in data streams. In *PODS*, pages 1–16, June 2002.

[BCCN06] Véronique Benzaken, Giuseppe Castagna, Dario Colazzo, and Kim Nguyen. Type-based xml projection. In *VLDB*, pages 271–282, 2006.

[BE04] David Botzer and Opher Etzion. Self-tuning of the relationships among rules' components in active databases systems. *IEEE Trans. Knowl. Data Eng.*, 16(3):375–379, 2004.

[BGAH07] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, pages 363–374, 2007.

[BGHJ09] Lars Brenna, Johannes Gehrke, Mingsheng Hong, and Dag Johansen. Distributed event stream processing with non-deterministic finite automata. In *DEBS*, 2009.

[BMM$^+$04] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, pages 407–418, 2004.

[BW01] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.

[CA08] Sharma Chakravarthy and Raman Adaikkalavan. Events and streams: harnessing and unleashing their synergy! In *DEBS*, pages 1–12, 2008.

[CCD$^+$03] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, pages 269–280, 2003.

[CJC06] Shyh-Kwei Chen, Jun-Jang Jeng, and Henry Chang. Complex event processing using simple rule-based event correlation engines for business performance management. In *CEC/EEE*, page 3, 2006.

[CKAK94] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB*, pages 606–617, 1994.

[Cro04] Christopher Crowell. Event correlation and root cause analysis. *White Paper of CA Inc.*, 2004.

[DCR$^+$08] Luping Ding, Songting Chen, Elke A. Rundensteiner, Jun'ichi Tatemura, Wang-Pin Hsiung, and K. Selçuk Candan. Runtime semantic query optimization for event stream processing. In *ICDE*, pages 676–685, 2008.

[Dem09] Hiroshi Dempo. Qos evaluations of distributed event orchestration system. In *DEBS*, 2009.

[DGP$^+$07] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[DMRH04] Luping Ding, Nishant Mehta, Elke A. Rundensteiner, and George T. Heineman. Joining punctuated streams. In *EDBT*, pages 587–604, 2004.

[Etz07] Opher Etzion. Semantic approach to event processing. In *DEBS*, page 139, 2007.

[FNS$^+$99] Peter Fröhlich, Wolfgang Nejdl, Michael Schroeder, Carlos Viegas Damásio, and Luís Moniz Pereira. Using extended logic programming for alarm-correlation in cellular phone networks. In *IEA/AIE*, pages 343–352, 1999.

[GADI08] Daniel Gyllstrom, Jagrati Agrawal, Yanlei Diao, and Neil Immerman. On supporting kleene closure over event streams. In *ICDE*, pages 1391–1393, 2008.

[gar] Gartner Inc. *http://www.gartner.com*.

[GC$^+$07] Daniel Gyllstrom, Eugene Wu 0002, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. Sase: Complex event processing over streams (demo). In *CIDR*, pages 407–411, 2007.

[GD94] Stella Gatziu and Klaus R. Dittrich. Detecting composite events in active database systems using petri nets. In *RIDE-ADS*, pages 2–9, 1994.

[GJS92] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. Composite event specification in active databases: Model & implementation. In *VLDB*, pages 327–338, 1992.

[HBR$^+$05] Jeong-Hyon Hwang, Magdalena Balazinska, Alex Rasin, Ugur Çetintemel, Michael Stonebraker, and Stanley B. Zdonik. High-availability algorithms for distributed stream processing. In *ICDE*, pages 779–790, 2005.

[HG00] C. Harris and S. Gass. Encyclopedia of MS/OR. 2000.

[KF00] Poshan Kam and Ada W. Fu. Discovering temporal patterns for interval-based events. In *DaWaK*, pages 317–326, 2000.

[KF09] Ingmar Kellner and Ludger Fiege. Viewpoints in complex event processing: industrial experience report. In *DEBS*, 2009.

[KJP09] Alexander Kozlenkov, David Jeffery, and Adrian Paschke. State management and concurrency in event processing. In *DEBS*, 2009.

[KNV03] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, March 2003.

[Koz03] Dexter Kozen. Automata and computability. In *W.H.Freeman and Company, New York*, 2003.

[KSSS04] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *VLDB*, pages 228–239, 2004.

[LLD$^+$07] Ming Li, Mo Liu, Luping Ding, Elke A. Rundensteiner, and Murali Mani. Event stream processing with out-of-order data arrival. In *ICDCS Workshops*, 2007.

[LLG$^+$09] Mo Liu, Ming Li, Denis Golovnya, Elke A. Rundensteiner, and Kajal T. Claypool. Sequence pattern query processing over out-of-order event streams. In *ICDE*, pages 784–795, 2009.

[LMR08a] Ming Li, Murali Mani, and Elke A. Rundensteiner. Constraint-aware xslt evaluation. In *ER*, pages 524–525, 2008.

[LMR08b] Ming Li, Murali Mani, and Elke A. Rundensteiner. Efficiently loading and processing xml streams. In *IDEAS*, pages 59–67, 2008.

[LMR08c] Ming Li, Murali Mani, and Elke A. Rundensteiner. Elf: A constraint-aware xquery engine for processing xml streams with minimized memory footprint. In *ICSC*, pages 494–495, 2008.

[LMR08d] Ming Li, Murali Mani, and Elke A. Rundensteiner. Semantic query optimization for processing xml streams with minimized memory footprint. In *EDBT Workshops*, pages 27–36, 2008.

[LMR$^+$09] Ming Li, Murali Mani, Elke A. Rundensteiner, Di Wang, and Tao Lin. Interval event stream processing. In *DEBS*, 2009.

[LMRL09a] Ming Li, Murali Mani, Elke A. Rundensteiner, and Tao Lin. Constraint-aware event stream processing. In *DEBS*, 2009.

[LMRL09b] Ming Li, Murali Mani, Elke A. Rundensteiner, and Tao Lin. E-tec: A constraint-aware query engine for pattern detection over event streams. In *ICSC*, pages 565–566, 2009.

[LMRL10] Ming Li, Murali Mani, Elke A. Rundensteiner, and Tao Lin. Constraint-aware complex event pattern detection over streams. In *DASFAA*, pages 199–215, 2010.

[LMRLona] Ming Li, Murali Mani, Elke A. Rundensteiner, and Tao Lin. Complex event pattern detection over streams with interval-based temporal semantics. In preparation for conference submission.

[LMRLonb] Ming Li, Murali Mani, Elke A. Rundensteiner, and Tao Lin. Constraint-aware complex event pattern detection over streams. In preparation for journal submission.

[LMT$^+$05] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, pages 311–322, 2005.

[LRE09] Geetika T. Lakshmanan, Yuri G. Rabinovich, and Opher Etzion. A stratified approach for supporting high throughput event processing applications. In *DEBS*, 2009.

[LTS$^+$08] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: a new architecture for high-performance stream systems. *PVLDB*, 1(1):274–288, 2008.

[Luc07] David Luckham. The power of events: An introduction to complex event processing in distributed enterprise systems. *ACM Trans. Database Syst.*, 2007.

[LZR06] Bin Liu, Yali Zhu, and Elke A. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *SIGMOD Conference*, pages 347–358, 2006.

[MM09] Yuan Mei and Samuel Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD Conference*, pages 193–206, 2009.

[MRLD08] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Advanced event processing and notifications in service runtime environments. In *DEBS*, pages 115–125, 2008.

[MS07] Carolyn McGregor and Michael Stacey. High frequency distributed data stream event correlation to improve neonatal clinical management. In *DEBS*, pages 146–151, 2007.

[NB94] Bernhard Nebel and Hans-Jurgen Burckert. Reasoning about temporal relations: A maximal tractable subclass of allen's interval algebra. In *AAAI*, pages 356–361, 1994.

[NB95] Bernhard Nebel and Hans-Jurgen Burckert. Reasoning about temporal relations: A maximal tractable subclass of allen's interval algebra. *J. ACM*, 42(1):43–66, 1995.

[NRJ04] Steven Noel, Eric Robertson, and Sushil Jajodia. Correlating intrusion events and building attack scenarios through attack graph distances. In *ACSAC*, pages 350–359, 2004.

[PD99] Norman W. Paton and Oscar Diaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, 1999.

[PHL08] Dhaval Patel, Wynne Hsu, and Mongli Lee. Mining relationships among interval-based events for classification. In *SIGMOD*, pages 393–404, 2008.

[PV09] Adrian Paschke and Paul Vincent. A reference architecture for event processing. In *DEBS*, 2009.

[QL04] Xinzhou Qin and Wenke Lee. Attack plan recognition and prediction using causal networks. In *ACSAC*, pages 370–379, 2004.

[RB06] Grigore Rosu and Saddek Bensalem. Allen linear (interval) temporal logic - translation to ltl and monitor synthesis. In *CAV*, pages 263–277, 2006.

[RDS$^+$04] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, pages 1353–1356, 2004.

[SC$^+$09] Thanh Tran 0002, Charles Sutton, Richard Cocci, Yanming Nie, Yanlei Diao, and Prashant J. Shenoy. Probabilistic inference over rfid streams in mobile environments. In *ICDE*, pages 1096–1107, 2009.

[SM04] Josef Schiefer and Carolyn McGregor. Correlating events for monitoring business processes. In *ICEIS (1)*, pages 320–327, 2004.

[SMMP09] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter R. Pietzuch. Distributed complex event processing with query rewriting. In *DEBS*, 2009.

[SPL96] Praveen Seshadri, Hamid Pirahesh, and T. Y. Cliff Leung. Complex query decorrelation. In *ICDE*, pages 450–458, 1996.

[SRM05] Hong Su, Elke A. Rundensteiner, and Murali Mani. Semantic Query Optimization for XQuery over XML Streams. In *VLDB*, pages 1293–1296, 2005.

[SW02] A. Schmidt and F. Wass. XMark: a benchmark for XML data management. In *VLDB*, pages 974–985, 2002.

[Tom96] David Toman. Point vs. interval-based query languages for temporal databases. In *PODS*, pages 58–67, 1996.

[vAEE$^+$09] Rainer von Ammon, Christoph Emmersberger, Thomas Ertlmaier, Opher Etzion, Thomas Paulus, and Florian Springer. Existing and future standards for event-driven business process management. In *DEBS*, 2009.

[WAR08] Seth White, Alexandre Alves, and David Rorke. Weblogic event server: a lightweight, modular application server for event processing. In *DEBS*, pages 193–200, 2008.

[WBG08] Karen Walzer, Tino Breddin, and Matthias Groch. Relative temporal constraints in the rete algorithm for complex event detection. In *DEBS*, pages 147–155, 2008.

[WC07] Shinyi Wu and Yenliang Chen. Mining nonambiguous temporal patterns for interval-based events. *IEEE Trans. Knowl. Data Eng.*, 19(6):742–758, 2007.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.

[WGB08] Karen Walzer, Matthias Groch, and Tino Breddin. Time to the rescue - supporting temporal reasoning in the rete algorithm for complex event processing. In *DEXA*, pages 635–642, 2008.

[WLL$^+$09] Mingzhu Wei, Mo Liu, Ming Li, Denis Golovnya, Elke A. Rundensteiner, and Kajal T. Claypool. Supporting a spectrum of out-of-order event processing technologies: from aggressive to conservative methodologies. In *SIGMOD*, pages 1031–1034, 2009.

[WLRM06] Mingzhu Wei, Ming Li, Elke A. Rundensteiner, and Murali Mani. Processing recursive xquery over xml streams: The raindrop approach. In *ICDE Workshops*, page 85, 2006.

[WR09] Song Wang and Elke A. Rundensteiner. Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing. In *EDBT*, pages 299–310, 2009.

[WRGB06] Song Wang, Elke A. Rundensteiner, Samrat Ganguly, and Sudeept Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *VLDB*, pages 619–630, 2006.

[WRML08] Mingzhu Wei, Elke A. Rundensteiner, Murali Mani, and Ming Li. Processing recursive xquery over xml streams: The raindrop approach. *Data Knowl. Eng.*, 65(2):243–265, 2008.

[WSL$^+$06] S. Wang, H. Su, M. Li, M. Wei, S. Yang, E. A. Rundensteiner, and M. Mani. R-sox: Runtime semantic query optimization over xml streams. In *VLDB*, pages 1207–1210, 2006.

[XN04] Dingbang Xu and Peng Ning. Alert correlation through triggering events and common resources. In *ACSAC*, pages 360–369, 2004.

[ZRH04] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD Conference*, pages 431–442, 2004.