Extending Event Sequence Processing: New Models and Optimization Techniques

by

Mo Liu

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 6, 2012

APPROVED:

Prof. Elke A. Rundensteiner
Worcester Polytechnic Institute
Advisor

Prof. Daniel J. Dougherty
Worcester Polytechnic Institute
Committee Member

Prof. Murali Mani
University of Michigan, Flint
External Committee Member

Prof. Yanlei Diao
University of Massachusetts, Amherst
External Committee Member

Prof. Ismail Ari
Ozyegin University
External Committee Member

Prof. Craig E. Wills
Worcester Polytechnic Institute
Head of Department

To my parents.

Abstract

Complex event processing (CEP) has become increasingly important for tracking and monitoring applications ranging from health care, supply chain management to surveillance. Most of state-of-the art CEP systems assume events arrive in order. However, imperfections in events delivery are common due to the variance in the network latencies. Out-of-order event processing strategies must be designed to achieve robust query processing. Monitoring applications submit a workload of complex event queries to track sequences of events over different abstraction levels. As these systems mature the need for increasingly complex queries supporting nesting of sequence (SEQ), AND, OR and negation arises, while the state-of-the-art CEP systems mostly support single flat sequence queries. New CEP models supporting nested and multi-dimensional queries with associated efficient processing techniques are essential to assure real-time responsiveness and scalability.

First, to lay the foundation of out-of-order event processing, we address the problem of processing flat pattern queries on event streams with out-of-order data arrival. State-of-the-art event stream processing technology experiences significant challenges when faced with out-of-order data arrival including output blocking, huge latencies, memory resource overflow, and incorrect result generation. We

design two alternate solutions: aggressive and conservative strategies respectively to process sequence pattern queries on out-of-order event streams. The aggressive strategy produces maximal output under the optimistic assumption that out-of-order event arrival is rare. The conservative method works under the assumption that out-of-order data may be common, and thus produces output only when its correctness can be guaranteed. Our experimental study evaluates the robustness of each method, and compares the respective scope of applicability with state-of-art methods using workloads composed of flat sequence queries.

Second, to support queries over different abstraction levels, we propose a novel *E-Cube* model which combines CEP and OLAP techniques for efficient multi-dimensional flat sequence pattern analysis at different abstraction levels. Our analysis of the interrelationships in both concept abstraction and pattern refinement among queries facilitates the composition of these queries into an integrated *E-Cube* hierarchy. Based on this *E-Cube* hierarchy, strategies of drill-down (refinement from abstract to more specific patterns) and of roll-up (generalization from specific to more abstract patterns) are developed for the efficient workload evaluation. The proposed execution strategies reuse intermediate results along both the concept and the pattern refinement relationships between queries. Based on this foundation, we design a cost-driven adaptive optimizer called *Chase* that exploits the above reuse strategies for optimal *E-Cube* hierarchy execution. The experimental studies comparing alternate strategies on a real world financial data stream under different workload conditions demonstrate the superiority of the *Chase* method. In particular, our *Chase* execution in many cases performs ten fold faster than the state-of-art strategy for real stock market query workloads.

Last, we tackle nested CEP query processing. Without the design of an optimized execution strategy for nested sequence queries, an iterative nested execution strategy would typically be adopted by default. The rigid process of first undertaking the construction of sequence results for the outer operators and then iteratively for each outer result to construct sequence results for the inner operators is not efficient as it misses critical opportunities for optimization. Not only are substantial resources wasted on first constructing subsequences just to be subsequently discarded, but also opportunities for shared execution of nested subexpressions are overlooked. As foundation, to overcome this shortcoming, we introduce *NEEL*, a CEP query language for expressing nested CEP pattern queries composed of sequence, negation, AND and OR operators. To allow flexible execution order, we devise a normalization procedure that employs rewriting rules for flattening a nested complex event expression. To conserve CPU and memory consumption, we propose several strategies for efficient shared processing of groups of normalized *NEEL* subexpressions. These strategies include prefix caching, suffix clustering and customized "bit-marking" execution strategies. We design an optimizer to partition the set of all CEP subexpressions in a *NEEL* normal form into groups, each of which can then be mapped to one of our shared execution operators. Lastly, we evaluate our technologies by conducting a performance study to assess the CPU processing time using real-world stock trades data. Our results confirm that our *NEEL* execution in many cases performs 100 fold faster than the traditional iterative nested execution strategy for real stock market query workloads.

In summary, this dissertation innovates several techniques at the core of a scalable *E-Analytic* system to achieve efficient, scalable and robust methods for in-memory multi-dimensional nested pattern analysis over high-speed event streams.

Acknowledgments

The work presented in this thesis would not be what is today without the support and contribution of many people. Foremost, I would like to express my gratitude to my advisor, Prof. Elke A. Rundensteiner, for her great patience, understanding, continuous support and encouragement. The many fruitful discussions with her have shaped all aspects of this work.

My thanks also go to Prof. Dan Dougherty who spend lots of time guiding me during my PhD study and give lots of valuable suggestions to my dissertation. I also thank other members of my Ph.D. committee, Prof. Ismail Ari, Prof. Murali Mani and Prof. Yanlei Diao, who provided important feedback to my dissertation proposal, my comprehensive-exam and dissertation drafts. All these helped to improve the presentation and content of this dissertation.

The friendship of Mingzhu Wei, Di Wang, Di Yang, Venkatesh Raghavan, Abhishek Mukherji, Hao Loi, Xika Lin, Medhabi Ray, Lei Cao, Han Wang and all the other previous and current DSRG members is much appreciated. The life in Worcester becomes much more interesting with their company. I would like to thank Denis Golovnya, Kajal Claypool and Luping Ding for their collaboration on the ESPO project. I thank Medhabi Ray for her collaboration on the NestedCEP project. They have contributed to many interesting and good-spirited discussions related to this research.

I also thank Worcester Polytechnic Institute and the Computer Science Department for giving me the opportunity to study and also providing TAship during my studies. Especially, I thank HP Labs for their funding support via the HP Labs Innovation Research Program. I would like to thank Dr. Chetan Gupta, Dr. Song Wang, Dr. Umeshwar Dayal and Abhay Mehta for their collaboration on this research and all their encouragement and feedback.

Last, but not least, I would like to thank my husband Ming Li for his understanding and love during the past few years. My parents receive my deepest gratitude and love for their dedication and the many years of unconditional support and love.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

The recent advances in hardware and software have enabled the capture of different measurements of data in a wide range of fields. Applications that generate rapid, continuous and large volumes of event streams include readings from sensors, such as physics, biology and chemistry experiments, weather sensors [FJK$^+$05, Mou03, Uni02], health sensors [SB03], network sensors [Uni02], online auctions, credit card operations [Pet03], financial tickers [ZS02], web server log records [AK00], etc. Given these developments, the world is poised for a sea-change in terms of variety, scale and importance of applications enabled by the real-time analysis and exploitation of such event streams - from dynamic traffic management, environmental monitoring to health care alike. Clearly, the ability to infer relevant patterns from these event streams in real-time to make near instantaneous yet informed decisions is absolutely crucial for these mission critical applications. Next let us motivate this need using several concrete example applications.

- **Shoplifting.** Let us consider a popular application for tracking goods in a retail store [WDR06] where RFID tags are attached to each product and RFID readers are placed at strategic locations throughout the store, such as shelves, checkout counters and the store exit. The path of one product from the shelf to store exit can be tracked as it passes the different RFID readers, and the events generated from the RFID readers can be analyzed to detect theft. For example, if a shelf and a store exit readings for a product are read, but the RFID tag is not read at any of the checkout counters prior to the store exit, then a natural conclusion may be that the product is being shoplifted.

- **Health care.** Consider reporting unsafe medical equipments in a hospital. Let us assume that the tools for medical operations are RFID-tagged. The system monitors the histories (e.g., records of surgical usage, washing, sharpening, disinfection, etc.) of the tools. When a nurse puts a box of surgical tools into a surgical table equipped with RFID readers, the computer may display warnings such as “This tool must be disposed”. A query accomplishing this monitors is after being recycled and washed, a tool is being put back into use without being first sharpened, disinfected and then checked for quality assurance. Consider another example in preventing hospital-acquired infections for healthcare workers [FK08] [JD02]. The system continuously tracks healthcare workers and concurrently reminds the workers at the appropriate moments to perform hand hygiene. A surveillance system may want to monitor the hand hygiene violation caused by a healthcare worker who exited a room but did not clean his hands within 15 seconds.

- **Tag-based evacuation systems.** Consider an evacuation system where RFID technology would be used to track the mass movement of people and other objects during natural disasters. Tags are attached to people and other objects. Tags transmit position related information to a base station. Terabytes of RFID data could be generated by such a tracking system. Facing such a huge volume of RFID data, emergency personnel need to be able to perform pattern detection on various dimensions at different granularities in real-time. In particular, one may need to monitor people movement and traffic patterns of needed goods and resources (say, water and blankets) at different levels of abstraction such as types of goods and types of locations in order to ensure fast and optimized relief efforts. For example, federal government personnel may monitor movement of people from cities in Texas to Oklahoma for global resource placement; while local authorities may focus on people movement starting from the Dallas bus station, traveling through the Tulsa bus station, and ending in the Tulsa hospital within 48 hours (a time window) to determine the need for additional means of transportation.

The Problem of Complex Event Analysis. Common across the above scenarios is a need to process complex queries over huge volumes, and potentially unbounded, streaming data in real-time at various abstraction levels in a robust manner. Event data may arrive out-of-order at the event processing engine. Stream speeds can be extremely high on the order of megabytes per second or more [ZW07]. Furthermore, streaming event data tends to have many dimensions (time, location, objects), with each dimension possibly hierarchical in nature. In addition, the query requests can be nested in nature composed of negation, recursion, sequencing and other powerful operators to express the pattern of interest. To complicate matters even further, such systems are typically faced with a huge number of pattern requests, all specified to operate against the same high volume stream, while still requiring near real-time responsiveness. Detecting complex patterns in high-rate event streams requires substantial CPU resources. We target the efficient processing of complex pattern queries which are nested or at multiple levels of abstraction over extremely high-speed event streams. In short, these applications share the common need for a special-purpose event stream technology capable of robust processing of complex nested queries and analyzing vast amount of multi-dimensional data to enable multi-faceted online, operational decision making.

1.2 State-of-the-Art

The naive method for dealing with out-of-order arrival of events, called *K-slack* [Shi04], buffers the arriving data for *K* time units. However, as the average latencies change, *K* may become either too large, thereby buffering un-needed data and introducing unnecessary inefficiencies and delays for the processing, or too small, thereby becoming inadequate for handling the out-of-order processing of the arriving events and resulting in inaccurate results. To handle out-of-order data arrival, the authors in [LTS$^+$08] propose to apply explicit stream progress indicators, such as punctuation or heartbeats, to unblock and purge operators. The authors focus on out of order handling for operators such as aggregation and join. However, the authors don't consider out-of-order handling for the sequence operator SEQ with negation over event streams. Recently, the authors from MSR [CGM10] apply punctuation and revision processing over disordered streams for dynamic patterns, where the pattern (query) itself can change over time.

Existing techniques such as traditional online analytical processing (OLAP) systems are not designed for real-time pattern-based operations [CD97, HRU96, GHQ95], while state-of-the-art Complex Event Processing (CEP) systems designed for pattern matching tend to be limited in their expressive capability. More importantly they do not support OLAP operations [CKAK94, WDR06, BGAH07]. State-of-the-art OLAP technology is set-based (i.e., unordered) aggregates over scalar values [GHQ95]. Hence, in the context of event streams where the order of events is important, OLAP is insufficient in supporting efficient event sequence analysis. Thus in the dissertation, we set out to design a novel *event analytics model* that effectively leverages CEP and OLAP techniques for efficient multi-dimensional event pattern analysis at different abstraction levels. Given a workload of CEP pattern queries, our *event analytics technology* would exploit interrelationships between CEP pattern queries in terms of both concept and pattern refinement among these queries for optimized shared processing and maximal reuse of intermediate results – thus saving critical computational and memory resources.

One of the most flexible features of a query language is the nesting of operators [Kim82, MHM04]. Without this capability, users are severely restricted in forming complex patterns in a convenient and succinct manner. Conceptually, the state-of-art CEP systems such as SASE [WDR06], ZStream [MM09] and Cayuga system [BDG$^+$07] support nested queries as negation could be viewed as a special case of one-level deep nesting. However, because these systems utilize two step execution method, namely, the results satisfying the non-negation part are first constructed and then filtered if event instances which match the negation part exist, such forced execution ordering can miss optimization opportunities.

SASE+ [ADGI08] is a declarative language for specifying complex event patterns over streams. The semantics of the language is rich, spanning three dimensions in the Kleene closure definition as well as involving negation and composition. SASE+ queries can be composed by feeding the output of one query as input to another. However, the output of the first query is restricted to the atomic simple type. SASE+ does nested query processing and SASE+ doesn't support negation over composite event type. K*SQL [MZZ10] can express complex patterns on relational streams and sequences and can query data with complex structures, e.g, XML and genomic data. However, they don't support applying negation over composite event types. While CEDR [BGAH07] allows applying negation over composite event types within their proposed language, the execution strategy for such nested queries is not discussed. A declarative query language LINQ [PR08] used in Microsoft StreamInsight [Ae09] allows nested queries by composing query templates. However, no optimization is introduced for processing negation over composite event types.

1.3 Research Challenges

What is common across the motivating scenarios in Section 1.1 is a need to process complex queries over huge volumes, and potentially unbounded, streaming data in real-time at various abstraction levels in a robust manner. As analyzed in Section 1.2, we observe Complex Event Processing (CEP) faces several critical challenges:

Imperfections in Event Delivery. Events may arrive out-of-order to an CEP engine. To handle imperfections in event delivery and define consistency guarantees on the output is of great importance in robust query processing. When processing sequence pattern queries, state-of-the-art event stream processing technology [WDR06] experiences significant challenges with out-of-order data arrival including output blocking, huge system latencies, memory resource overflow, and incorrect result generation. We need to devise techniques to solve these problems. One commonly applied method is *K-slack* [Shi04]. It buffers the arriving data for K time units which would incur large latency. Recently, the authors [LTS$^+$08] propose to apply explicit stream progress indicators, such as punctuation or heartbeats, to unblock and purge operators. However, the authors don't consider out-of-order handling for event streams and, in particular, not for order-sensitive operators such as CEP sequences and negation.

Theory. One of the most interesting and flexible features of a query language is the composition of operators to an arbitrary depth [Kim82, MHM04]. Without this capability, users are severely restricted in forming complex patterns in a convenient and succinct manner. However, no clean syntax and semantics for nested CEP queries is designed. Most of the existing CEP systems [WDR06, MM09] only support flat pattern queries. Lacking a precise formal specification limits the opportunities for query optimization and query rewrites.

Querying Multi-Dimensional Data. There are numerous emerging applications, such as online financial transactions, IT operations management, and sensor networks that generate real-time streaming data. This streaming data has many dimensions (time, location, objects) and each dimension can be hierarchical in nature. One important common problem over such data is to be able to analyze multiple pattern queries that exist at various abstraction levels in real-time. What is more, a CEP system needs to support multi-dimensional analysis of event streams at different abstraction levels. However, the state-of-art systems [CD97, HRU96, GHQ95, CKAK94, WDR06, BGAH07] either don't support pattern queries or don't support OLAP operations. Strategies for supporting queries at different concept and pattern hierarchies must be devised and efficient computation and data sharing methods among such queries need to be designed.

Multi-Query Optimization. Multiple queries can be evaluated more efficiently together than independently, because it is often possible to share state and computation. Multi-query optimization (MQO) techniques are proposed to avoid evaluating shared query subexpressions more than once. Multiple-query optimization [Sel88, RSSB00, Fin82] typically focuses on static relational databases. It identifies common subexpressions among queries such as common joins or filters. However, multiple expression sharing for stack-based pattern evaluation for CEP queries has not yet been studied.

Nested Patterns. Processing nested patterns opens many new theoretical and practical directions such as designing processing strategies for such complex nested pattern queries. Neither processing nor optimization mechanisms for nested CEP queries have been proposed in the literature to date.

1.4 Contributions of This Dissertation

The dissertation aims to solve the core issues described in Section 1.3. The dissertation focus on the design, implementation, and evaluation of a novel complex event processing methodology that tackles several of the key shortcomings of existing technologies. The proposed method for in-memory multi-dimensional sequential pattern analysis over high-speed event streams is designed to be highly efficient and scalable. The dissertation objective is to produce the detected patterns quickly and improve computational efficiency by sharing results among queries using a unified processing infra-structure. The main contributions of this dissertation include the following.

Sequence Pattern Query Processing over Out-of-Order Event Streams. The above Nested CEP and E-Cube work assume events arrive in order. We break this assumption and propose aggressive and conservative strategies respectively to process flat sequence pattern queries on out-of-order event streams. The aggressive strategy produces maximal output under the optimistic assumption that out-of-order event arrival is rare. In contrast, to tackle the unexpected occurrence of an out-of-order event and with it any premature erroneous result generation, appropriate error compensation methods are designed. The conservative method works under the assumption that out-of-order data may be common, and thus produces output only when its correctness can be guaranteed. A partial order guarantee (POG) model is proposed under which such correctness can be guaranteed. For robustness under spiky workloads, both strategies are supplemented with persistent storage support and customized access policies.

E-Cube: Multi-Dimensional Event Sequence Analysis Using Hierarchical Pattern Query Sharing. Multi-dimensional analysis over event pattern queries with concept and pattern refinement is supported. Given a set of queries, based on interrelationships in terms of both concept and pattern refinement among these queries, ECube composes the queries into an integrated E-Cube hierarchy. I design several alternate stream processing strategies that allow reuse of intermediate results along both the concept and the pattern refinement relationships between queries, thus saving computations and memory. Both strategies of drill-down (refinement from the abstract to the more specific pattern) and of roll-up (generalization from the specific to the more abstract pattern) are developed for evaluation of the given set of sequence pattern queries including negation. Design a cost-driven optimizer for multi-query execution, called Chase, that exploits the above strategies for ECube hierarchy execution. It determines an optimal global ordering for maximal re-use.

High-performance Nested CEP Query Processing over Event Streams. I identify the lack of nested CEP query syntax and of understanding their semantics in the literature. I introduce the nested CEP language *NEEL* that supports the flexible nesting of AND, OR, Negation and SEQ operators at any level. Formal semantics for the *NEEL* language are proposed. A set of equivalence rules for rewriting *NEEL* expressions satisfying our language constraints with simple predicates, along with proofs of their correctness are provided. I propose a normalization procedure that employs these rewriting rules to transform a nested CEP query with simple predicates into an equivalent non-nested query. In addition, I show proofs of its properties. By reducing forced ordering between the different level of query expressions, the normalized expression exposes opportunities for query optimization. The sequence subexpressions produced when flattening a normalized NEEL query are shown to often be similar. They share many common primitive event types. I propose several strategies for physical operators that implement the shared execution of a set of such similar yet not identical normalized subexpressions, including prefix caching, suffix clustering and a customized "bit-marking" method. These shared operators could potentially be applied to queries forming a pattern hierarchy. The size of the search space for all possible expression partitions exploiting sharing of partial computations is shown to be exponential. Thus, we propose an effective cost-based search heuristic for establishing groupings of subexpressions – each then mappable to one of the above shared execution physical operators. We thoroughly evaluate the optimized *NEEL* execution technology through experiments comparing it to the state-of-the-art technique, namely iterative nested execution. Our results confirm that our *NEEL* execution in many cases performs 100 fold faster than the traditional execution for real stock market query workloads.

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides the preliminaries of this dissertation proposal. Chapter 3 proposes the techniques for sequence pattern query processing over out-of-order event streams. Chapter 4 discusses the proposed mechanisms for multi-dimensional event sequence analysis using hierarchical pattern query sharing. Chapter 5 contains nested CEP query language, rewriting rules, a normalization procedure and shared query processing mechanism. Finally, Chapter 6 contains a discussion of the issues grouping an integration of nested, multi-dimensional and out-of-order event processing into one powerful analysis system. Chapter 7 concludes the dissertation and points out future work.

Chapter 2

Complex Event Processing Basics

2.1 Event Model

An **event instance** is an occurrence of interest denoted by lower-case letters (e.g.,'*e*'). An event instance can be either *primitive* (smallest, atomic occurrence of interest) or *composite* (a list of constituent primitive event instances).

An **event type** E of an instance e_i describes the essential features associated with the event instance e_i denoted by $e_i.type$. Each event type is associated a set of *attributes*; each attribute has a corresponding domain of possible *values*. There are tw distinguished attributes, shared by all event types, called ts and te, taking values in the natural numbers modeling time. Typically the domains will have predicates defined over them; for example we can compare timestamps by $\preceq$, etc. There may be other, domain-specific attributes. A *composite event instance* is (simply) a set of events. If $S = \{e_1, \ldots, e_n\}$ is a composite event instance, define the start and end times for S as follows:$S.ts = \min\{e_i.ts \mid 1 \leq i \leq n\}$ and $S.te = \max\{e_i.te \mid 1 \leq i \leq n\}$.

2.2 Pattern Query Language

In the following, I briefly present the language adopted from the literature [WDR06]. I will describe the proposed nested complex pattern query language in Chapter 5.

<**Query**>::= PATTERN <exp> WITHIN <window> [RETURN <set of primitive events>]

Table 2.1: Pattern Query Language

The PATTERN clause retrieves event instances specified in the event expression from the input stream. The PATTERN clause retrieves event instances specified in the event expression from the input stream. The qualification in the PATTERN clause further filters event instances by evaluating predicates applied to potential matching events. The WITHIN clause specifies a time period within which all the events of interest must occur in order to be considered a match. The time period is expressed as a sliding window, though other window semantics could also be applied. A set of histories is returned with each history equal to one query match, i.e., the set of event instances that together form a valid match of the query specification. Clearly, additional transformation of each match could be plugged in to the RETURN clause.

Operators in the PATTERN clause. The sequence operator SEQ(A a, B b) finds results composed of a and b instances where the b instance of event type B follows the a instance of event type A in an event stream within a specified time window. The AND operator AND(A a, B b) finds results composed of a and b instances within a specified time window, and their order does not matter. The OR operator

OR(A a, B b) returns results composed of either a or b within a specified time window.

2.3 State-of-the-art Pattern Query Evaluation

I will describe the operator formal semantics in Section 5.1.2.Below, I briefly describe how to evaluate each operator.

State-of-the-art Stack Based Pattern Query Evaluation. First, each pattern query q_i is compiled into a query plan. Beyond commonly used relational-style operators like select, project, join, group-by and aggregation, we support the Window Sequence operator (denoted by WinSeq(E_1 ,..., E_n, window)), Window AND operator (denoted by WinAND(E_1 ,..., E_n, window)) and Window OR operator (denoted by WinOR(E_1 ,..., E_n, window)). q_i extracts all matches of instances within the sliding stream window as specified in query q_i.

WinSeq first extracts all matches to the generating expressions specified in the query, and then filters out events based on boolean expressions as specified in the query. We briefly describe the implementation strategy of the SEQ operator. We adopt the state-of-art stack-based strategy for execution [WDR06, Jag08, GADI08]. An indexing data structure named *SeqState* associates a stack with each event type in each operator node. Each received event instance is simply appended to the end of the corresponding stack. If an event type occurs twice, we will make two stacks of the same event type. Event instances are augmented with pointers ptr_i to the most recent events in the previous stack to facilitate quick locating of related events in other stacks during result construction. The arrival of an event instance e_m of the last event type E_m of a query q_i in the topmost operator node triggers the compute function of q_i. The result construction is done by a depth first search along instance pointers ptr_i rooted at that last arrived instance e_m. All paths composed of edges "reachable" by that root e_m correspond to one matching event sequence returned for q_i. When boolean expressions are specified in WinSeq, then during sequence construction any edges "reachable" from the root e_m are skipped if an instance of the boolean expression ! E_i is found or no event instance of the $\exists$ E_i boolean constraint can be found in the corresponding stream position. Events that are outdated based on the window constraints are purged from *SeqState* when a new event instance arrives.

WinOr returns an event e if e matches one of the event expressions specified in the WinOr operator. The implementation of *WinOr* operator is straight forward. All events satisfying the event expressions listed in the *WinOr* operator are returned if these events were not outputted before.

WinAnd is designed to work like a sort-merge join. A data structure called AndState is utilized for the *WinAnd* operator. AndState associates a stack with each positive event type. In each stack of type E_i, its instances are naturally sorted from top to bottom in the order of their timestamps. All events of types listed in the *WinAnd* operator are appended at the end of the corresponding stacks. Whenever a new event instance e_i is inserted, the *WinAnd* compute is initiated. The *WinAnd* operator doesn't distinguish between the ordering of event occurrences. In *WinAnd*, we say a boolean expression ! E ($\exists$ E) is satisfied for a match of the generating expression if events of type E don't (do) exist within the window scope of the match. Purge of the *WinAnd* state removes all outdated event instances based on window constraints. Any old event instance e_i kept is purged from the bottom of stack once an event instance e_k with (e_k.ts - e_i.ts) $>$ W is received.

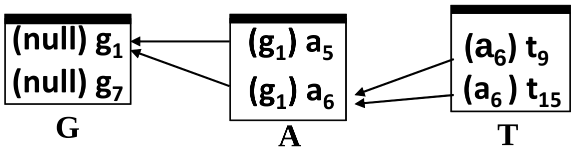


Figure 2.1: Stack Structure for q_3 in Figure 4.1

Example 1 *Figure 2.1 shows the event instance stacks for the pattern query* q_3 *= SEQ(G g, A a, T t)). In each stack, its instances are naturally sorted from top to bottom by their timestamps. When* t_{15} *of type* $Tulsa$ *arrives, the most recent instance in the previous stack of type* $Austin$ *is* a_6*. The pointer of* t_{15} *is* a_6*, as shown in the parenthesis preceding* t_{15}*. As* $Tulsa$ *is the last event type in* q_3*,* t_{15} *triggers result construction. Two results* $<g_1, a_5, t_{15}>$ *and* $<g_1, a_6, t_{15}>$ *are constructed involving* t_{15}*.*

Chapter 3

Sequence Pattern Query Processing over Out-of-Order Event Streams

In this Chapter, we will discuss how to process out-of-order events for flat SEQ queries expressed by the pattern query language in Table 3.1. The proposed techniques have been implemented and experimentally evaluated in an event processing system developed at WPI. This work has been published as one ICDE paper [LLG$^+$09] and one SIGMOD demo [WLL$^+$09].

3.1 Motivation

Consider a networked RFID system where RFID reader R_1 transmits its events to the event processing system *EPS* over a Wi-Fi network,while reader R_2 transmits over a wireless network, and reader R_3 transmits its events over a local area network. The variance in the network latencies, from milliseconds in wired LANs to 100s of seconds for a congested Wi-Fi network, often cause events to arrive out-of-sync with the order in which they were tracked by the RFID readers. Furthermore, machine or partial network failure or intermediate services such as filters, routers, or translators may introduce additional delays. Intermediate query processing servers also may introduce disorder [Mou03], e.g., when a window is defined on an attribute other than the natural ordering attribute [Cha03], or due to data prioritization [Vij99]. This variance in the arrival of events makes it imperative that the EPS can deal with both in-order as well as out-of-order arrivals efficiently and in real-time.

Out-of-order arrival of events[1], when not handled correctly, can result in significant issues as illustrated by the motivating example below. Let us consider a popular application for tracking books in a bookstore [WDR06] where RFID tags are attached to each book and RFID readers are placed at strategic locations throughout the store, such as book shelves, checkout counters and the store exit. The path of the book from the book shelf to store exit can be tracked as it passes the different RFID readers, and the events generated from the RFID readers can be analyzed to detect theft. For example, if a book shelf and a store exit register the RFID tag for a book, but the RFID tag is not read at any of the checkout counters prior to the store exit, then a natural conclusion may be that the book is being shoplifted. Such a query can be expressed by the pattern query *(S, !C, E)* which aims to find sequences of types *SHELF-READING (S)* and *EXIT-READING (E)* with no events of type *COUNTER-READING (C)* between them. If events

[1] If an event instance never arrives at our system, our model assumes that it never actually happened. Event detection and transmission reliability in a network is not the focus of our work.

of type C (negative query components) arrive out-of-order, we cannot ever output any results if we want to assure correctness of results. This holds true even if the query has an associated window. So no shoplifting will be detected. Also, operators cannot purge any event instances which may match with future out-of-order event instances. In the example above, no events of types *SHELF-READING(S)*, *COUNTER-READING(C)* and *EXIT-READING(E)* can be purged. This causes unbounded stateful operators which are impractical for processing long-running and infinite data streams. Customized mechanisms are needed for event sequence query evaluation to tackle these problems caused by out-of-order streams.

The only available method for dealing with out-of-order arrival of events, called *K-slack* [Shi04], buffers the arriving data for *K* time units. A sort operator is applied on the *K-unit* buffered input as a pre-cursor to in-order processing of events. The biggest drawback of K-slack is rigidity of the *K* that cannot adapt to the variance in the network latencies that exists in a heterogenous RFID reader network. For example, one reasonable setting of K may be the maximum of the average latencies in the network. However, as the average latencies change, *K* may become either too large, thereby buffering un-needed data and introducing unnecessary inefficiencies and delays for the processing, or too small, thereby becoming inadequate for handling the out-of-order processing of the arriving events and resulting in inaccurate results.

To address the above shortcomings, we propose two strategies positioned on the two ends of the spectrum where out-of-order events are the norm on one end and the exception in the other. In contrast to K-slack type solutions [SW04], our proposed solutions can process out-of-order tuples as they arrive without being forced to first sort them into a globally "correct" order. The conservative method designed for the scenario where out-of-order events are the norm exploits runtime streaming metadata in the form of partial order guarantee (*POG*) thereby permitting the use of unbounded stateful operators and maximally unblocking operators. Memory is effectively utilized to maintain potentially useful data. The aggressive solution designed to handle mostly in-order events outputs sequence results immediately without waiting for any potentially out-of-order events. For the unexpected scenario that out-of-order events do arise, a compensation technique is utilized to correct any erroneous results. This targets applications that require up-to-date results even at the risk of temporally imperfect results to assure delayed correctness.

3.2 Out-of-Order Event

Consider an event stream S: e_1, e_2, ..., e_n, where e_1.ats < e_2.ats < ... < e_n.ats. For any two events e_i and e_j ($1 \leq i, j \leq n$) from S if $e_i.ts < e_j.ts$ and $e_i.ats < e_j.ats$, we say the stream is an *ordered event stream*. If however e_j.ts < e_i.ts and e_j.ats > e_i.ats, then e_j is flagged as an *out-of-order event*. Stream S in Figure 3.1(a) lists events in their arrival order, thus event c_9 received after d_{17} is an out-of-order event.

3.3 Problems Caused By Out-Of-Order Data Arrival

3.3.1 Problems for WinSeq Operator

Current event stream processing systems [WDR06, Ahm04] rely on purging of the *WinSeq* operator to efficiently and correctly handle in-order event arrivals. An event instance e_i is purged when it falls out of the window W, i.e., when a new event instance e_k with e_k.ts - e_i.ts > W is received. This purging is considered

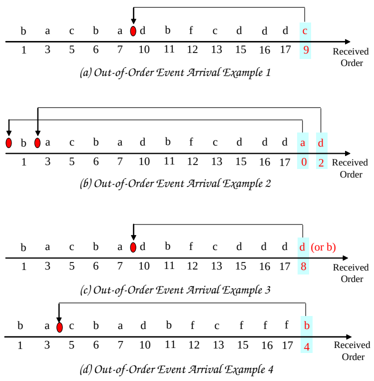


Figure 3.1: Out-of-Order Event Arrival Example

"safe" when all events arrive in-order. However, with out-of-order event arrivals such a "safe" purge of events is no longer possible. Consider that an out-of-order event instance e_j (e_j.ts $<$ e_k.ts) arrives after e_k. In this scenario, if e_k is purged before the arrival of e_j, potential result sequences wherein e_j is matched with some event e_k are lost.

While this loss of results can be countered by not purging *WinSeq* state, in practice this is not feasible as it results in storing infinite state for the *WinSeq* operator.

Example 2 *For the stream in Figure 3.1(c), suppose the out-of-order event d_8 arrives after d_{17} ($d_8.ats > d_{17}.ats$), d_8 should form a sequence output $<a_3, b_6, d_8>$ with a_3 and b_6. However* WinSeq *state purging would have already removed a_3 thus destroying the possibility for this result generation.*

Observation 1: A purge of the *WinSeq* state (*SeqState*) is "unsafe" for out-of-order event arrivals resulting in loss of results. Not applying purge to *SeqState* results in unbounded memory usage for the *WinSeq* operator.

3.3.2 Problems for WinNeg Operator

With out-of-order data arrival, window-based purge of *NegState* is also not "safe", because it may cause the generation of wrong results. A negative event instance e_i will be purged once an event e_k with $(e_k.ts - e_i.ts) > W$ is received. When an out-of-order *positive* event instance e_j ($e_j.ts < e_k$.ts) arrives after the purge of a negative event instance e_i, this may cause the *WinSeq* operator to generate some incorrect sequence results that should have been filtered out by the negative instance e_i. Similarly, an out-of-order *negative* event instance e_i may be responsible for filtering out some sequence results generated by *WinSeq* previously. In short, this *negation state purge* is unsafe, because it may cause unqualified out-of-order event sequences to not be filtered out by *WinNeg*.

Example 3 *For the stream in Figure 3.1(d), assume out-of-order event instance b_4 comes after f_{17}. Suppose* WinSeq *sends up the out-of-order sequence $<a_3, b_4, d_{10}>$ to* WinNeg. WinNeg *should determine that $<a_3, b_4, d_{10}>$ is not a qualified sequence because of the negative event c_5 between b_6 and d_8. However, if* NegState *purge would already have removed c_5, then this sequence would now wrongly be output.*

Observation 2. We observe the dilemma that on the one hand purging is essential to assure that the state size of *NegState* does not grow unboundedly. On the other hand, any purge on *NegState* is unsafe for out-of-order input event streams because wrong sequence results may be generated.

Observation 3. *WinNeg* can never safely output any sequence results for out-of-order input streams, because future out-of-order negative events may render any earlier result incorrect. Hence, *WinNeg* is a *blocking operator* causing the queries to never produce any results.

3.4 Levels of Correctness

We define criteria of output "correctness" for event sequence processing.

Ordered output. The *ordered output* property holds if and only if for any sequence result t = $<e_1, e_2, ..., e_n>$ from the system, we can guarantee that for every future sequence result t' = $<e_1', e_2', ..., e_n'>$, $e_n.ts \leqslant e'_n.ts$. We refer to sequence results that don't satisfy the property as *out-of-order output*.

Immediate output. The *immediate* property holds if and only if every sequence result will be output as soon as it can be determined that no current negative event instance filters it out.

Permanently Valid. The property *permanently valid* holds if and only if at any given time point t_{cur}, all output result sequences from the system so far satisfy the query semantics given full knowledge of the complete input sequence. That is, for any sequence result t = $<e_1, e_2, ..., e_n>$, it should satisfy (1) the sequence constraint $e_1.ts \leq e_2.ts \leq e_3.ts$ ... $\leq e_n.ts$; (2) the window constraint (if any) as $e_n.ts$ - $e_1.ts \leq W$; (3) the predicate constraints (if any) and (4) the restriction on the negation filtering (if there is a negative type E_{neg} between positive event type E_i and E_j then no current or future received event instance e_{neg} of type E_{neg} satisfies $e_i.ts \leq e_{neg}.ts \leq e_j.ts$).

Eventually Valid. We define eventually valid property to be weaker than *permanently valid*. At any time t_{cur}, all output results meet conditions (1) to (3) from above. Condition (4) is relaxed as follows: if in the query between event type E_i and E_j there is a negation pattern E_{neg} then (4.1') no e_{neg} of type E_{neg} exists in the current *NegState* with $e_i.ts \leq e_{neg}.ts \leq e_j.ts$ and (4.2') if in the future e_{neg} of type E_{neg} with e_{neg}.ats $> t_{cur}$ satisfies $e_i.ts \leq e_{neg}.ts \leq e_j.ts$, then results involving e_i and e_j become invalid.

The *permanently* and *eventually valid* defined above are two different forms of *valid* result output.

Complete output. If at time t_{cur} a sequence result t = $<e_1, e_2, ..., e_n>$ is known to satisfy the query semantics defined in (1) to (4) in the *permanently valid* category above or those defined in the *eventually valid* category then the sequence result t = $<e_1, e_2, ..., e_n>$ will also be output at time t_{cur} by the system.

Based on this categorization, we now define several notions of output correctness. Some combination of these categories can never arise. For example, it is not possible that an execution strategy produces permanently correct un-ordered results immediately. The reason is that with out-of-order event arrivals, if sequence results are output immediately then they cannot be guaranteed to remain correct in the future. Similarly, it is not possible that output tuples produced are only eventually correct and at the same time are in order. The reason is that we cannot assure that sequences sent by some later compensation computation do not lead to out-of-order output. Also, it is not possible that out-of-order tuples can be output in order yet immediately. The reason is that out-of-order event arrivals can lead to out-of-order output. We now introduce four combinations as levels of output correctness that query execution can satisfy:

- **Full Correctness**: ordered, immediate output, permanently valid and complete output.
- **Delayed Correctness**: ordered, permanently valid and eventually complete

output.

- **Delayed Unsorted Correctness**: unordered, permanently valid, and complete output.

- **Convergent Unsorted Correctness**: immediate output, 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 and we must live with delayed correctness.

In some applications *delayed unsorted correctness* may be equally accepted as strict delayed but ordered correctness. Sequence results may correspond to independent activities in most scenarios and the ordering of different outputs is thus typically not important. For instance, if book1 or book2 was stolen first is not critical to a theft detection application. Sorting the sequence results will cause increased even possibly prohibitively large response time. *Delayed Unsorted Correctness* is thus a practical requirement. For example, in the RFID-based medicine transportation scenario, between the medicine cabinet and usage in the hospital, the medical tools cannot pass any area exposed to heat nor can they be near any unsanitary location. In this scenario, correctness is of utmost importance while some delay can be tolerated.

On the other hand, in applications where correctness is not as important as system response time, then the *convergence unsorted correctness* may be a more appropriate category. The detection of shoplifting of a high price RFID tagged jewelry would require a quick response instead of a guaranteed valid one. Actions can be taken to confront the suspected thief and in the worst case, an apology can be given later if a false alarm is confirmed. In the rest of the paper, we design a solution for each of the identified categories.

3.5 Naive Approach: K-slack

K-slack is a well-known approach for processing unordered data streams [Shi04]. We now classify *K-slack approach* into the *delayed correctness* category. As described in the introduction, the *K-slack* assumption holds in situations when predictions about network delay can be reliably assessed. Large *K* as required to assure correction will add significant latency. We briefly review K-slack which can be applied for situations when the strict *K-slack* assumption indeed holds. Our slack factor is based on time units, which means the maximum out of orderness in event arrivals is guaranteed to be *K* time units. With *K* so defined, proper ordering can be achieved by buffering events in an input queue until they are at least *K* time units old before allowing them to be dequeued. We set up a clock value which equals the largest occurrence timestamp seen so far for the received events. A dequeue operation is blocked until the smallest occurrence timestamp *ts* of any event in the buffer is less than c - K, where c is the clock value.

The functionalities of *WinSeq* and *WinNeq* in the *K-slack* solution are the same as those in the ordered input case because data from the input buffer would only be passed in sorted order to the actual query system.

3.6 Proposed Aggressive and Conservative Strategies

3.6.1 Conservative Query Evaluation

Overview of Partial Order Guarantee Model We now propose a solution, called conservative query evaluation, for the category of *delayed unsorted correctness*. The general idea is to use meta-knowledge to safely purge *WinSeq* and *WinNeg* states and to unblock *WinNeg* (addressing the problems in Section 3.3). Permanent valid is achieved because results are only reported when they are known to be final. Relative small memory consumption is achieved by employing purging as early as possible.

To safely purge data, we need meta-knowledge that gives us some guarantee about the nonoccurrence of future out-of-order data. A general method for meta-knowledge in streaming is to interleave dynamic constraints into the data streams, sometimes called punctuation [Lup04].

Partial Order Guarantee Definition. Here we now propose special time-oriented metadata, which we call *Partial Order Guarantee (POG)*. *POGs* guarantee the future non-occurrence of a specified event type. *POG* has associated a special metadata schema *POG* = $<$*type*, *ts*, *ats*$>$ where *type* is an event type E_i, *ts* is an occurrence timestamp and *ats* is an arrival timestamp. *POG* p_j indicates that no more event e_i of type $p_j.type$ with an occurrence timestamp $e_i.ts$ less than $p_j.ts$ will come in the stream after p_j, i.e., ($e_i.ats > p_j.ats$ implies $e_i.ts > p_j.ts$).

Many possibilities for generating *POGs* exist, ranging from source or sensor intelligence, knowledge of access order such as an index, to knowledge of stream or application semantics [Pet03]. In fact, it is easy to see that due to the monotonicity of the time domain, such assertions about time stamps tend to be more realistic to establish compared to guarantees about the nonoccurrence of certain content values throughout the remainder of the possibly infinite stream. We note that network protocols can for instance facilitate generation of this knowledge about timestamp occurrence. Note that the TCP/IP network protocol guarantees in-order arrival of packets from a single host. Further, TCP/IP's handshake will acknowledge that certain events have indeed been received by the receiver based upon which we then can savely release the next *POG* into the stream. Henceforth, we assume a logical operator, called punctuate operator [Pet03], that embeds *POGs* placed at each stream source.

Using *POGs* is a simple and extremely flexible mechanism. If network latency were to fluctuate over time, this can naturally be captured by adjusting the *POG* generation without requiring any change of the query engine. Also, the query engine design can be agnostic to particularities of the domain or the environment. While it is conceivable that *POGs* themselves can arrive out-of-order, a punctuate operator could conservatively determine when *POGs* are released into the stream based on acknowledged receival of the events in question. Hence, in practice, out-of-order *POG* may be delayed but would not arrive prematurely. Clearly, such delay or even complete loss of a *POG* would not cause any errors (such as incorrect purge of the operator state), rather it would in the worst case cause increased output latency. Fortunately, no wrong results will be generated because the *WinNeg* operator would simply keep blocking until the subsequent *POG* arrives.

POG-Based Solution for WinSeq

POGSeq State. We add an array called *POGSeq State* to store the *POGs* received so far with one array position for each positive event type in the query. For each event type, we store the largest timestamp which is sufficient due to our assumption of *POG* ordering (see Section 3.6.1).

Tuple Processing Insert. In-order events are inserted as before. The simple append semantics is no longer applicable for the insertion of out-of-order positive event instances into the state. Instead out-of-order event $e_i \in E_i$ will be placed into the corresponding stack of type E_i in *SeqState* sorted by occurrence timestamp. The *PreEve* field of the event instance e_k in the adjacent stack with e_k.ts $> e_i$.ts will be adjusted to e_i if (e_k.*PreEve*).ts is less than e_i.ts.

Compute. In-order event insertion triggers computation as usual. The insertion of an out-of-order positive event e_i triggers an out-of-order sequence computation. This is done by a backward and forward depth first search in the DAG. The forward search is rooted at this instance e_i and contains all the virtual edges reachable from e_i. The backward search is rooted at event instances of the accepting state and contains paths leading to and thus containing the event e_i. One final root-to-leaf path containing the new e_i corresponds to one matched event sequence. If e_i belongs to the accepting (resp. starting) state, the computation is done by a backward (resp. forward) search only.

Purge. Tuple processing will not cause any state purging.

POGs Processing

Purge. The arrival of a *POG* p_k on a positive event type triggers the safe purge of the *WinSeq State*, as explained below.

Insert. If *WinSeq* receives a *POG* p_k on a positive event type, we update the corresponding *POGSeq* state POGSeq$[i] := p_k$.ts if p_k.ts is greater than the current *POG* time for p_k.type. If the positive event type is listed just before one negative event type in a query, we pass p_k to *WinNeg*. If *WinSeq* receives a *POG* p_k on a negative event type, we also pass p_k to *WinNeg*.

Definition 1 *A positive event e_i is* purge-able *henceforth no valid sequence result $<e_1, ..., e_i, ..., e_n>$ involving e_i can be formed.*

POG-Triggered Purge. Upon arrival of a *POG* p_k, we need to determine whether some event e_i with e_i.type $\neq p_k$.type can be purged by p_k. By Definition 1, we can purge e_i if it can't be combined with either current active events or potential out-of-order future events of type p_k.type to form valid sequence results.

Algorithm 1 Singleton-POG-Purge

Input: (1) Event $e_i \in E_i$ (2) $p_k \in POG$
Output: Boolean (indicating whether event e_i was purged by p_k

1 **if** ($p_k.ts<e_i.ts$) || ($p_k.type==e_i.type$)
2 **then return** false;
3 **else**
4 **if** (E_k = p_k.type listed after E_i in query Q)
5 **if** ($e_i.ts$ is within [$p_k.ts$ - W, $p_k.ts$])
6 **then return** false;
7 **else**
8 **if** (current events of type p_k.type exist
9 within $[ei.ts, ei.ts+W]$ in *WinSeq*)
10 **then return** false;
11 **else** purge event e_i; **return** true; **endif endif**
12 **else** // E_k is listed before E_i in query Q
13 **if** (no events of p_k.type exist within [$e_i.ts$ - W, $e_i.ts$] in *WinSeq*)
14 **then** purge event $e_i \in E_i$; **return** true;
15 **else return** false; **endif endif**
16 **endif**

Algorithm 1 depicts the purge logic for handling out-of-order events using *POG* semantics. In lines 1 and 2, we cannot purge e_i because an event instance e_k of p_k.type with e_k.ts $>$ p_k.ts can still be combined with e_i to form results. In lines 4, 5 and 6, we cannot purge e_i if $e_i.ts$ is within [$p_k.ts$ - W, $p_k.ts$] for e_i could be composed with an event instance e_k of p_k.type with occurrence timestamp e_k.ts $>$ p_k.ts and e_k.ats $>$ p_k.ats. In lines 8, 9, 10, we cannot purge e_i for even though p_k can guarantee no out-of-order events of type p_k.type can be combined with e_i. Some current event instance e_k can still be combined with e_i. To understand Algorithm 1, let us look at the following example.

Example 4 *Consider purging when evaluating sequence query SEQ(A, B, !C, D) within 7 mins on the data in Figure 3.1(b). Assume after receiving events* a_0 *and* d_2 *(both shaded), we receive a POG* p_k *= <A, 1> indicating that no more events of type A with timestamp less than or equal to 1 will occur. For there are no events of type A before* b_1 *in window W, we can safely purge* b_1*.*

Optimized POG-Triggered Purge. By examining only one *POG* p_k at a time, Algorithm 1 can guarantee an event e_i can be purged successfully if no event instance e_k of type p_k.type (e_i. type $\neq$ p_k.type) exists within window W. However, even though events of different *POG* types exist, they may not satisfy the sequence constraint as specified in one query. We need to make use of the knowledge provided by a set of *POGs* as together they may prevent construction of sequence results.

In Algorithm 2 from line 1 to 7, we check whether e_i can form results with event instances of type listed before E_i in Query Q. We update the *checking* value once we find an instance of p_k.type. We need to continue the instance search after timestamp *checking* for the next type in the *POGSeq state*. The checking order guarantees the sequential ordering constraint among existing event instances of *POG* types. Similarly from line 8 to 15, the algorithm checks whether e_i can form results with event instances of type listed after E_i in Query Q. Example 5 illustrates this.

Example 5 *Given the data in Figure 3.1(d), let's consider purging* a_7 *for query SEQ(B, A, B, D, F) within 10 mins. Assume after receiving* b_4*, we receive two POGs (*p_1 *= <B,17>,* p_2 *= <D,17>).* b_6 *of type B exists before* a_7*.* b_{11} *of type B exists after* a_7*. However, no existing event instances of type D exist in the time* *interval [11, 7+10]. Due to p_2, we know no future events of type D will fall into [11, 7+10]. So a_7 is purge-able.*

Algorithm 2 POG-Set-Purge

Query Q: "SEQ(E_1, E_2 ,..., E_n) within W";
Input: Event $e_i \in E_i$
Output: Boolean (whether e_i was purged by the existing *POG* Set.)

int checking = e_i.ts - W;
for (each *POG* p_k in *POGSeq* that p_k.type is before e_i.type in Q)
if (p_k.ts $>$ e_i.ts)
  if (no current event e_k of p_k.type in [checking, e_i.ts])
  then purge event $e_i \in E_i$; **return** true;
  else checking = min(e_k.ts); **endif endif**
endfor
checking = e_i.ts;
for (each *POG* p_k in *POGSeq* that p_k.type is after e_i.type in Q)
if (p_k.ts $\geq$ e_i.ts + W)
  if (no event e_k of type p_k.type in [checking, e_i.ts + W])
  then purge event $e_i \in E_i$; **return** true;
  else checking = min(e_k.ts); **endif endif**
endfor
return false

POG-Based Solution for WinNeg

POGNeg State. An in-memory array called *POGNeg State* is used to store *POGs* of negative event types sent to *WinNeg*. The length of *POGNeg* corresponds to the number of negative event types in the query. For each negative event type, we only store one *POG* with its largest timestamp so far. POGNeg$[i]$:= p_k.ts if p_k.ts is greater than the current *POG* time for p_k.type.

Holding Set. A set named *holding set* is maintained in *WinNeg* to keep the candidate event sequences which cannot yet be safely output by *WinNeg*.

Tuple Processing Additional functionalities beyond *WinNeg*are:

Insert. If *WinNeg* receives output sequence results from *WinSeq*, it stores them in the holding set. If *WinNeg* receives a negative event, *WinNeg* stores it in the negative stack.

Compute. When *WinNeg* receives sequence results, after the computation, *WinNeg* will put candidate results in the *holding set*. When *WinNeg* receives an out-of-order negative event, the negative event will remove some candidate results from the holding set per the query semantics. No results are directly output in either case.

POGs Processing

Insert. Once *WinNeg* receives a *POG* p_k on a negative (resp. positive) event type, it updates the POGNeg$[i]$ = p_k.ts.

Compute. Let us assume the sequence query $SEQ(E_1, E_2, ..., E_i, !NE, E_j, ..., E_n)$ where *NE* is a negation event type. When we receive a *POG* p_k = <*NE, ts*>, an event sequence "$e1$, $e2$ ..., e_i, e_j, ... e_n" maintained in *WinNeg* can be output from the holding set if $e_j.ts < p_k.ts$.

Now assume the negation type is at an end point of the query such as SEQ(E_1, E_2,..., E_n, !NE). Then any output sequence $<e_1, e_2, e_3, ..., e_n>$ from *WinSeq* will be put into the holding set of *WinNeg* if no NE event exists in *NegState* with a time stamp within the range of [e_n.ts, e_1.ts + W]. When we receive a *POG* p_k = <*NE*, ts> which satisfies p_k.ts $>$ $e_1.ts$ + W, this sequence can be safely output by *WinNeg*.

Example 6 *Given query SEQ(A, B, !C, D) and the data in Figure 3.1(c), when* d_{10} *is seen,* WinSeq *produces* $<a_3, b_6, d_{10}>$ *as output and sends it up to* WinNeg. *At this moment, the* NegState *of* WinNeg *holds the event instance* c_5. $c_5.ts$ *is not in the range of [6,10]. However* WinNeg *cannot output this tuple because potential out-of-order events may still arrive later. Assume after receiving event* d_{17}*, we then* *receive POG* p_i = *<C,10>. So future out-of-order events of type C, if any, will never have a timestamp less than 10.* WinNeg *can thus safely output sequence result* <a_3, b_6, d_{10}>.

Purging. For the negative events kept in the *WinNeg* state, Algorithms 1 can be utilized to safely purge *WinNeg*.

For illustration purposes, we discussed the processing of one negative event in the query. Algorithms can be naturally extended to also handle queries with more than one negation pattern.

3.6.2 Aggressive Query Evaluation

Overview We now propose the aggressive method to achieve *convergent unsorted correctness* category. The goal is to send out results with as small latency as possible based on the assumption that most data arrives in time and in order. In the case when out-of-order data arrival occurs, we provide a mechanism to correct the results that have already been erroneously output. Two requirements arise. One, traditionally streams are append-only [Dou92, GÖ05, Dan03, Arv03], meaning that data cannot be updated once it is placed on a stream. A traditional append-only event model is no longer adequate. So a new model must be designed. Two, to enable correction at any time, we need access to historical operator states until safe purging is possible. The upper bounds of *K-slack* could be used for periodic safe purging of the states of *WinSeq* and *WinNeg* operators when event instances are out of Window size + K. This ensures that data is kept so that any prior computation can be re-computed from its original input as long as still needed. Further, *WinSeq* and *WinNeg* operators must be equipped to produce and consume compensation tuples.

Given that any new event affects a limited subset of the output sequence results, we minimize run-time overhead and message proliferation by generating only new results. That is, we generate delta revisions rather than regenerating entire results. We extend the common append-only stream model to support the correction of prior released data on a stream. Two kinds of stream messages are used: **Insertion tuple** **<+, t>** is induced by an out-of-order positive event, where "t" is a new sequence result. **Deletion tuple** **<-, t>** is induced by an out-of-order negative event, such that "t" consists of the previously processed sequence. Deletion tuples cancel sequence results produced before which are invalidated by the appearance of an out-of-order negative event. Applications can thus distinguish between the types of tuples they receive.

Compensation-Based Solution for WinSeq

Insert. Same as the POG-based *WinSeq* Insert function.

Compute. In-order event insertion triggers computation as usual. If a positive out-of-order event e_i is received, e_i will trigger the construction of sequence results in *WinSeq* that contain the positive event. The computation is the same as the Compute function introduced in Section 3.6.1. If a negative out-of-order event e_i is received, the negative event will trigger the construction of spurious sequence results in *WinSeq* that have the occurrence of the negative instance between the constituent positive instances as specified in a query. These spurious sequence results will be sent up to the *WinNeg* operator followed by the negative event e_i. See Algorithm 3 for details.

Example 7 *The query is SEQ(A, !C, B) within 10 mins. For the stream in Figure 3(a), when an out-of-order negative event* c_9 *is received, new spurious sequence* *results* $<a_3, b_{11}>$, $<a_7, b_{11}>$ *are constructed in* WinSeq *for* $a_3.ts < c_9.ts < b_{11}.ts$ *and* $a_7.ts<c_9.ts<b_{11}.ts$ *and sent to* WinNeg.

Purge. If some maximal arrival delay K is known, then any event instance e_i kept in *SeqState* is safely purged once an event e_k with (e_k.ts - e_i.ts) $>$ window $W+K$ is received.

Algorithm 3 Out-of-order Processing in WinSeq

Query "EVENT SEQ(E_1, E_2, ..., E_i, !E_j, E_k, .., E_n)" within W
Input: Out-of-order Event e_t
Output: Results, Negative events

if (e_t.type==E_j)
then
  WinSeq generates spurious results $<e_1$, e_2, ..., e_i, e_k, ..., $e_n>$
  with e_i.ts $<$ e_t.ts $<$ e_k.ts and (e_n.ts - e_1.ts $\leq W$)
  and sends them to *WinNeg* along with e_i
else //e_t.type $\neq E_j$
  $<+$, e_1, e_2, ..., e_t, ..., $e_n>$ with (e_n.ts - e_1.ts $\leq W$)
  is constructed by WinSeq and sent to *WinNeg*
endif

Compensation-Based Solution for WinNeg

Insert. When candidate results or negative instances are received, *WinNeg* will insert them as usual.

Compute. If the *WinNeg* operator receives spurious results from the *WinSeq* operator, *WinNeg* first checks whether these spurious results would have been invalidated by the negative event instances already in *WinNeg* before. If not, the *WinNeg* operator will send out these spurious results as compensation tuples of the deletion type.

Purge. Same as compensation-based *WinSeq* Purge.

Example 8 *As in Example 7,* $<a_3, b_{11}>$ *and* $<a_7, b_{11}>$ *are sent to* WinNeg *as*

Algorithm 4 Out-of-order Processing in WinNeg

Query "EVENT SEQ(E_1, E_2, ..., E_i, !E_j, E_k, .., E_n)" within W
Input: 1 Results sent from WinSeq; 2 Out-of-Order Negative Event e_t
Output: Compensation tuple

if marked spurious results are received from *WinSeq*
boolean output = true;
    for each $<e_1, e_2, ..., e_i, e_k, ..., e_n>$ sent from *WinSeq*
    for each $e_j \in E_j$ stored in *WinNeg*
        if(e_i.ts $<$ e_j.ts $<$ e_k.ts)
            then output = false; **break; endif**
    endfor
    if output $==$ true
    then $<$-, $e_1, e_2, ..., e_i, e_k, ..., e_n>$ is output.
    endif
    output = true; **endfor**
endif
if results are regular (not marked spurious)
then
boolean output = true;
    for each $<e_1, e_2, ..., e_i, e_k, ..., e_n>$ or $<$+, $e_1, e_2, ..., e_i$,
        $e_k, ..., e_n>$ sent from *WinSeq*
Compute in *WinNeg*) **endfor**
endif
Insert e_t into the negative stack.

marked spurious results. (a_3, b_{11}) was filtered by c_5 in WinNeg *for $a_3.ts < c_5.ts < b_{11}.ts$. So only $<a_7, b_{11}>$ is sent out as compensation tuple $<-, a_7, b_{11}>$.*

3.7 Disk-Based Extensions

Thus far we have assumed that sufficient memory was available. However, large window sizes or bursty event streams might cause memory resource shortage during query processing. In such rare cases, we would employ a disk spilling strategy, where a block of oldest memory-resident event instances is chosen as victim and flushed to disk when the memory utilization passes a set threshold. We store historical information at the operator level, that is the states of *WinSeq* and *WinNeg* are stored as frames indexed by time. To avoid context switching, we use two separate buffers. One stores newly incoming events, and the other is dedicated to load temporarily events back from disk for out-of-order handling.

Whenever an event instance e_i arrives out of order, and its event instances within W are stored in disk, then we first need to load the event window frame into *SeqState* and *NegState*. This incurs overhead due to extra I/O costs for bringing the needed slices of the historical event stream into the buffer.

There is a tradeoff between the aggressiveness with which this process is run, and the benefits obtained. To address the tradeoff, we design policies for mode selection. One criteria we consider is the likelihood that many results would be generated by this correction processing. Assuming uniformity of query match selectivities, we use the number of out-of-order events that fall into the same logical window (physical disk page) as indicator of expected result generation productivity. Further, we employ a task priority structure to record the yet to be handled events and the correspondingly required pages.

For each page that is required to be used, we maintain the out-of-order events yet to be processed. We also keep track of the expected execution time for each page. If the total number of required times for one page is greater than the activation threshold α or the expected execution time is greater then some threshold β, we load that page and trigger the execution of tuples in this batch.

3.8 Related Work

Most stream query processing research has assumed complete ordering of input data [Shi04, Lup04]. Thus they tend to work with homogeneous streams (time-stamped relations), meaning each stream contains only tuples of the same type. The semantics of general stream processing which employs set-based SQL-like queries is not sensitive to the ordering of the data. While clearly ordering is core for the sequence matching queries we are targeting here.

There has been some initial work in investigating the out-of-order problem for generic (homogenous-input) stream systems, with the most common model being *K-slack* [Shi04, Dan03]. *K-slack* assume the data may arrive out-of-order at most by some constant *K* time units (or *K* tuples). Using *K-slacks* for state purge has limitations in practical scenarios as real network latencies tend to have a long-tailed distribution. This means for any *K* value, there exits a probability that the latency can go beyond the threshold in the future (causing erroneous results). Furthermore K-slack has the shortcoming that *WinSeq* state would need to keep events while considering only the worst case scenario (i.e., it must conservatively go with the largest network delay). Our conservative solution could easily model such K-slack assumption, yet freeing the query system from having to hard-code such knowledge.

[BGAH07] proposes a spectrum of consistency levels and performance trade-offs in response to out-of-order delivery. We borrow their basic ideas for our problem analysis, though their consistency levels are determined by the input stream blocking time in an alignment buffer and state size.

Borealis [Est06] extends Aurora in numerous ways, including revision processing. They introduce a data model to specify the deletion and replacement of previously delivered results. But their work is not designed for event systems, nor are any concrete algorithms shown for revision processing. They propose to store historical information in connection points. To design efficient customized query processing with out-of-order support, we instead store prior state information at the operator level to assure minimal information as required for compensation processing is maintained. The notion of negative tuples in [GÖ05] and revision tuples in Borealis [RMCZ06] both correspond to models to communicate compensation. Though [GÖ05] does not deal with out of order data.

[SW04] proposes heartbeats to deal with uncoordinated streams. They focus on how heartbeats can be generated when sources themselves do not provide any. Heartbeats are a special kind of punctuation. The heartbeats generation methods proposed in [SW04] could be covered by our punctuate operator. But how heartbeats can be utilized in out-of-order event stream processing is not discussed.

[Lup04, Pet03] exploit punctuations to purge join operator state. [Jin05] leverages punctuations to unblock window aggregates in data streams. We propose partial order guarantee (*POG*) based on different namely *occurrence related punctuation* semantics for event stream processing.

Our concept of classification of correctness has some relationships with levels of correctness for warehouse view maintenance categories defined in [Yue95].

Lastly, our work adopts the algebraic query architecture designed for handling sequence queries over event streams [Pra94, Mar99, WDR06]. These systems do not focus on the out-of-order data arrival problem.

Chapter 4

Multi-Dimensional Event Sequence Analysis Using Hierarchical Pattern Query Sharing

In this Chapter, we will discuss how to support multi-dimensional analysis over flat SEQ pattern queries expressed by the pattern query language in Table 3.1 with concept and pattern refinement. The proposed techniques have been implemented and experimentally evaluated in an event processing system developed at WPI in collaboration with HP Labs. This work has been published as one SIGMOD paper [LRG$^+$11b] and one ICDE demo [LRG$^+$10a].

4.1 Introduction

4.1.1 Motivation

There are numerous emerging applications, such as online financial transactions, IT operations management, and sensor networks that generate real-time streaming data. This streaming data has many dimensions (time, location, objects) and each dimension can be hierarchical in nature. One important common problem over such data is to be able to analyze multiple pattern queries that exist at various abstraction levels in real-time.

One example is data from transportation systems. In many metropolitan areas such as London, Moscow and Beijing, mass transit agencies issue their passengers near-field contactless (NFC) or contact-based smart cards for fast payment and convenient access to metros, buses, light-rails, and places such as museums. In addition to people's movements, these agencies are also beginning to continuously track the position and status of their vehicles. The collected data continuously flows to a central location in the form of structured event streams for storage. Unfortunately, their analysis lags. Officials are demanding tools that can help them analyze the current status of these complex systems in real-time and over different abstractions levels. Such knowledge would enable them to make strategic decisions about issues such as resource scheduling, route planning, variable pricing, etc. However today, they can only obtain aggregate (weekly, or even monthly) statistics through offline analysis, thus missing critical opportunities that could be gained via real-time analysis.

Another example is an evacuation system where RFID technology is used to track mass movement of people and goods during natural disasters. Terabytes of

RFID data could be generated by such a tracking system. Facing a huge volume of RFID data, emergency personnel need to perform pattern detection on various dimensions at different granularities in real-time. In particular, one may need to monitor people movement and traffic patterns of needed resources (say, water and blankets) at different levels of abstraction to ensure fast and optimized relief efforts. Figure 4.1 lists several sample "pattern queries" for such a scenario. For example, during hurricane Ike federal government personnel may monitor movement of people from cities in Texas to Oklahoma represented by the pattern SEQ(TX, OK) for global resource placement as in q_1; while local authorities in Dallas may focus on people movement starting from the Dallas bus station, traveling through the Tulsa bus station, and ending in the Tulsa hospital within a 48 hours time window as in q_5 to determine the need for additional means of transportation. The rest of the queries in Figure 4.1, including the concepts of negation, predicates and query hierarchy refinements, will be elaborated upon later in Section 4.2.

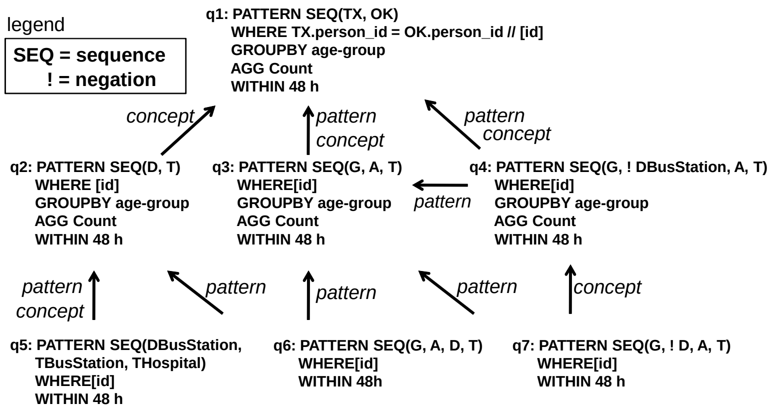


Figure 4.1: Sample pattern queries organized hierarchically.

Common across the above scenarios is a need to process and query large volumes of streaming sequence data in real-time at various abstraction levels. This is exactly the problem we tackle in this Chapter. Detecting complex patterns in high-rate event streams requires substantial CPU resources. The authors in [GWYL05] observe that with increasing stream arrival rates and large operator states, the computing resources typically become strained before the memory does. Temporary data flushing [LZR06] and highly efficient compressed data representations make a memory-limited scenario less likely. Therefore, our E-Cube solution targets the efficient processing of workloads of complex pattern detection queries at multiple levels of abstraction over extremely high-speed event streams by effectively leveraging their CPU resource utilization.

E-Cube leverages two existing technologies, OLAP and CEP. Traditional OLAP aims to provide answers to analytical queries that are multi-dimensional in nature via aggregation [CD97, HRU96, GHQ95]. Complex Event Processing (CEP) systems demonstrate sophisticated capabilities for pattern matching [CKAK94, DGP+07, WDR06] in real-time by processing huge volumes of complex stream data. However, these technologies by themselves are not always sufficient. Current CEP systems don't support queries over different concept abstraction levels. In addition, they don't support the efficient computation for multiple such queries at different concept and pattern hierarchies concurrently. In short, state-of-the-art CEP systems do not support OLAP operations, and thus are not suitable for multi-dimensional event analysis at different abstraction levels. The state-of-art OLAP solutions [LKH+08, GHL06, HCD+05] either don't support real-time streams at all, or they do not tackle CEP sequence queries. Hence, in the context of event streams where the order and sequence of events are important, OLAP is insufficient in supporting efficient event sequence analysis. Section 4.7 further discusses deficiencies of the state of art.

The rest of the Chapter is organized as follows: Section 4.2 introduces the design details of our E-Cube model and operations. Section 4.3 describes our optimal algorithm called Chase for E-Cube evaluation. Section 4.4 introduce our reuse-based pattern evaluation strategies. Section 4.5 presents plan adaption. Section 5.5 shows the evaluation results. Section 4.7 discusses related work.

Unordered (i.e., set-based) event pattern operators such as conjunctions (AND) and disjunctions (OR) can be defined in a similar manner [MM09]. Expressions with unordered event pattern operators can be rewritten into a normal form composed of AND and SEQ operators [LRG$^+$11a]. Compositions of SEQ operators can also be used to generate more complex patterns, but for brevity we leave extensions to nested queries as future work here. Instead, we henceforth focus on sequential pattern queries denoted by SEQ and their multi-dimensional analysis in this Chapter.

In the literature, handling queries with different predicates, aggregates and window sizes has been addressed by previous research using sliced time windows and shared data fragments [WRGB06, KWF06, LMT$^+$05]. In this Chapter, we instead focus on the combination of pattern and concept hierarchies as in Section 4.2.

4.2 E-Cube model

Based on the CEP query model introduced in Chapter 2, we now define our E-Cube model. A concept hierarchy is commonly used to summarize information at different levels of abstraction [HCC92]. Here, we focus on event specific features and thus on concept hierarchies over event types. A concept hierarchy applies to primitive event types in the same way as it applies to other concepts in the literature [HCC92]. Event concept hierarchies for primitive event types are predefined by system administrators using domain knowledge.

Definition 2 *An* **event concept hierarchy** *is a tree where nodes correspond to event types. The most specific event types reside at the leafs of the tree, while progressively more general event types reside higher and higher in the tree, with the most general event type residing at the apex of the tree. An event type E_k that is a descendent (resp. ancestor) of an event type E_j in an event concept hierarchy is at a finer (resp. coarser) level of abstraction than E_j, denoted by $E_k <_c E_j$ (resp. $E_k >_c E_j$)*

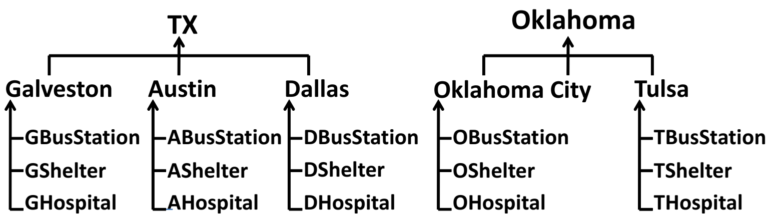


Figure 4.2: Concept Hierarchy of Primitive Event Types

Figure 4.2 shows an example event concept hierarchy for primitive event types in our RFID-based tracking scenario. We can use different dimensions to create event types that belong to a *concept hierarchy* [1]. For example, event types in our sample application incorporate semantics of both geographical locations and service station types (hospital, bus, shelter) into one hierarchy. Event instances can be interpreted to be of types at different abstraction levels in such an event concept hierarchy. For example, an instance of type DBusStation can also be interpreted to be of the more coarse types Dallas or TX. The refinement relationships among

[1]Composing over sequences does not preclude traditional set based aggregates over attribute values, but that is not our focus here.

composite event types are defined by Definitions 3 and 4. A financial concept hierarchy is given later in Figure 4.10.

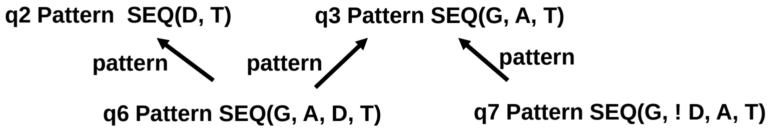


Figure 4.3: Pattern Hierarchy

Definition 3 Query Concept Refinement. *A pattern query* q_k = *SEQ(*E_{1k} *,...., !* E_{hk} *,...,* E_{mk}*) is* **coarser** *than* q_j = *SEQ(*E_{1j} *,...., !* E_{hj} *,...,* E_{mj}*), denoted by* $q_k >_c q_j$*, if (I) for all negative event types* E_{hk} *and* E_{hj}*,* $E_{hj} >_c E_{hk} \vee E_{hj}.type == E_{hk}.type$ *and (II) for all positive event types* E_{ik} *and* E_{ij}*,* $E_{ik} >_c E_{ij} \vee E_{ik}.type == E_{ij}.type$ *and (III) for* $1 \leq l \leq m$*,* $\exists$ *(*E_{lk}*,* E_{lj}*) such that* $E_{lk}.type \neq E_{lj}.type$*.*

The non-existence (existence) of a negative (positive) event type at a coarser (finer) concept level enforces more constraints as compared to a negative (positive) event type at a finer (coarser) concept level. In Figure 4.1, q_1 is at a coarser concept level than q_2, denoted by $q_1 >_c q_2$ because TX $>_c$ D and OK $>_c$ T. q_4 is at a coarser concept level than q_7, denoted by $q_4 >_c q_7$, as the negative type D in q_4 is coarser than DBusStation in q_7 (D $>_c$ DBusStation).

Definition 4 Query Pattern Refinement. *A pattern query* q_k = *SEQ(*E_{1k} *,...,* E_{ik} *,...,* E_{mk}*) is* **coarser** *than* q_j = *SEQ(*E_{1j} *,...,* E_{ij} *,...,* E_{nj}*), denoted by* $q_k >_p q_j$*, if (I)* $\forall$ $E_{lk} \in q_k$*,* $\exists$ $E_{hj} \in q_j$ *with* $E_{lk}.type == E_{hj}.type$ *and (II)* $\forall$ *(*E_{lk}*,* E_{tk}*) pairs* $\in q_k$ *with* $l < t$*, then* $\exists$ *(*E_{vj}*,* E_{wj}*) pair* $\in q_j$ *with* $v < w$ *such that* $E_{lk}.type == E_{vj}.type$ *and* $E_{tk}.type == E_{wj}.type$ *and (III)* $\exists$ E_{vj} *such that* $E_{vj} \in q_j$*,* $E_{vj} \notin q_k$*.*

In other words, we can roll-up a pattern q_k to a coarser (finer) level by deleting (inserting) one or more event types from (into) q_k. For example, in Figure 4.3, which contains a subset of the SEQ queries from Figure 4.1, the pattern query q_3 is at a coarser level than q_6, denoted by $q_3 >_p q_6$, because q_6 enforces the existence of more event types and associated sequential event relationships than q_3. Similarly, the pattern query q_3 is at a coarser level than q_7, denoted by $q_3 >_p q_7$, because q_7 includes one extra negative event type D. All event types in Figure 4.3 are at the same concept level, but at different levels in the pattern hierarchy.

Definition 5 *An* **E-Cube hierarchy** *is a directed acyclic gra-ph H where each node corresponds to a pattern query q_i and each edge corresponds to a pairwise refinement relationship between two pattern queries as defined in Definitions 3 and 4. Each directed edge $<q_i, q_j>$ is labeled with either the label "concept" if $q_i <_c q_j$ by Definition 3, "pattern" if $q_i <_p q_j$ by Definition 4 or both to indicate the refinement relationship among the two queries q_i and q_j.*

Definition 5 says that a pattern query q_i can be rolled up into another pattern query q_j by either changing one or more positive (negative) event types to a coarser (finer) level along the event concept hierarchy of that event type (by Def. 3), changing the pattern to a coarser level (by Def. 4), or both. Figure 4.1 shows an example E-Cube hierarchy. The E-Cube hierarchy helps us to achieve better performance in multi-query evaluation because it provides a blue-print for shared online pattern filtering and rapid result sharing[2], as will be explained in Section 4.4.

[2] As observed in [HCD+05], for streaming data, it is not feasible to materialize the full cube over the space of multi-level sequences. Instead we only materialize cuboids corresponding to the user queries.

Definition 6 **E-Cube** *is an E-Cube hierarchy (see Definition 5) where each pattern query is associated with its query result instances. Each individual pattern query along with its result instances in E-Cube is called an* **E-cuboid**.

Operations on E-Cube. We propose an extension of OLAP operations, namely, pattern-drill-down, pattern-roll-up, concept-roll-up and concept-drill-down for pattern queries in our E-Cube hierarchy. OLAP-like operations on E-Cube allow users to navigate from one E-cuboid to another in E-Cube.

[Pattern-drill-down] The operation pattern-drill-down(q_m, list [$Type_{ij}$, Pos_{kj}]) applied to q_m inserts a list of n event types with the event type $Type_{ij}$ into the position Pos_{kj} of q_m ($1 \leq$ j $\leq$ n).

[Concept-drill-down] The operation concept-drill-down(q_m, list [($Type_{mj}$, $Type_{nj}$), Pos_{kj}]) applied to q_{mj} drills down a list of event types from $Type_{mj}$ to $Type_{nj}$ ($Type_{mj} >_c Type_{nj}$) at the position Pos_{kj} of q_m ($1 \leq$ j $\leq$ n).

[Pattern-roll-up] The operation pattern-roll-up(q_m, list[$Type_{ij}$, Pos_{kj}]) applied to q_m deletes a list of n event types with the event type $Type_{ij}$ from the position Pos_{kj} of q_m ($1 \leq$ j $\leq$ n).

[Concept-roll-up] The operation concept-roll-up(q_m, list[($Type_{mj}$, $Type_{nj}$), Pos_{kj}]) applied to q_m rolls up a list of event types from $Type_{mj}$ to $Type_{nj}$ ($Type_{mj} <_c Type_{nj}$) at the position Pos_{kj} of q_m ($1 \leq$ j $\leq$ n).

Example 9 *In Figure 4.1, we apply a pattern-drill-down operation on q_3 = SEQ(G, A, T) specified by pattern-drill-down(q_3, [(!D, 2)]) and we get q_7 = SEQ(G, !D, A, T). We can apply a concept-drill-down operation on q_1 = SEQ(TX, OK) specified by concept-drill-down(q_1, [(TX, D, 1)]) and we get q_2 = SEQ(D, T). Similarly, we apply a pattern-roll-up operation on q_6 = SEQ(G, A, D, T) specified by pattern-*

roll-up(q_6, [(G, 1), (A, 2)]) and we get q_2 = SEQ(D, T). Also, we apply a concept-roll-up operation on q_2 = SEQ(D, T) by concept-roll-up(q_2, [(D, TX, 1)]) and we get q_1 = SEQ(TX, OK).

The results of pattern-drill-down (pattern-roll-up) can be computed by our general-to-specific (specific-to-general) reuse with only pattern changes as introduced in Section 4.4.1 (Section 4.4.4). The results of concept-drill-down (concept-roll-up) can be computed by our general-to-specific (specific-to-general) evaluation with only concept changes as introduced in Section 4.4.2 (Section 4.4.5).

Hierarchical Event Storage. We design compact *hierarchical instance stacks (HIS)* to hold event instances processed by E-Cube. HIS provides *shared storage* of events across different concept and pattern abstraction levels. Each instance is stored in only one single stack even though it may semantically match multiple event types in an event type concept hierarchy, namely, the finest one in E-Cube hierarchy. HIS is populated with event instances as the stream data is consumed. The stack based query evaluation in Section 2.3 could be easily extended to access event instances in hierarchical stacks instead of flat stacks.

4.3 Optimal E-Cube Evaluation

Our objective is to produce query results quickly and improve computational efficiency by sharing results among queries in a unified query plan. Instead of processing each pattern in our E-Cube hierarchy independently using the stack-based strategy explained in Section 2.3, we now design strategies to compute one pattern from other previously computed patterns within the E-Cube hierarchy.

More precisely, we set out to exploit the concept and pattern relationships between queries identified by the E-Cube model to promote reuse and to reduce redundant computations among queries. In particular, we consider two orthogonal aspects as in the table below, namely, (1) abstraction detection: drill down vs. roll up in E-Cube hierarchy, and (2) refinement type: pattern or concept refinement. More precisely, we consider the following cases: (a-b) general-to-specific with only pattern or concept changes respectively; (c) general-to-specific with simultaneous pattern and concept changes; (d-e) specific-to-general with only pattern or concept changes respectively; (f) specific-to-general with simultaneous pattern and concept changes.

	Direction of Reuse	
Refinement Type	General→Specific	Specific→General
Pattern Only	Section 4.4.1	Section 4.4.4
Concept Only	Section 4.4.2	Section 4.4.5
Both Refinements	Section 4.4.3	Section 4.4.6

Given a workload of pattern queries, our E-Cube system will first translate them into an E-Cube hierarchy H, and then design a strategy to determine an optimal evaluation ordering for all queries in the E-Cube hierarchy such that the total execution cost is minimized. To achieve our goal of finding the best overall execution strategy for the complete workload captured by the E-Cube hierarchy, we consider three choices when evaluating each query q_i in H;

- (I) compute q_j independently by stack-based join, denoted by $C_{compute(qj)}$;
- (II) conditionally compute q_j from one of its ancestors q_i by general-to-specific evaluation, denoted by $C_{compute(qj|qi)}$;

- (III) conditionally compute q_j from one of its descendants q_i by specific-to-general evaluation, denoted by $C_{compute(qj|qi)}$.

C_{qi} represents the computation cost which is either $C_{compute(qi)}$ or $C_{compute(qi|qj)}$ for some q_i in H. We will analyze all pairwise opportunities and detailed physical strategies of how to achieve reuse in each case along with cost models in Sections 4.4.

4.3.1 Problem Mapping to Weighted Directed Graph

Given the three alternatives (I), (II) and (III) described above, a valid execution ordering of a query workload expressed by an E-Cube hierarchy H is defined as below.

Definition 7 *An* **execution ordering** *O_i(H) for queries in an E-Cube hierarchy H represents a partial order of n computation strategies for the n queries in H, O_i(H) = $<O_{i1}$,..., O_{ij} ,..., $O_{in}>$ such that for $1 \leq j \leq n$, O_{ij} selects one of the three computation strategies (I), (II) or (III) for a query $q_j \in H$. If q_j's computation method is a conditional computation $C_{compute(qj|qi)}$ then q_i must be listed before q_j in O_i. Each query q_j is computed exactly once. Each execution ordering O_i(H) for H has an associated computation cost, denoted by Cost(O_i(H)) as shown in Equation 4.1.*

$$Cost(O_i(H)) = \sum_{j=1}^{n, q_j \in H} C_{q_j}$$
$$\text{where } C_{q_j} \text{ is equal to the cost to compute } q_j \text{ as selected by } O_{ij}; \tag{4.1}$$

For an execution ordering O_i(H), each query q_j in H is either computed from scratch or from another query q_i in H. Put differently, each query q_j has one and only one computation source. Thus clearly no computation circles can exist in an O_i(H) ordering. Let us prove this by contradiction. Given two queries q_i and q_j, assume q_i were computed from q_j and q_j were computed from q_i. Then no q_i and q_j results could ever be computed as the two queries would deadlock waiting indefinitely to compute results from each other.

Definition 8 *The* **optimal execution ordering***, denoted by O-opt(H), is the execution ordering O-opt such that* $\forall$ *i, Cost(O-opt(H))* $\leq$ *Cost(O_i(H)) with Cost() defined in Equation 4.1.*

Problem 1 *Given an E-Cube hierarchy H, the E-Cube optimization problem is to find an optimal execution ordering O-opt(H) for all queries in H as defined in Definition 8.*

We now illustrate that the E-Cube optimization problem as defined in Problem 1 can be mapped into a well-known graph problem. Given this re-formulation as shown in Definition 9, we can reuse solutions from the literature to efficiently find an optimal solution to our problem.

Definition 9 Graph Mapping. *Given an E-Cube hierarchy H, we define a directed weighted graph G = (V, E) where* $|V| = |queries \in H| + 1$; $|E| = 2 \times |edges \in H| + |queries \in H|$. *A mapping from the graph H to G, m: H → G, is defined as follows:* **(I)** $\forall$ $q_i \in H$, *there is a one-to-one mapping to one vertex* v_i *in G. To include the option of self-computation into G, we add one special vertex* v_0 *as root into V, called virtual ground.* **(II)** $\forall <q_i, q_j>$ *refinement relationships in H, there exist two edges* $e(v_i, v_j)$ *and* $e(v_j, v_i) \in E$. $\forall$ $v_i \in G$ *where* $v_i \neq v_0$, *we insert a directed edge* $e(v_0, v_i)$ *into E to model that node* v_i *is computed from "the ground"* v_0 *(i.e., from scratch).* **(III)** *Computation costs are assigned as weights on each corresponding directed edge according to our cost model (see Section 4.4 and Appendix). Each directed edge* $e(v_0, v_i) \in E$ *is assigned an associated weight* $w(v_0, v_i)$ *equal to* $C_{compute(qi)}$ *(choice I). Each directed edge* $e(v_i, v_j) \in E$ *with* $v_i \neq v_0$ *and* $v_j \neq v_0$ *is assigned a weight* $w(v_i, v_j)$ *to denote* $C_{compute(qj|qi)}$ *(choices II/III).*

Lemma 1 *All pattern and concept refinement relationships in H along with their respective computation costs are captured as edges and weights in the graph G, respectively. All possibilities of self-computation for all queries in H, along with their respective computation costs, are captured as edges and weights in the graph G.*

Proof Sketch: All independent and conditional computation relationships are captured by directed edges between vertices. Computation costs are attached to these directed edges. Thus all possible alternative solutions of computing all queries in H are now represented by G.

Example 10 *Figure 4.4(a) shows the weighted directed graph G for modeling the E-Cube hierarchy H shown in Figure 4.1. Each vertex with the number i denotes* *the query q_i from Figure 4.1. In total, eight nodes are created in the graph G representing q_1-q_7 and the virtual ground v_0. The arrow labeled with 12 from the virtual ground to v_3 represents the fact that the cost to compute q_3 from scratch is 12. The arrow labeled with 5 from v_1 to v_3 represents the fact that the cost to compute q_3 from q_1 is 5.*

4.3.2 Solution for Optimal Execution Ordering

After constructing the directed graph G, Lemma 2 and Theorem 4.2 are defined as below to solve Problem 1.

Lemma 2 *After mapping an E-Cube hierarchy H to a weighted directed graph G by Definition 9, an optimal execution ordering $O_i(H)$ for H is equal to a minimum cost spanning tree MST over G.*

Proof: Consider a directed graph, G(V, E), where V and E are the set of vertices and edges, respectively. Associated with each edge e(v_i, v_j) is a cost weight w(v_i, v_j). The MST problem is to find a rooted directed spanning tree MST of G such that the sum of costs associated with all edges in the MST is the minimum cost among all possible spanning trees. An MST is a graph which connects, without any cycle, all vertices of V in G with |V| - 1 edges, i.e., each vertex, except the root, has one and only one incoming edge. For the optimal execution ordering O-opt(H), except the virtual ground v_0 (root), every query (vertex) has one and only one computation source modeled by an incoming edge in MST. By Definition 7, no computation circles exist in O-opt(H). For each of the |V| - 1 queries (virtual ground not included), one computation source (incoming edge) is selected. |V| - 1 edges are selected such that the sum of computation costs (edge associated costs) is the minimum among all possible execution ordering O_i(H). In summary, finding an optimum execution plan with lowest cost for H is equivalent to finding an MST in G [GGST86, Edm67].

Theorem 4.1 *Solving Problem 1 for an E-Cube hierarchy H is equivalent to solving the MST problem for the corresponding G created by the mapping from H defined by Definition 9.*

Proof sketch: Proof naturally follows from Lemma 2.

Since there are many solutions in the literature for solving the well-known minimum spanning tree MST graph problem, any of these MST algorithms that works on (cyclic) directed graphs could be applied. Our optimizer, called Chase (Cost-based Hybrid Adaptive Sequence Evaluation), applies the Gabow algorithm [GGST86] in detecting the MST over a directed graph. The pseudocode for our Chase strategy is given in Figure 4.5. Line 02 in Figure 4.5 applies the Gabow algorithm [GGST86]. The key idea of the Gabow algorithm is to find edges which have the minimum cost to eliminate cycle(s) if any. The algorithm consists of two phases. The first phase uses a depth-first strategy to choose roots for growth steps. The second phase consists of expanding the cycles formed during the first phase, if any, in reverse order of their contraction, discarding one edge from each cycle to form a spanning tree in the original graph. The algorithm recursively finds the tree in the new graph until no circles exist. By braking the cycle into a tree, an MST is guaranteed to be returned eventually. For details see [GGST86].

Example 11 *The example in Figure 4.4 illustrates our use of the Gabow algorithm. The algorithm finds the edge(s) which have the minimum cost to eliminate*

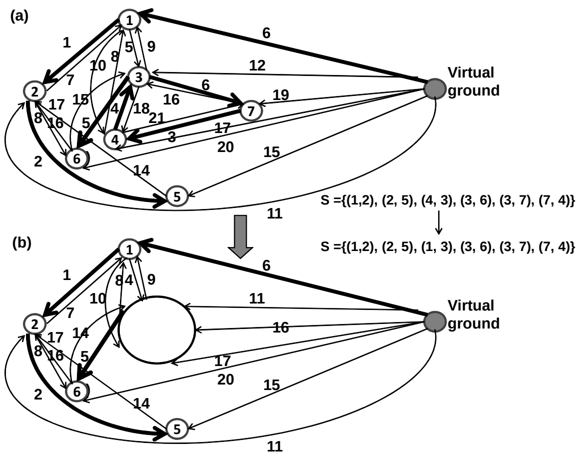


Figure 4.4: Use of Gabow Algorithm in our Optimal Solution

cycle(s) if any. For each vertex, the incoming edge with the minimum cost is selected (bold arrow) in Figure 4.4(a). We observe that vertices representing queries q_3*,* q_4 *and* q_7 *form a circle in the Gabow algorithm. In Figure 4.4(b), we observe that the edge from vertex 1 to the cycle has the minimum cost among all the ingoing edges to the circle. And vertex 1 points to vertex 3 in the cycle. Thus, the contraction technique finds the minimum cost replacing edge e(4, 3) by edge e(1, 3). Hence the cycle is eliminated.*

Theorem 4.2 *The execution ordering decided by our Chase executor (Figure 4.5) is guaranteed to find the optimal solution for the E-Cube optimization defined in Problem 1.*

Proof sketch: Since the MST algorithm [GGST86] is guaranteed to find the optimal MST solution, so is Chase.

Theorem 4.3 *The time complexity of the Chase algorithm is O(E + VlogV) [GGST86].*

```
Chase Evaluation (
Q={q1 ,..., qi ,..., qn}--Queries;
Wij-- The weight for the edge from vi to vj;
Rqi--Results of qi)
G graph = DirectedGraphConstruction(Q)
    // construct weighted directed graph for Q (Section 4.3.1)
MinimumSpannigTree(G, w) (Section 4.3.2);
    i = 0;
    // compute optimum execution ordering
    // and store in optArray
while(i <= optArray.size)
{if (compute qi independently)
compute qi by stack-based join
if(compute qi from its child qj)
compute qi by specific-to-general
    (Sections 4.4.4 4.4.5 4.4.6)
if(compute qi from its parent qk)
compute qi by general-to-specific
    (Sections 4.4.1 4.4.2 4.4.3)
cache Rqi; i++; }
```

Figure 4.5: Chase Executor

Proof sketch: As we map our optimization problem into the MST problem, the complexity of our Chase strategy is the same as that of the MST algorithm we deploy [GGST86].

Chase automatically yet efficiently optimizes the execution of a set of queries in E-Cube. Doing this operation manually would not only be time consuming but also difficult for humans to detect the optimal solution for larger E-Cube hierarchies. On the other hand, the Chase strategy clearly scales even for larger number of queries in the E-Cube hierarchy. Therefore, Chase contributes to both performance and scalability of our E-Cube system.

Table 4.1: Terminology Used in Cost Estimation

Term	Definition
$C_{compute(qi\|qj)}$	The evaluation cost for query q_i basing on evaluation results for q_j
$C_{compute(qi)}$	The cost of computing results for a query q_i independently
$\|S_i\|$	Number of tuples of type E_i that are in time window TW_P. This can be estimated as $Rate_E$ * TW_P * P_E
TW_P	Time window specified in a pattern query P
$Rate_E$	Rate of primitive events for the event type E
P_E	Selectivity of all single-class predicates for event class E. This is the product of selectivity of each single-class predicate of E.
$Pt_{Ei,Ej}$	Selectivity of the implicit time predicate of subsequence (E_i, E_j). The default value is set to 1/2.
$P_{Ei,Ej}$	Selectivity of multi-class predicates between event class E_i and E_j. If E1 and E2 do not have predicates, it is set to 1.
$\|R_E\|$	Number of results for the composite event E
C_{type}	The unit cost to check type of one event instance
$q_i.length$	The number of event types in a query q_i
$NumE$	Number of total events received so far
$NumRE$	Number of relevant events received of the types in query set Q
C_{access}	The cost of accessing one event
C_{app}	The unit cost of appending one event to a stack and setting up pointers for the event
C_{ct}	The unit cost to compare timestamp of one event instance with another one

4.4 Reuse-Based Pattern Evaluation Strategies

We now address the six alternative scenarios of reuse indicated in Section 4.3 by designing customized execution strategies for query processing that maximally reuse the previously computed results. Challenges related to partial sharing of subpatterns, extraction of non-matches via event negation, and redundancy elimination are tackled. Cost models for each of the strategies are developed.

```
General-to-specific evaluation with only pattern changes (
$q_i$ and $q_j$ are queries in a pattern hierarchy
with $q_i >_p q_j$; $R_{qi}$ -- the results of $q_i$)
$R_{q_j}$ = $R_{q_i}$
for every negative $E_k \in q_j$ but $E_k \notin q_i$
$R_{q_j}$ = checkNegativeE( $R_{q_j}$, $E_k$, $q_j$)
for every positive $E_i \in q_j$ but $E_i \notin q_i$
 if(joining events in $R_{q_j}$ and $E_i$ are
 sorted and pointers exist)
 $R_{q_j}$ = stack-based-join( $R_{q_j}$, $E_i$);
 else if(events are sorted with no pointers)
 $R_{q_j}$ = merge-join( $R_{q_j}$, $E_i$);
 else $R_{q_j}$ = sorted-merge-join( $R_{q_j}$, $E_i$);
checkNegativeE($R_{q_j}$, $E_k$, $q_j$)
for each result $r_i \in R_{q_j}$
 if($E_k$ events exist in the specified interval)
 remove $r_i$
```

Figure 4.6: General-to-Specific Evaluation in Pattern Hierarchy

4.4.1 General-to-Specific with Pattern Changes

Considering only pattern changes, the computation of the lower level query can be optimized by reusing results from the upper level query. The two sharing cases are stated as below. Given queries q_i and q_j ($q_i >_p q_j$) in a pattern hierarchy and the results of q_i, then the results for q_j can be constructed as bellow. In **case I: Differ by positive types,** we join the results of q_i with the events of positive types listed in q_j but not in q_i. In **case II: Differ by negative types** we filter the results from q_i that don't satisfy the sequence constraints formed by negative event types listed in q_j but not in q_i. Figure 4.6 depicts the pseudocode for general-to-specific evaluation guided by the pattern hierarchy.

For **case I** above, the costs for the compute operation depend on two key factors, namely (1) if pointers exist between joining events and (2) if the re-used result is ordered or not on the joining event type. Assume two pattern queries q_i = SEQ(E_i, E_j, E_k) and q_j = SEQ(E_i, E_j, E_k, E_m, E_n) differ by two positive event types E_m and E_n. Also, let us assume pointers exist between events of type E_m and E_n. To compute q_j, we first construct results for SEQ(E_m, E_n) by an efficient stack-based join. These results will by default be sorted by E_n's timestamp. We then join these results with q_i results using the most appropriate join method. Table 4.1 shows the factors used in the cost estimation in Equation 4.2.

$$\begin{aligned} C_{compute(qj|qi).gp} =& |S_m| * |S_n| * Pt_{E_m,E_n} * P_{E_m,E_n} \\ & + |R_{SEQ(E_m,E_n)}| log |R_{SEQ(E_m,E_n)}| \\ & + |R_{qi}| * |R_{SEQ(E_m,E_n)}| * Pt_{E_k,E_m} \\ & * P_{E_k,E_m} + |R_{SEQ(E_m,E_n)}| + |R_{qi}| \end{aligned} \tag{4.2}$$

For **case II**, assume two pattern queries q_i = SEQ(E_m, E_n) and q_j = SEQ(E_m, ! E_k, E_n) differ by one negative event type E_k. For every q_i result, it can be returned for q_j if no E_k events are found between the particular interval in q_j. The cost formula is shown in Equation 4.3.

$$\begin{aligned} C_{compute(qj|qi).gp} =& |S_m| * |S_n| * Pt_{E_m,E_n} * P_{E_m,E_n} * \\ & (1 - Pt_{E_m,E_k} * Pt_{E_k,E_n}) \end{aligned} \tag{4.3}$$

Besides this computation sharing, we can also achieve online pattern filtering and thus potentially save the computation costs of q_i completely ($C_{compute(qi)}$). The idea is that, if a pattern q_i is at a coarser level than a pattern q_j, and a matching attempt with q_i fails, then there is no need to carry out the evaluation for q_j. That is, q_j being stricter is guaranteed to fail as well.

Example 12 *Given pattern queries q_3, q_6 and q_7 in Figure 4.1, q_3 and q_6 differ by one event type D and q_3 and q_7 differ by one event type $!D$. We check the results for q_3 first. If no new matches are found, then we know that the results for q_6 and q_7 would also be negative. Thus, we can skip their evaluation. If new matches for q_3 are found, as no pointers exist between results of q_3 and events of type D. Yet the joining attributes for T and D, namely, $D.ts$ and $T.ts$ are sorted on timestamps. We thus can apply the fairly efficient merge join to compute q_6.*

4.4.2 General-to-Specific with Concept Changes

Considering only concept changes, composite results constructed involving events of the highest event concept level are a super set of pattern query results below it in a E-Cube hierarchy. The lower level query can be computed by reusing and further filtering the upper query results.

Given two pattern queries q_i and q_j with only concept changes ($q_i >_c q_j$) on **positive** event types, our cost model is formulated in Equation 4.4. For each result of q_i, we interpret the event types for the constructed composite event instances to determine which of them indeed match a given lower level type. The strategy becomes less efficient as the number of results to be re-interpreted increases.

$$C_{compute(qj|qi).gc} = |R_{qi}| * C_{type} * q_i.length \tag{4.4}$$

Example 13 *In Figure 4.1, from q_1 to q_2 only the concept hierarchy level is changed. q_1 is computed before q_2 and the results are cached. As all results of q_2 satisfy q_1, q_2 can be computed simply by re-interpreting the q_1 results. If one result with component events of types TX and OK is also a composite event with types D and T, that particular result will be returned for q_2. Otherwise, the result will be filtered out.*

Given two pattern queries q_i = SEQ(E_m, ! E_{k1}, E_n) and q_j = SEQ(E_m, ! E_k, E_n) with only concept changes ($q_i >_c q_j$) on **negative** event types where E_k is a super concept of E_{k1} in the event concept hierarchy. To facilitate query sharing, we rewrite q_j into the expression shown in Equation 4.5. For every q_i result, it can be returned for q_j if no E_{k2}, E_{k3} ... and E_{kn} events are found between the position in specified query.

$$SEQ(E_m, !E_k, E_n) = SEQ(E_m, !E_{k1} \wedge ...! \wedge E_{kn}, E_n) \tag{4.5}$$

Example 14 *In Figure 4.1, when computing q_7 from q_4 , each q_4 result is qualified for q_7 if no DHospital and DShelter events exist between G and A events.*

4.4.3 General-to-Specific with Concept & Pattern Refinement

Given q_i and q_j in an E-Cube hierarchy with simultaneous concept and pattern changes ($q_i >_{cp} q_j$), the cost to compute the child q_j from the parent q_i corresponds to Equation 4.6. The main idea is to consider this as a two step process that composes the strategies for concept and then pattern-based reuse (or, vice versa) effectively with minimal cost.

$$C_{compute(qj|qi)} = \min_{p}(C_{compute(p|qi)} + C_{compute(qj|p)})$$
$$\text{where } p \text{ has either only concept or only pattern changes from } q_i \text{ and } q_j\text{, respectively.} \tag{4.6}$$

4.4.4 Specific-to-General with Pattern Changes

Given queries q_i and q_j ($q_i >_p q_j$) in a pattern hierarchy and the results of q_j, then q_i can be computed by reusing q_j results and unioning them with the delta results not captured by q_j. Our compute operation includes two key factors, namely, *result reuse* and *delta result computation*. Figure 4.7 depicts the pseudocode for the specific-to-general evaluation.

In general, assume q_i = SEQ(E_i, E_j, E_k) is refined by an extra event E_m into q_j = SEQ(E_i, E_m, E_j, E_k). q_j results are reused for q_i and SEQ(E_i, ! E_m, E_j, E_k) results are the delta results. The cost model is given in Equation 4.7. This specific-to-general computation for a pattern hierarchy would need to check the non-existence of a possibly long intermediate pattern for delta result computation when two queries differing by more than one event type. These overhead costs in some cases may not warrant the benefits of such partial reuse. When two queries differ by **negative** event types, the specific-to-general method is similar to above except that during delta result computation we need to compute some additional sequence results filtered in the specific query due to the existence of events of negative types.

Specific-to-general evaluation with only pattern changes (q_i and q_j are queries in a pattern hierarchy with $q_i >_p q_j$; R_{qi} -- the results of q_i)

R_{q_i} = ReuseSubpatternResult(q_i, q_j, R_{q_j})
R_{q_i} = R_{q_i} $\cup$ $ComputeDeltaResults$(q_i, q_j)

ReuseSubpatternResult(q_i, q_j, R_{q_j})
for each result $r_k \in R_{q_j}$
for each component $e_i \in r_k$
if(e_i.type $\notin q_j$ $\wedge$ e_i.type $\in q_i$)
remove e_i from r_k;

ComputeDeltaResults(q_i, q_j)
for each positive event type E_i or SEQ(E_i ,..., E_k)$\in q_j$ but $\notin q_i$
construct results for q_i with events failed in q_j due to non-existence of E_i or SEQ(E_i, E_j, ..., E_k) events
for each negative event type $E_i \in q_j$ but $\notin q_i$
construct results for q_i with events failed in q_j due to existence of E_i events

Figure 4.7: Specific-to-General Evaluation in Pattern Hierarchy

$$
\begin{aligned}
C_{compute(qi|qj).sp} = & |R_{qj}| * C_{type} * q_j.length + |S_k| * |S_j| \\
& * Pt_{E_j,E_k} * P_{E_j,E_k} + |S_k| * |S_j| \\
& * Pt_{E_j,E_k} * P_{E_j,E_k} * |S_i| * P_{E_i,E_j} \\
& * P_{E_i,E_j} * (1 - P_{E_i,E_j} * P_{E_m,E_j} * \\
& P_{E_i,E_j} * P_{E_m,E_j})
\end{aligned}
\tag{4.7}
$$

Example 15 *Figure 4.8 shows the hierarchical instance stacks for pattern queries q_3 and q_6 in Figure 4.1. Result reuse and delta result computation for q_3 are explained below.*

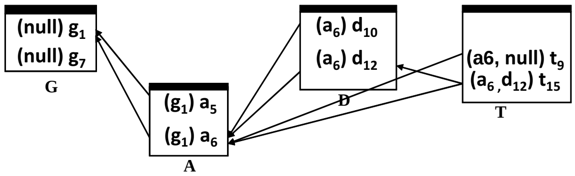


Figure 4.8: Stack Structure for q_3 and q_6 in Figure 4.1

ReuseSubpatternResult. *q_3 is computed from the results of q_6 by subtracting subsequences composed of positive event types G, A and T. For example, in Figure 4.8, the result $< g_1, a_5, d_{10}, t_{15} >$ for q_6 is first generated using the stack-based join method. Then $< g_1, a_5, t_{15} >$ is prepared for q_3 by removing the event d_{10} of the event type D, because D is not listed in q_3. Lastly, we check whether this result is duplicated before returning it for q_3.*

ComputeDeltaResults. *Some sequences may not have been constructed for q_6 due to the non-existence of events of type D. However, such sequence results must now be constructed for q_3. In this case, each instance of type T has one pointer to an A event for q_3 and another pointer to a D event for q_6. Hence, for a T event that doesn't point to any D event, we can infer that a sequence involving this T event would not have been constructed for q_6. This T event thus should trigger its sequence construction for q_3 by a stack-based join. If one T event points to both an A and a D event, then the A and D events may still not satisfy the time constraints. If the timestamp of the A event is greater than the timestamp of the D event, sequence construction is triggered by such T event for q_3. In Figure 4.8, we observe that t_9 doesn't point to any D event. Hence sequence results $< g_1, a_5, t_9 >$ and $< g_1, a_6, t_9 >$ are constructed for t_9 by a stack-based join. The conditional cost to compute q_3 includes the costs of result reuse and the cost to compute $SEQ(G, A, !D, T)$ results.*

4.4.5 Specific-to-General with Concept Changes

The result set of a higher concept abstraction level is a super set of all the results of pattern queries below it. Thus upper level query can be computed in part by reusing the lower level query results. The lower level pattern query is computed first. Then all these results are also returned for the upper level pattern. In addition, the events of the higher event type concept level not captured by the lower queries must also be constructed. Such specific-to-general computation requires no extra interpretation costs as compared to the general-to-specific evaluation. Given two pattern queries q_i and q_j with only concept changes ($q_i >_c q_j$), our cost model is formulated by Equation 4.8.

$$C_{compute(qi|qj).sc} = C_{compute(qi)} - C_{compute(qj)} \tag{4.8}$$

Example 16 *Figure 4.9 shows the hierarchical instance stacks for q_1 to q_2 in Figure 4.1. From q_1 to q_2 only concept relationships are refined. Results for q_2 $\{dh_{10}$, $ts_{33}\}$, $\{dh_{16}$, $ts_{33}\}$ are computed first. And these results are also returned for q_1. Next, we need to compute the delta results belonging to q_1 that were not captured by q_2. In Figure 4.9, the pointers between D and T are already traversed during the evaluation of q_2. The other pointers between D and OK, TX and OK, TX and T need now to be traversed. Results $\{ah_{12}, oh_{15}\}$, $\{ah_{10}, oh_{15}\}$, $\{ah_{12}, oh_{38}\}$, $\{as_{18}, os_{38}\}$, $\{dh_{10}, os_{38}\}$, $\{dh_{18}, os_{38}\}$, $\{ah_{12}, ts_{33}\}$, $\{as_{18}, ts_{33}\}$ are constructed for q_1.*

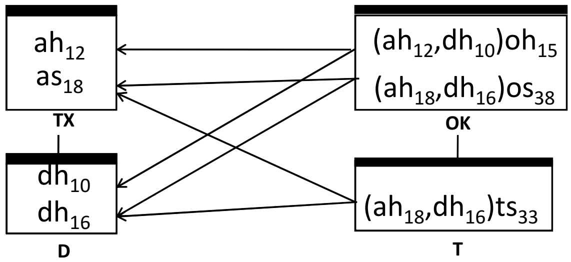


Figure 4.9: Stack Structure for q_1 and q_2 in Figure 4.1

4.4.6 Specific-to-General with Concept & Pattern Refinement

Given q_i and q_j in an E-Cube hierarchy with simultaneous concept and pattern changes ($q_i >_{cp} q_j$), we first find one intermediate query p with either only concept or pattern changes from q_j so that query p minimizes Equation 4.9. As above, we then compute results in two stages from q_j to p and from p to q_i by using specific-to-general evaluation with first only pattern and then only concept changes or vice versa effectively with minimal cost.

$$C_{compute(qi|qj)} = \min_p (C_{compute(p|qj)} + C_{compute(qi|p)})$$
$$\text{where } p \text{ has either only concept or only pattern changes from } q_i \text{ and } q_j\text{, respectively.} \tag{4.9}$$

4.5 Plan Adaptation

High variability in input stream rates and selectivities may render an initially optimal execution ordering not optimal or possibly even ineffective after some time. A query could be added to or removed from the system as well. To recompute the query execution order on the fly, we maintain a running estimate of the statistics.

When the statistics vary by more than some error threshold θ, we re-run the Chase optimizer in a separate system thread to generate a new ordering recommendation. If the performance improvement predicted by the cost model is greater than a given performance threshold γ, we then install the new updated plan.

To change the execution ordering on the fly, we would need to simply switch from utilizing one result buffer to another buffer space for conditional computation. The process for changing the query execution ordering on-line thus uses the following steps:

1. Discard intermediate results based on the execution ordering after finishing the result computation for the current input event e_i;
2. Rebuild intermediate results based on the newly determined execution ordering as if it were the first round before starting to process the next instance e_{i+1} from input stream. No results are output during this preparation stage.

The advantage of our adaptation method is its simplicity. More sophisticated adaptive strategies that may incrementally reuse some of the intermediate results to minimize the recalculation effect [ZRH04] could be designed. However, the complexity of such a method may offset its potential gains. We thus leave this analysis as future work.

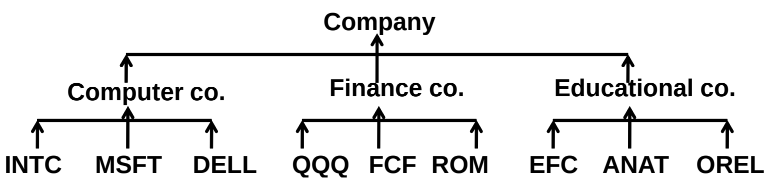


Figure 4.10: Company Concept Hierarchy

4.6 Performance Evaluation

The primary objective of our experimental study is to compare four different strategies, namely, *state-of-the-art*, *a pure top-down*, *a pure bottom-up* and *our optimized Chase* strategies for E-Cube evaluation, and to determine their respective scope of applicability. As explained in Section 2.3, the state-of-the-art method processes queries independently using stack-based query evaluation [WDR06]. The *top-down* (*bottom-up*) method proceeds by evaluating general (specific) patterns first and then iteratively processing patterns lower (higher) in the E-Cube hierarchy (Section 4.4). Finally, the Chase method applies the Chase optimizer to construct and then utilize the optimal cost-based reuse strategy (Section 4.3).

4.6.1 Experimental Setup

We implement our proposed E-Cube framework inside the X[3] stream management system [GWA+09a] using Java. We ran the experiments on Intel Pentium IV CPU 2.8GHz with 1GB RAM. We evaluated our techniques using real stock trades data from [sto]. The data contained stock ticker, timestamp and price information. We used sliding window of size 1 second in the experiments. The portion of the trace we used contained 10000 unique event instances. The arrival rate was set to 2000 tuples/sec. Stock data is served all up "immediately". But data is processed in terms of time windows based on the application timestamp attached. A concept hierarchy for stock companies is built as in Figure 4.10. The performance metric *result latency* is the accumulative time difference between the sequence output time and the arrival time of the latest event instance composed into the sequence

[3] name removed for sigmod anonymous reviewing.

result. We compared the result latency of various strategies using different pattern query sets. Specifically, the financial sector is very sensitive to query result latency and uses extensive CPU resources to achieve this goal. We start with controlled query sets where we control one single parameter (pattern or concept) and later also conduct larger typical workloads mixed the two types of workloads to demonstrate a more realistic concurrent CEP query processing scenario. The results are extremely encouraging showing benefits of using our adaptive Chase strategy over all other methods.

We first tested the cost models (Equations 4.2-4.9) to verify that they accurately reflect the system performance. We ran these experiments on all the pattern query workloads given below. We found that the estimates produced by our cost model for the four methods correctly reflected the actual system behavior of the four alternative methods (state-of-the-art, top-down, bottom-up and Chase).

4.6.2 Scenarios with Pattern Hierarchy Queries

In this first experiment, we compare the four methods (state-of-the-art, top-down, bottom-up and Chase) evaluating queries forming a pure pattern hierarchy (i.e., no concept changes). The root query size is increased from 3 to 5 in the workloads 1, 2 and 3. Figure 4.11(a) shows the average result latency (ms) of the four methods and speedup of the top-down (chase) method over the state-of-the-art method. Figure 4.11(b) shows the accumulative result latency for workload 2. We observe that the top-down method generates results faster than the state-of-the-art and the bottom-up methods. It outperforms the others because it avoids result recomputation by applying conditional computation. We also notice that the average latency difference between the state-of-the-art method and the top down increases as the result sharing length increases from 3 to 5 due to reuse and computational savings. The speed-up factor of the method chosen by Chase over the state-of-the-art method starts at x8 at length 3, increasing to x10 and x28 for lengths 4 and 5, respectively. The bottom up method generates results slower than the other methods, because it introduces an extra delta result computation cost (see Section 4.4.4 for explanation).

```
Workload 1 (shared length 3):
  q1 = SEQ(INTC, ! MSFT, FCF)
  q2 = SEQ(INTC, ! MSFT, FCF, ROM)
  q3 = SEQ(INTC, ! MSFT, FCF, EFC)
  q4 = SEQ(INTC, ! MSFT, FCF, ANAT)
  q5 = SEQ(INTC, ! MSFT, FCF, OREL)

Workload 2 (shared length 4):
  q6 = SEQ(DELL, INTC, ! MSFT, FCF)
  q7 = SEQ(DELL, INTC, ! MSFT, FCF, ROM)
  q8 = SEQ(DELL, INTC, ! MSFT, FCF, OREL)
  q9 = SEQ(DELL, INTC, ! MSFT, FCF, ANAT)
  q10= SEQ(DELL, INTC, ! MSFT, FCF, QQQ)


Workload 3(shared length 5):
  q11 = SEQ(QQQ, DELL, INTC, ! MSFT, FCF)
  q12 = SEQ(QQQ, DELL, INTC, ! MSFT, FCF, ROM)
  q13 = SEQ(QQQ, DELL, INTC, ! MSFT, FCF, ANAT)
  q14 = SEQ(QQQ, DELL, INTC, ! MSFT, FCF, OREL)
  q15 = SEQ(QQQ, DELL, INTC, ! MSFT, FCF, EFC)
```

4.6.3 Scenarios with Concept Hierarchy Queries

Next, we compare methods for evaluating query workloads with only concept changes. We ran experiments on workloads 4, 5 and 6 below. Figure 4.11(c) shows the average result latency of the three methods for each workload and Figure 4.12(a) shows the accumulative result latency for workload 4. We observe that the bottom up method now produces results faster than the other methods. This is because results from q_{17}, q_{18} and q_{19} are reused for q_{16}. The top down method is better than the state-of-the-art method in workload 4 because a large percentage of q_{16} results match the child query q_{17} (only one concept change). The top down method does even worse than the state-of-the-art method in workloads 5 and 6. This is because in the top down method, we need to check the types of component events for each result of q_{16}. When only a small percentage of q_{16} results match children queries q_{18} and q_{19}, direct result computation (state-of-the-art method) is better than result interpretation (top down method) in the concept hierarchy.

```
Workload 4:
  q16 = SEQ(Computer, Finance, Education)
  q17 = SEQ(Computer, Finance, EFC)


Workload 5:
  q16 = SEQ(Computer, Finance, Education)
  q18 = SEQ(Computer, QQQ, EFC)


Workload 6:
  q16 = SEQ(Computer, Finance, Education)
  q19 = SEQ(INTC, QQQ, EFC)
```

4.6.4 Scenarios with Representative Mixed Workloads

We compare the four methods with workloads involving both concept and pattern changes. This Chase optimizer took 16 ms to find the optimal execution ordering. We designed workloads 7 and 8 to be representative and interesting mixes of changes. DELL stock belongs to Computer and QQQ, FCF, ROM stocks belong to Finance. EFC, ANAT and OREL stocks belong to Education. Figures 4.12(b) and 4.12(c) show the accumulative result latency of the four methods, respectively. As expected, Chase produces results faster than the others. On closer analysis in Chase for workload 7, q_{20} is executed first and its results are reused for q_{27} using the bottom up method and for q_{21}, q_{22}, q_{23} and q_{24} by the general-to-specific evaluation. Results of q_{24} are reused for q_{25} using the general-to-specific evaluation. Results of q_{27} are reused for q_{26} and q_{16} by the specific-to-general evaluation and for q_{28} by the general-to-specific evaluation. Workload 8 is similar to workload 7. In other words, Chase carefully selects the optimal combination of execution and reuse strategies.

```
Workload 7:
  q20 = SEQ(DELL, QQQ, ANAT)
  q21 = SEQ(DELL, QQQ, ANAT, ROM)
  q22 = SEQ(FCF, DELL, QQQ, ANAT)
  q23 = SEQ(DELL, QQQ, ANAT, OREL)
  q24 = SEQ(DELL, QQQ, ANAT, INTC)
  q25 = SEQ(DELL, QQQ, ANAT, INTC, EFC)
  q16 = SEQ(Computer, Finance, Education)
  q26 = SEQ(Computer, Finance, ANAT)
  q27 = SEQ(DELL, Finance, ANAT)
  q28 = SEQ(QQQ, DELL, Finance, ANAT)


Workload 8:
  q16 = SEQ(Computer, Finance, Education)
  q29 = SEQ(Computer, Finance, OREL)
  q30 = SEQ(INTC, QQQ, Education)
  q31 = SEQ(INTC, QQQ, EFC)
```

```
q32 = SEQ(MSFT, INTC, QQQ, EFC)
q33 = SEQ(INTC, QQQ, EFC, DELL)
q34 = SEQ(Computer, ROM, Education)
q35 = SEQ(Computer, ROM, ANAT)
q36 = SEQ(INTC, ROM, ANAT)
```

Accumulative CPU processing time means the wall clock time for processing an item e_i in stock trades measured by ($T_{end.ei}$ - $T_{start.ei}$) where $T_{start.ei}$ represents the system time when our processing engine starts processing the data item e_i and $T_{end.ei}$ represents the system time when the engine finishes processing the data item e_i. It is an atomic process, i.e., our processing engine won't stop processing that tuple until it is fully processed. In a complementary set of experiments we measure the CPU-only execution time as shown in Figures 4.13-4.14. These experiments were conducted using the same workloads 1-8. This finding shows that the strategies are mostly CPU-bound and not I/O bound. Other findings include (1) The top down method runs on average 10 fold faster than the state-of-the-art and the bottom up methods for queries with only pattern changes as depicted in Figures 4.13(a), 4.13(b). (2) The bottom up method runs on average 2 times faster than the state-of-the-art and the top down methods for queries with only concept changes as in Figure 4.13(c), 4.14(a). (3) For a mixed workload, the Chase method constantly outperforms the other methods as shown in Figures 4.14(b), 4.14(c).

4.7 Related Work

Traditional OLAP focuses on static pre-computed and indexed data sets and aims to quickly provide answers to analytical queries that are multi-dimensional in nature [CD97, HRU96, GHQ95]. OLAP techniques allow users to navigate the data at different abstraction levels. However, the state-of-the-art OLAP technology tends to be set-based instead of sequence based [GHQ95]. Further, aggregation (count, sum, max, ave) is conducted over scalar values, namely, the set of values within a single column such as salary, and not over ordered sequences. Hence, in the context of event patterns where the order of events is important, OLAP is insufficient in supporting efficient multi-dimensional event sequence analysis.

The state-of-art OLAP solutions [LKH$^+$08, GHL06, HCD$^+$05] either don't support real-time streams at all, or they do not tackle CEP sequence queries. The work that is most closely related to ours is Sequence OLAP [LKH$^+$08] which proposed to support OLAP operations for sequences. However, sequence OLAP does not support the notion of concept refinement for pattern queries as done in our work. Second, sequence OLAP preprocesses all data off-line, and then inserts the data into inverted indices. Thereafter, the results are joined using the inverted indices. In short, Sequence OLAP neither supports incremental maintenance of its precomputed index, nor streaming, nor negation in sequence - while these are all contributions of our work. Such (static) techniques used in Sequence OLAP are inappropriate in a stream setting.

A second related work is Flow Cube [GHL06] which constructs a data warehouse of RFID-tagged commodity flow. The commodity flowgraph captures the major movement trends and significant deviations of the items over time. It can be viewed at multiple levels by changing the level of abstraction of path stages. However, it neither support streaming data nor concept hierarchies. Furthermore, it does not consider any optimization algorithms for hierarchical pattern query evaluation such as sequence reuse nor the cost-driven Chase method which is our core contribution. This line of work also does not consider event negation, which is covered in our system. Lastly, Stream Cube [HCD$^+$05] has recently been proposed to facilitate online multi-dimensional analysis of stream data. However, it provides neither result reuse strategies nor any cost analysis for pattern queries including neither sequence nor negation.

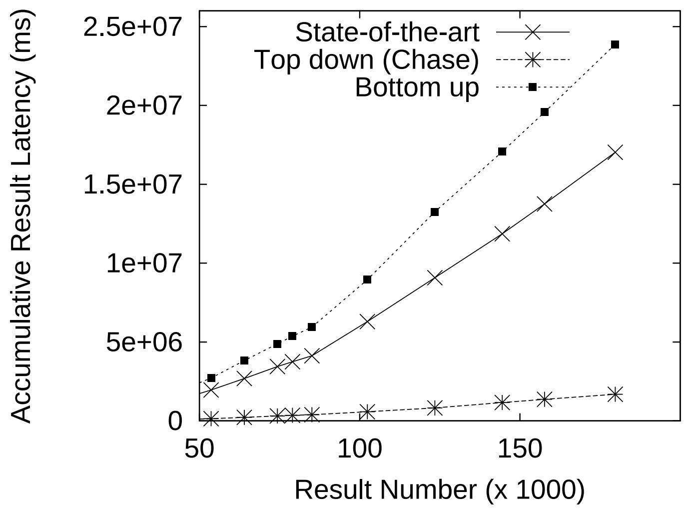
Complex Event Processing (CEP) systems demonstrate sophisticated capabilities for pattern matching [CKAK94, DGP$^+$07, WDR06]. Yet, they do not support OLAP-like operations for multi-dimensional event sequence analysis at different abstraction levels. We borrow a variety of techniques from CEP, including stack-based joins [WDR06] and cost models for stack-based joins [MM09]. However, work in CEP has not studied hierarchical pattern refinement relations, such as concept hierarchies as proposed in our work. CEP systems such as Cayuga [DGP$^+$07, HRK$^+$09], SASE [WDR06] and ZStream [MM09] focus on event sequence detection over streams. However, these systems do not address the issue of supporting queries at different concept and pattern hierarchies nor do they design efficient computation strategies for processing multiple such queries. Recently, work in CEP has considered pushing negation into sequence processing [MM09]. We exploit this as part of our proposed solution for determining if additional delta results must be generated in the specific-to-general reuse.

Multiple-query optimization (MQO) in databases [Sel88, RSSB00, Fin82], typically focussed on static relational databases. MQO identifies common subexpressions among queries such as common joins or filters. Multiple-query optimization (MQO) for stack-based pattern evaluation for CEP queries has not yet been studied, in particular, sharing for CEP queries with negation and concept refinements was an open problem prior to our work.

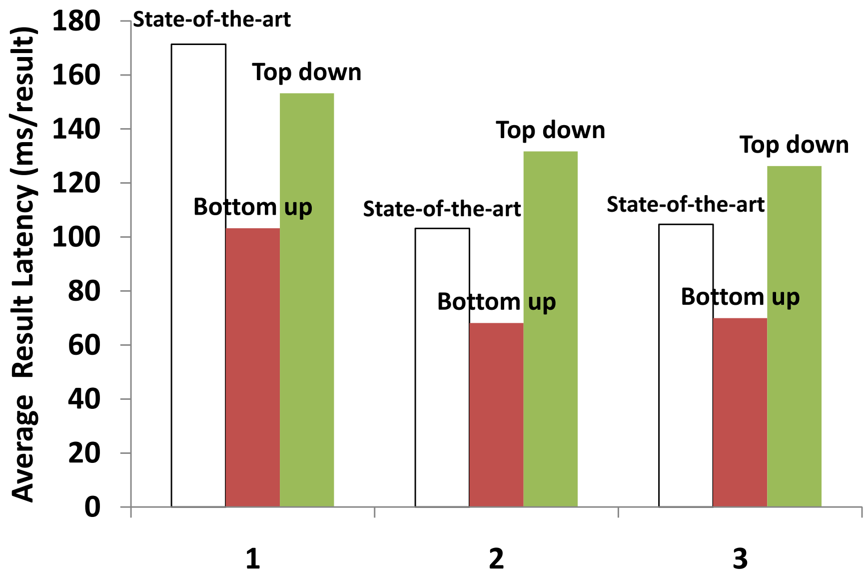
Lastly, [CHC+06] proposes sharing among XML queries, in particular, prefix sharing and suffix clustering. However, they neither consider concept nor pattern hierarchies.

Length	State-of-the-art	Top down (Chase)	Bottom up	Speedup
3	0.835	0.11	1.29	7.59
4	96.415	9.71	136.39	9.93
5	5252.5	188.16	11593	27.92

(a) Workload with only Pattern Changes: Average Result Latency (ms/result)

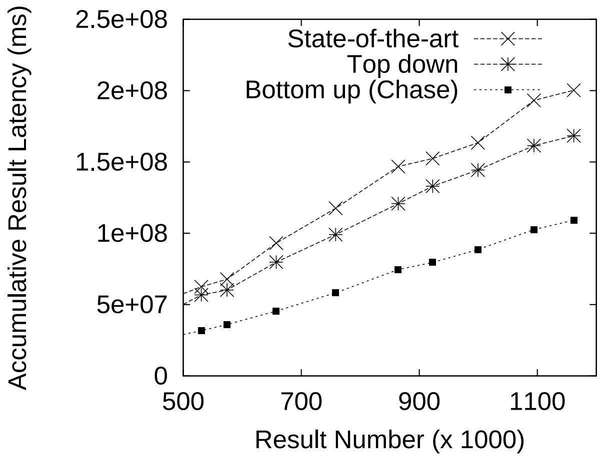


(b) Workload with only Pattern Changes

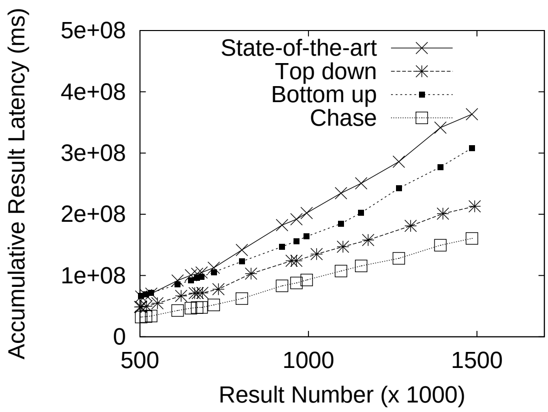


(c) Workload with only Concept Changes

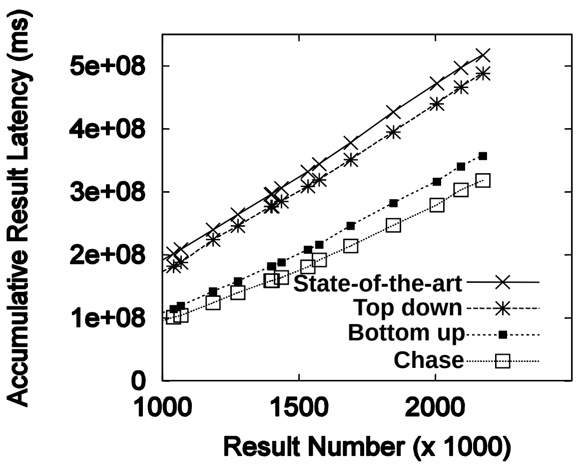
Figure 4.11: Controlled Workloads.



(a) Workload with only Concept Changes



(b) Workload 7

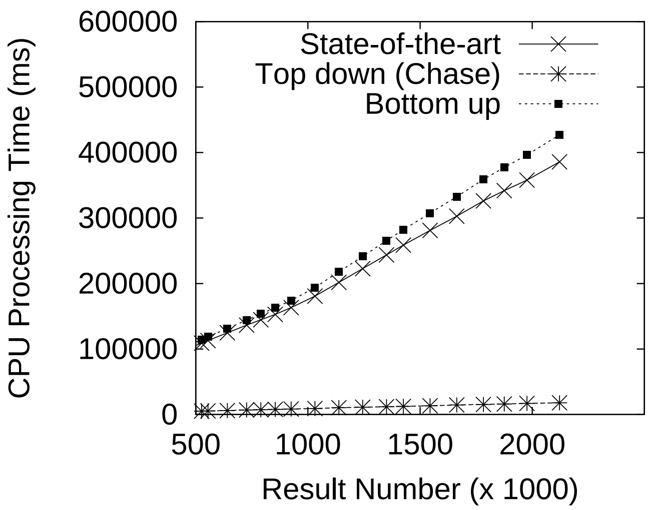


(c) Workload 8

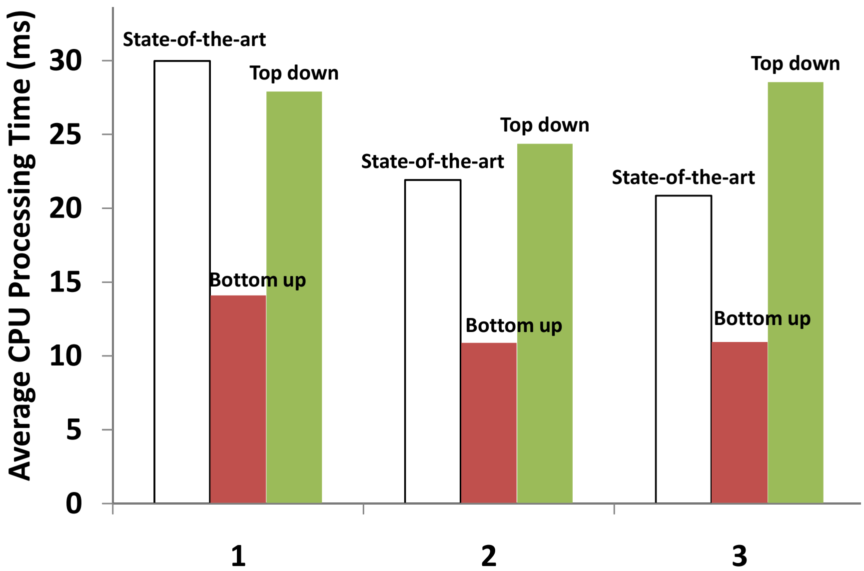
Figure 4.12: (a) Controlled Workload; (b)(c) Complex Query Workloads with Both Refinement Relationships.

Length	State-of-the-art	Top down (Chase)	Bottom up
3	0.28	0.04	0.31
4	7.77	1.2	7.87
5	384.8	17.75	426.57

(a) Workloads 1-3 with only Pattern Changes: Average CPU Processing Time (ms/result)

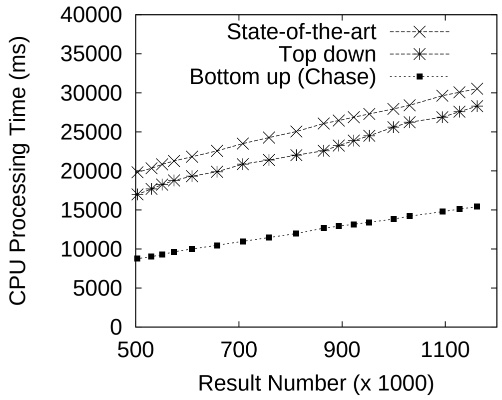


(b) Workload 2 with only Pattern Changes

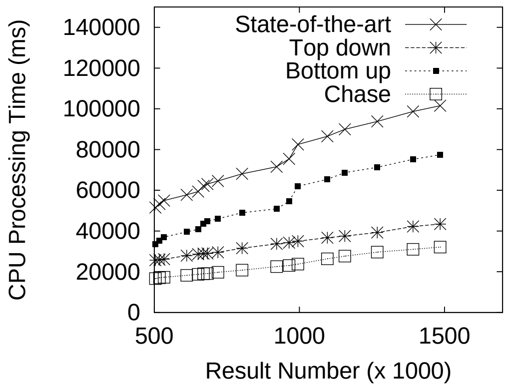


(c) Workloads 4-6 with only Concept Changes

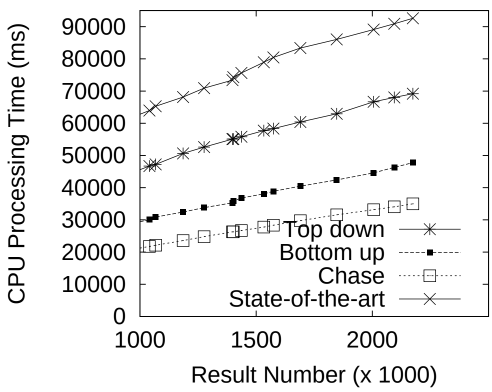
Figure 4.13: Controlled Workloads.



(a) Workload 4 with only Concept Changes



(b) Workload 7



(c) Workload 8

Figure 4.14: (a) Controlled Workload; (b)(c) Complex Query Workloads with Both Refinement Relationships.

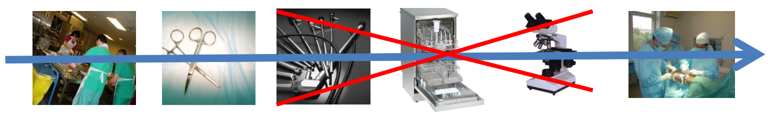
Chapter 5

High-performance Nested CEP Query Processing over Event Streams

The proposed techniques have been implemented and experimentally evaluated in an event processing system developed at WPI in collaboration with HP. This work has been published as one ICDE paper [LRG$^+$11c] and two workshop papers [LRR$^+$10, LRG$^+$10b].

5.1 Introduction

Complex event processing (CEP) has become increasingly important in modern applications ranging from supply chain management for RFID tracking to real-time intrusion detection [WDR06, BDG$^+$07, MM09]. CEP must be able to support sophisticated pattern matching on real time event streams including the arbitrary nesting of sequence (SEQ), AND, OR and the flexible use of negation in such nested patterns. For example, consider reporting contaminated medical equipments in a hospital [BP02, SrCL+05, TFR+09]. Let us assume that the tools for medical operations are RFID-tagged. The system monitors the histories of the equipment (such as, records of surgical usage, washing, sharpening and disinfection). When a healthcare worker puts a box of surgical tools into a surgical table equipped with RFID readers, the computer would display warnings such as "The tool must be disposed". Query Q_1 (Figure 5.1) expresses this critical condition that after being recycled and washed, a surgery tool is being put back into use without first being sharpened, disinfected and then checked for quality assurance. Such complex sequence queries may contain complex negation specifying the non-occurrence of composite subpatterns, such as negating the composite event of sharpened, disinfected and checked subsequences.



```
PATTERN SEQ(Recycle r, Washing w,
            ! SEQ(Sharpening s, Disinfection d, Checking c),
            Operating o, o.ins-type="surgery")
WITHIN 1 hour
```

Figure 5.1: Example Query Q_1

One of the most interesting and flexible features of a query language is the nesting of operators to an arbitrary depth [Kim82, MHM04]. Without this capability, users are severely restricted in forming complex patterns in a convenient and succinct manner. From certain point of view, the state-of-art CEP systems such as SASE [WDR06], ZStream [MM09] and Cayuga system [BDG+07] support nested queries as negation could be viewed as a special case of one-level deep nesting. However, these systems use two step execution method. Namely, the results satisfying the non-negation part are first constructed and then filtered if event instances which math the negation part exist. Such forced execution ordering misses optimization opportunities. SASE+ [ADGI08] extended from SASE supports Kleene Closure and provides a special syntax for allowing (i.e., skipping) irrelevant tuples in between those that match a given pattern. K*SQL [MZZ10] can express complex patterns on relational streams and sequences and can query data with more complex structures, e.g, XML and genomic data. However, they don't support applying negation over composite event types. While CEDR [BGAH07] allows applying negation over composite event types within their proposed language, the execution strategy for such nested queries is not discussed. A declarative query language LINQ [PR08] used in Microsoft StreamInsight [Ae09] allows nested queries by composing query templates. However, no optimization is introduced for processing negation over composite event types.

Without the design of an optimized execution strategy for nested sequence queries, an iterative nested execution strategy would typically be adopted by default [SPL96, LRR$^+$10, BKMH06]. Namely, first all component events matching the outer query are identified. In our example, we thus would compute all matching composite events consisting of SEQ(Recycle, Washing, Operating) subsequences. Thereafter, for each outer SEQ(Recycle, Washing, Operating) match, the results for the nested inner subsequences are iteratively computed, i.e., in this case, (Sharpening, Disinfection, Checking) subsequences. As last step, each outer candidate sequence result will be filtered by the non-existence of the inner subsequence match between the Washing reading and Operating reading. This process of first rigidly undertaking the construction of sequence results for the outer operators and then constructing sequence results for the inner operators is not efficient as it misses critical opportunities for optimization as illustrated below.

Problem 1: Candidate sequence results generated may later simply be discarded – thus wasting precious resources. For example in the above query Q_1, the generation of the sequence results for the outer subexpression SEQ(Recycle, Washing, Operating) may all be wasted as during normal medical procedures as inner sequences of type (Sharpening, Disinfection, Checking) would indeed exist between all event pairs of Washing and Operating. This unnecessary event generation to be later discarded wastes precious memory and CPU processing resources.

Problem 2: Full results satisfying the nested negated subexpression, such as instances that match the subsequence SEQ(Sharpening s, Disinfection d, Checking c) in Q_1 will be repeatedly constructed and processed for each outer candidate. However, knowing the existence of only one (Sharpening s, Disinfection d, Checking c) event between Washing and Operating events would be sufficient for filtering a candidate.

5.1.1 NEEL: The Nested Complex Event Language

We now briefly introduce the *NEEL*[1] query language for specifying complex nested event pattern queries as an extension of basic non-nested languages from the literature [WDR06, DGP$^+$07]. Its BackusNaur Form (BNF) syntax is shown in Table 3.1 while an example query using this syntax has been shown in the introduction, namely, query Q_1 in Figure 5.1. *NEEL* supports the nesting of AND, OR, Negation and SEQ at any level.

[1] NEEL stands for **Ne**sted Complex **E**vent Query **L**anguage.

<**Query**>::= PATTERN <generating exp>
WITHIN <window>
[RETURN <set of primitive events>]
<generating exp> ::=
SEQ(X, [<qual>])
| **AND**(X, [<qual>])
| **OR**((<generating exp>)$^+$, [<qual>])
| (<primitive-event type>, [<var>], [<qual>])
X ::= (boolean expression,)*, generating exp, query*
query ::= generating exp | boolean exp
boolean exp :: = **!** <generating exp> | $\exists$ <generating exp> | boolean exp $\vee$ boolean exp
<primitive-event type> ::= E_1 | E_2 | ...
<var> ::= event variable e_i
<qual>::= (<elemqual> ;)*
<elemqual> ::= <var>.attr <op> constant
<op> ::= $<$ | $>$ | $\leq$ | $\geq$ | $=$ | $! =$
<window>::= time duration w | tuple count c

Table 5.1: Event Expression for NEEL Query Language

Event expressions fall into two categories: generating and boolean expressions. Generating expressions return event histories and boolean expressions return boolean values (see Definition 12). The symbol "!" before an event expression *Exp* expresses the negation of *Exp* and indicates that *Exp* is not allowed to appear in the specified position [WDR06]. If *Exp* is a generating expression, ! *Exp* and $\exists$ *Exp* are boolean expressions. More precisely, it turns *Exp* into a boolean filter that checks if the result set returned by the sub-pattern preceded by ! is an empty set. The symbol "$\exists$" before an event expression *Exp* indicates that *Exp* must exist in the specified position.

Nested expressions. If *Exp* is an event expression, an application of SEQ, AND

and OR over *Exp* is again an event expression [CKAK94]. As shown below, Exp_i ,..., Exp_n are outer expressions of Exp_{i-1}. And Exp_1 ,..., Exp_{i-1} are inner expressions of Exp_i. Assume Exp_i = op(Exp_{i-1} ,..., E_j e_j, E_k e_k ,..., E_n e_n). The **variable scope** for primitive event instances such as e_j, e_k and e_n in Exp_i is within Exp_i and inner expressions of Exp_i.

```
Exp2 = op(Exp1 ,..., )
Exp3 = op(Exp2 ,..., )
...
Expi-1 = op(Expi-2 ,..., )
Expi = op(Expi-1, ,...,)
...
Expn = op(Expn-1 ,...,)

where op = SEQ, AND or OR;
```

Nested Boolean Expressions. A boolean expression *Exp* can be used as an inner expression to filter out the construction of an outer event expression. For example, in Q_2 the boolean expression *Disinfection* is a subexpression of the boolean expression ! SEQ(Sharpening s, ! Disinfection d, Checking c). The latter in turn is a subexpression of the outermost SEQ expression of Q_2. Q_2 states that $< r, w, o >$ is a valid match if either no *Sharpening* and *Checking* event pairs exist in the input stream between our *Washing* w and *Operating* o events in the outer match $< r, w, o >$, or otherwise if they do exist, then disinfection events must also exist between all *Sharpening* and *Checking* event pairs.

```
Q2 = PATTERN SEQ(Recycle r, Washing w,
```

```
! SEQ(Sharpening s, ! Disinfection d, Checking c),
Operating o)
```

Predicate Specification. The optional qualification [<qual>] in the PATTERN clause contains one or more predicates. In an expression *Exp*, we consider a simple predicate that only refers to a single event instance e_j of a primitive event type E_i in *Exp*. Simple predicates in an expression *Exp* are specified directly inside *Exp*. The treatment of join predicates are omitted. Join predicates on negation is ambiguous in semantics. Consider the query Q below. Q = PATTERN SEQ(Recycle r, ! Washing w, Operating o, r.attr1 + w.attr1 = o.attr1). It is not clear when the predicate r.attr1 + w.attr1 = o.attr1 is satisfied for a given event pair {r, o}, if we should return {r, o}. The reason we should return {r, o} is the predicate involving r and o are satisfied. However, we could not return {r, o} as a Washing event instance w with the specified predicate exists.

5.1.2 NEEL Semantics

Event History with Basic Operations

Definition 10 *Event history H is an ordered set of primitive event instances. Time constraint event history H[ts, te] is an ordered set of primitive event instances from history H with timestamps less than te and greater than ts.*

$$H[ts,te] = \{e|\forall e \in H \wedge (ts \leq e.ts \leq e.te \leq te)\}. \tag{5.1}$$

Assume the window size for an event expression is w. For sliding window *semantics, at any time t, we apply a query to the window constraint event history $H_w = H[ts, te]$ with te := t and ts := t - w where w is an integer representing the sliding window size.*

Definition 11 *$E_i[H_w]$ selects events of type E_i from window constrained event history H_w.*

$$E_i[H_w] \;=\; \{e | e \in H_w \wedge (e \in E_i)\}. \tag{5.2}$$

Notations

1). The notation $\overrightarrow{e_{1,n}}$ denotes an ordered sequence of event instances e_1, e_2, ... , e_n such that for all pairs (e_i, e_j) with $i < j$ in the sequence, e_i.ts $\leq e_i$.te $<$ e_j.ts $\leq e_j$.te holds.

2). The notation $set_{of}(e_{1,n})$ denotes the set $\{e_1, ..., e_n\}$.

3). The notation $set_{of}(\overrightarrow{e_{1,n}})$ denotes the set $\{e_1, ..., e_n\}$ with e_1.ts $\leq e_1$.te $<$ ... $<$ e_n.ts $\leq e_n$.te.

4). The notation $\Pi E_{1,n}$ denotes the cross product of event histories from E_1 to E_n. Namely, $\Pi E_{1,n}[H_w] = E_1[H_w] \times E_2[H_w] \times ... E_i[H_w] \times \ldots \times E_n[H_w]$.

5). We use the notation $<P_1(e_1)$, ... , $P_n(e_n)>$ to refer to a set of simple predicates applied to event instances $e_1, \ldots, e_n$ respectively. For ease of use, we use $\mathcal{P}$ as a shorthand for $<P_1(e_1)$, ... , $P_n(e_n)>$.

Operator Semantics

Definition 12 *Generating expressions return event histories while boolean expressions return boolean values. ! Exp[H_w] = T iff Exp[H_w] = $\emptyset$. $\exists$ Exp[H_w] = T iff Exp[H_w] $\neq \emptyset$.*

Definition 13 *[**SEQ operator**]. SEQ specifies a particular order in which the event instances of interest e_1, e_2 ,..., e_n must occur in order to correspond to a valid match. The event instances that satisfy specified time ordering and predicates are returned. $\Pi E_{1,n}$[H_w] and $\mathcal{P}$ are denoted in Section 5.1.2. The meaning of a SEQ expression (with boolean expressions) can be defined recursively in terms of the meanings of the subexpressions. Namely, in Equation 5.3 below, for 1 $<$ i $<$ n, E_i is a primitive event type.*

$$
\begin{aligned}
&SEQ(E_1\ e_1, E_2\ e_2, ..., E_i\ e_i, \ldots, E_n\ e_n, \mathcal{P})[H_w] \\
&= \{set_{of}(\overrightarrow{e_{1,n}}) | (\overrightarrow{e_{1,n}} \in \Pi E_{1,n}[H_w]) \wedge (\mathcal{P} == true)\}.
\end{aligned} \tag{5.3}
$$

Example 17 *Given SEQ(Recycle r, Washing w) and $H_3 = \{r_1, w_2, w_3\}$, SEQ(Recycle r, Washing w)[H_3] generates 2 event histories: $\{r_1, w_2\}$ and $\{r_1, w_3\}$.*

Definition 14 **SEQ with Negation !.** *Equation 5.4 below defines the SEQ operator with negation in the middle of a list of event types. We first identify $\{e_1$,..., e_i, e_{i+1} ,..., $e_n\}$ matching the generating event expression satisfying associated predicates. We then verify the non-existence of X instances between e_i and e_{i+1} events.*

$$SEQ(E_1\ e_1,...,E_i\ e_i,!X,E_{i+1}\ e_{i+1},...,E_n\ e_n,\mathcal{P})[H_w]$$
$$= \{set_{of}(\overrightarrow{e_{1,n}})|\overrightarrow{e_{1,n}} \in (\Pi E_{1,n}[H_w]) \wedge (\mathcal{P} == true) \tag{5.4}$$
$$\wedge X[H[e_i.te, e_{i+1}.ts]] = \emptyset\}.$$

SEQ(E_1 e_1 ,..., E_i e_i, ! X , E_{i+1} e_{i+1} ,..., E_n e_n, $\mathcal{P}$)[H_w] is the set of all those sequences $\{e_1$,..., e_i, e_{i+1} ,..., $e_n\}$ such that

(i) The time ordered event set $\{e_1$,..., e_i, e_{i+1} ,..., $e_n\}$ is in SEQ(E_1 e1 ,..., E_i e_i, E_{i+1} e_{i+1} ,..., E_n e_n , $\mathcal{P}$)[H_w], and

(ii) X [H'] is empty, where H' is the sub-history of [H_w] determined by the end-time of e_i and the start time of e_{i+1} if E_i and E_{i+1} are positive primitive event types. Otherwise, the left bound of H' is determined by the end-time of the event instance of the first positive event type from E_i, E_{i-1} ,..., to E_1. If E_i, E_{i-1} ,..., and E_1 are all negative, the left bound of H' is the same as the left bound of H_w. Similarly, the right bound of H' is determined by the start-time of the event instance of the first positive event type from E_{i+1}, E_{i+2} ,..., to E_n. If E_{i+1}, E_{i+2} ,..., and E_n are all negative, the right bound of H' is the same as the right bound of H_w.

Multiple negations could exist inside a SEQ. Negation could equally exist at the start or the end of the SEQ operator. Given a H_w, if negation exists at the start, the non-existence left time bound would be min($e_n.te - w$, H_w.ts). Similarly, if negation exists at the end, the non-existence right time bound would be max($e_1.ts +$ w, H_w.te). If negations are specified at both the start and the end of the SEQ operator, no negation match exists in either scopes of size w. Namely, the non-existence left time bound would be min($e_n.te - w$, H_w.ts) and the right time bound *would be max($e_1.ts + w$, $H_w.te$).*

If the specified events of the boolean expression ! E don't exist in the stream at the specified location, then we find a match for the event expression with negation(s). Multiple boolean expression ! E could also be specified in the SEQ operator. For example SEQ(Washing w, ! (Sharpening s, s.id = 1), Disinfection d, ! (Checking c, c.id = 2)).

Definition 15 SEQ with Exists $\exists$. *Equation 5.5 defines the SEQ operator with $\exists$ before event expressions. We first identify $\{e_1$,..., e_i, e_{i+1} ,..., $e_n\}$ matching the generating event expression satisfying associated predicates. We then verify the existence of X instances between e_i and e_{i+1} events of each candidate match history.*

$$\begin{aligned}&SEQ(E_1\ e_1,\ldots,E_i\ e_i,\exists X,E_{i+1}\ e_{i+1},\ldots,E_n\ e_n,\mathcal{P})[H_w]\\&\quad=\{set_{of}(\overrightarrow{e_{1,n}})|\overrightarrow{e_{1,n}}\in\Pi E_{1,n}[H_w]\wedge(\mathcal{P}==true)\wedge X[H[e_i.te,e_{i+i}.ts]]\neq\emptyset\}.\end{aligned}\tag{5.5}$$

SEQ(E_1 e_1 ,..., E_i e_i, $\exists$ X , E_{i+1} e_{i+1} ,..., E_n e_n, $\mathcal{P}$)[H_w] are the sets $\{e_1$,..., e_i, e_{i+1} ,..., $e_n\}$ such that

(i) The time ordered event instance set $\{e_1$,..., e_i, e_{i+1} ,..., $e_n\}$ is in SEQ(E_1 e1 ,..., E_i e_i, E_{i+1} e_{i+1} ,..., E_n e_n, $\mathcal{P}$)[H_w], and

(ii) X [H'] is not empty, where H' is the sub-history of [H_w] determined by the end-time of e_i and the start time of e_{i+1} if E_i and E_{i+1} are positive primitive event types. Otherwise, the left bound of H' is determined by the end-time of the event instance of the first positive event type from E_i, E_{i-1} ,..., to E_1. If E_i, E_{i-1} ,..., and E_1

are all negative, the left bound of H' is the same as the left bound of H_w. Similarly, the right bound of H' is determined by the start-time of the event instance of the first positive event type from E_{i+1}, E_{i+2} ,..., to E_n. If E_{i+1}, E_{i+1} ,..., and E_n are all negative, the right bound of H' is the same as the right bound of H_w.

Definition 16 *[***AND operator***]. We don't require event timestamp ordering among e_1, e_2 ,..., e_n in $\{e_1, e_2, ..., e_n\}$ in Equation 5.6. The meaning of a AND expression (with boolean expressions) can be defined recursively in terms of the meanings of the subexpressions. Namely, in Equation 5.6 below, for 1 $<$ i $<$ n, E_i is a primitive event type.*

$$\begin{aligned} &AND(E_1\ e_1, E_2\ e_2, ...E_n\ e_n, \mathcal{P})[H_w] \\ &\quad = \{set_{of}(e_{1,n}) | (set_{of}(e_{1,n}) \in \Pi E_{1,n}[H_w]) \wedge (\mathcal{P} == true)\}. \end{aligned} \tag{5.6}$$

Example 18 *Given AND(Recycle r, Washing w) and the partial input stream $\{w_1$, r_2, $w_3\}$ within the window. Then $\{\{r_2, w_1\}, \{r_2, w_3\}\}$ is generated.*

Definition 17 AND with Negation ! *Equation 5.7 defines the AND operator with negation. Negation ! X works like a filter. Each AND candidate result is returned if X[H_w] = $\emptyset$.*

$$\begin{aligned} &AND(E_1\ e_1, ..., E_i\ e_i, !X, E_{i+1}\ e_{i+1}, ..., E_n\ e_n, \mathcal{P}\})[H_w] \\ &\quad = \{set_{of}(e_{1,n}) | set_{of}(e_{1,n}) \in \Pi E_{1,n}[H_w] \wedge (\mathcal{P} == true) \wedge X[H_w] = \emptyset\}. \end{aligned} \tag{5.7}$$

AND(E_1 e_1 ,..., E_i e_i, $\exists$ X , E_{i+1} e_{i+1} ,..., E_n e_n, $\mathcal{P}$)[H_w] is the set of all those

$\{e_1$,..., e_i, e_{i+1} ,..., $e_n\}$ *such that*

(i) $\{e_1$,..., e_i, e_{i+1} ,..., $e_n\}$ *is in AND(*E_1 e_1 ,..., E_i e_i, E_{i+1} e_{i+1} ,..., E_n e_n , $\mathcal{P}$*)[*H_w*], and*

(ii) X [H_w*] is nonempty,*

Multiple negation could exist in AND. Positions of ! E in AND doesn't matter. AND operator must contain at least one positive expression.

Example 19 *Given AND(Recycle r, Washing w, ! Checking c) and the partial input stream* $\{c_1, w_2, r_3\}$*, no results are generated due to the existence of the event* $c_1 \in$ *Checking within the window constraint history.*

Definition 18 AND with Exists $\exists$**.** *Equation 5.8 defines the AND operator with* $\exists$. $\exists$ *X works like a filter. Each AND candidate result is returned if X[*H_w*] is not empty.*

$$\begin{aligned} &AND(E_1\ e_1,\ldots,E_i\ e_i,\exists X,E_{i+1}\ e_{i+1},\ldots,E_n\ e_n,\mathcal{P})[H_w] \\ &\quad = \{set_{of}(e_{1,n})|set_{of}(e_{1,n}) \in \Pi E_{1,n}[H_w] \wedge (\mathcal{P} == true) \wedge X[H_w] \neq \emptyset\}. \end{aligned} \tag{5.8}$$

AND(E_1 e_1 ,..., E_i e_i, $\exists$ *X ,* E_{i+1} e_{i+1} ,..., E_n e_n , $\mathcal{P}$*)[*H_w*] is the set of all those* $\{e_1$,..., e_i, e_{i+1} ,..., $e_n\}$ *such that*

(i) $\{e_1$,..., e_i, e_{i+1} ,..., $e_n\}$ *is in AND(*E_1 e_1 ,..., E_i e_i, E_{i+1} e_{i+1} ,..., E_n e_n , $\mathcal{P}$*)[*H_w*], and*

(ii) X [H_w*] is nonempty.*

Definition 19 *[OR operator]. Formally, the set-operator OR is defined as follows. An event history is returned for the OR operator.*

$$
\begin{aligned}
& OR(E_1\ e_1\ ,...,E_n\ e_n,\mathcal{P})[H_w] \\
& \quad = \{\{e_1\}|\{e_1\} \in E_1[H_w] \wedge (P_1(e_1) == true)\} \cup ... \cup \\
& \quad \{\{e_n\}|\{e_n\} \in E_n[H_w] \wedge (P_n(e_n) == true)\}
\end{aligned} \tag{5.9}
$$

OR with Boolean Expressions.

Boolean expressions including ! E and $\exists$ E are not allowed in the OR operator as OR connects generating expressions.

Example 20 *Assume that the query Q_2 = OR(Checking, Sharpening, Checking.insType = "scalpels"; Sharpening.insID = 15)[H_4]. The event history H = $\{c_1, c_2, c_6, s_8\}$ where c_1.insType = "forceps", c_2.insType = "scalpels", c_6.insType = "scalpels" and s_8.insID = 15. Then Q_2 returns a result history $\{\{c_6\}, \{s_8\}\}$.*

5.1.3 Nested CEP Query Plan Generation

A query expressed by a *NEEL* specification is one-to-one translated into a default nested algebraic query plan composed of the following algebraic operators: Window Sequence (*WinSeq*), Window And (*WinAnd*) and Window Or (*WinOr*). The same window *w* is as default applied to all operator nodes. During query transformation, each expression in the event pattern is mapped to one operator node in the query plan. For queries expressed by NEEL, predicates are placed into the positions as already specified by the NEEL expressions.

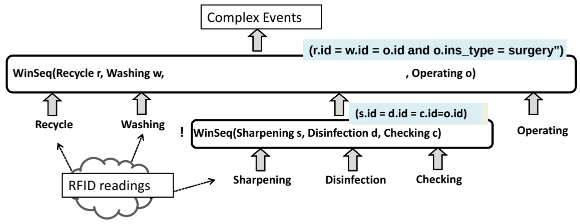


Figure 5.2: Basic Query Plan for Query Q_1 in Figure 5.1

5.1.4 Nested CEP Query Execution

Traditional Execution Strategy. Following the principle of top down iterative nested query execution for nested SQL queries [SC75], the outer query is evaluated first as context followed by its inner sub-queries. For every outer partial query result, a constrained window is passed down for processing each of its children sub-queries. These sub-queries compute results involving events within the constrained window. Qualified result sequences of the inner operators are passed up to the parent operator and the outer operator then joins its own local results with that of its generating sub-expressions. The outer sequence result is filtered if the result set of any of its boolean expressions ! E is not empty or the results of a boolean $\exists$ sub-query is empty. Finally, the process repeats when the outer query consumes the next instance e. We omit the detailed discussion and examples for nested queries with negation and predicates. Please refer to [LRR$^+$10] for details.

Discussion. Such nested query evaluation methodology suffers from several inefficiencies. For Q_1 in Figure 5.2, first, candidate results of SEQ(Recycle r, Washing w, Operating o) initially generated may later be discarded. Another potential performance waste is that full results for the inner boolean expression SEQ(Sharpening s, Disinfection d, Checking c) are constructed. These cases were also highlighted in problems 1 and 2 in the introduction. The just introduced nested query evaluation does not solve these problems. To overcome such inefficiencies, in Section 5.2, we will explore query rewriting techniques to flatten and optimize nested CEP expressions.

	Rule
FR	(1) $\mathbf{SEQ}(\mathbf{SEQ}(E_1\ e_1\ ,..., E_i\ e_i, \mathcal{P}), E_j\ e_j\ ,..., E_n\ e_n)$ $= \mathbf{SEQ}(E_1\ e_1\ ,..., E_i\ e_i\ ,..., E_n\ e_n, \mathcal{P})$. (2) $\mathbf{SEQ}(\mathbf{SEQ}(E_1\ e_1\ ,..., \exists\ (E_{i-1}\ e_{i-1}), E_j\ e_j, \mathcal{P}), E_{j+1}\ e_{j+1}\ ,..., E_n\ e_n)$ $= \mathbf{SEQ}(E_1\ e_1\ ,..., \exists\ (E_{i-1}\ e_{i-1}), E_j\ e_j\ ,..., E_n\ e_n, \mathcal{P})$ (3) $\mathbf{AND}(\mathbf{AND}(E_1\ e_1\ ,..., E_j\ e_j, \mathcal{P}), E_{j+1}\ e_{j+1}\ ,..., E_n\ e_n)$ $= \mathbf{AND}(E_1\ e_1\ ,..., E_j\ e_j\ ,..., E_n\ e_n, \mathcal{P})$. (4) $\mathbf{AND}(\mathbf{AND}(E_1\ e_1\ ,..., \mathbf{!}\ (E_i\ e_i, P_i(e_i)), E_j\ e_j, \mathcal{P}), E_{j+1}\ e_{j+1}\ ,..., E_n\ e_n)$ $= \mathbf{AND}(E_1\ e_1\ ,..., !\ (E_i\ e_i, P_i(e_i))\ E_j\ e_j,..., E_n\ e_n, \mathcal{P})$. (5) $\mathbf{OR}(\mathbf{OR}(E_1\ e_1\ ,..., E_i\ e_i), E_j\ e_j\ ,..., E_n\ e_n)$ $= \mathbf{OR}(E_1\ e_1\ ,..., E_i\ e_i, E_j\ e_j\ ,..., E_n\ e_n)$ (6) $\mathbf{SEQ}(\exists\ \mathbf{SEQ}(E_1\ e_1\ ,..., E_i\ e_i, \mathcal{P}), E_j\ e_j\ ,..., E_n\ e_n)$ $= \mathbf{SEQ}(\exists\ (E_1\ e_1)\ ,..., \exists\ (E_i\ e_i), E_j\ e_j\ ,..., E_n\ e_n, \mathcal{P})$ (7) $\mathbf{AND}(\exists\ \mathbf{AND}(E_1\ e_1\ ,..., E_i\ e_i, \mathcal{P}), E_j\ e_j\ ,..., E_n\ e_n)$ $= \mathbf{AND}(\exists\ (E_1\ e_1)\ ,..., \exists\ (E_i\ e_i), E_j\ e_j\ ,..., E_n\ e_n, \mathcal{P})$

Table 5.2: Rewriting Rules: FR(Flattening Rule)

	Rule
DR	(1) **SEQ**(E_1 e_1, **OR**(E_2 e_2 ,..., E_i e_i, $\mathcal{P}$), E_j e_j ... E_n e_n) = **OR**(**SEQ**(E_1 e_1, E_2 e_2, E_j e_j ,..., E_n e_n, $P_2(e_2)$)[H_w] , ... , **SEQ**(E_1 e_1, E_i e_i, E_j e_j ,..., E_n e_n, $P_i(e_i)$)) (2) **AND**(E_1 e_1, **OR**(E_2 e_2 ,..., E_i e_i, $\mathcal{P}$) ,..., E_n e_n) = **OR**(**AND**(E_1 e_1, E_2 e_2 ,..., E_n e_n, $P_2(e_2)$) ,..., **AND**(E_1 e_1, E_i e_i ,..., E_n e_n, $P_i(e_i)$)) (3) **SEQ**(E_1 e_1, $\exists$ E_2 e_2 $\vee$,..., $\vee$ $\exists$ E_i e_i, E_j e_j ... E_n e_n) = **OR**(**SEQ**(E_1 e_1, $\exists$ E_2 e_2, E_j e_j ,..., E_n e_n) ,..., **SEQ**(E_1 e_1, $\exists$ E_i e_i, E_j e_j ,..., E_n e_n)) (4) **AND**(E_1 e_1, $\exists$ E_2 e_2 $\vee$,..., $\vee$ $\exists$ E_i e_i ,..., E_n e_n) = **OR**(**AND**(E_1 e_1, $\exists$ E_2 e_2 ,..., E_n e_n) ,..., **AND**(E_1 e_1, $\exists$ E_i e_i ,..., E_n e_n)) (5) **SEQ**($\exists$ **OR**(E_1 e_1 ,..., E_i e_i), E_j e_j ,..., E_n e_n) = **OR**(**SEQ**($\exists$ (E_1 e_1), E_j e_j ,..., E_n e_n) ,..., **SEQ**($\exists$ (E_i e_i), E_j e_j ,..., E_n e_n)) (6) **AND**($\exists$ **OR**(E_1 e_1 ,..., E_i e_i), E_j e_j ,..., E_n e_n) = **OR**(**AND**($\exists$ (E_1 e_1), E_j e_j ,..., E_n e_n) ,..., **AND**($\exists$ (E_i e_i), E_j e_j ,..., E_n e_n))

Table 5.3: Rewriting Rules: DR(Distributive Rule)

	Rule
NPDR	(1) **! SEQ**(E_1 e_1 ,..., E_{i-1} e_{i-1}, E_i e_i) (right-to-left unroll) = ! (E_i e_i) $\vee$ $\exists$ SEQ(! SEQ(E_1 e_1 ,..., E_{i-1} e_{i-1}), E_i e_{i1}, ! (E_i e_{i2})) (2) **! SEQ**(E_1 e_1, E_2 e_2 ,..., E_i e_i) (left-to-right unroll) = ! (E_1 e_1) $\vee$ $\exists$ SEQ(! (E_1 e_{11}), E_1 e_{12}, ! (SEQ(E_2 e_2 ,..., E_i e_i))) (3) **! AND**(E_1 e_1 ,..., E_i e_i, $\mathcal{P}$) = ! (E_1 e_1, $P_1(e_1)$) $\vee$ ... $\vee$! (E_i e_i, $P_i(e_i)$) (4) **! AND**(E_1 e_1 ,..., **!** (E_i e_i, $P_i(e_i)$) ,..., E_j e_j, $\mathcal{P}$) = ! (E_1 e_1, $P_1(e_1)$) $\vee$ ... $\vee$ $\exists$ (E_i e_i, $P_i(e_i)$) ... $\vee$! (E_j e_j, $P_j(e_j)$) (5) **! OR**(E_1 e_1 ,..., E_i e_i, $\mathcal{P}$) = ! (E_1 e_1, $P_1(e_1)$) $\wedge$ ... $\wedge$! (E_i e_i, $P_i(e_i)$) (6) **! SEQ**(E_1 e_1 ,..., $\exists$ (E_i e_i) ,..., E_n e_n, $\mathcal{P}$) = **! SEQ**(E_1 e_1 ,..., E_i e_i ,..., E_n e_n, $\mathcal{P}$) (7) **! AND**(E_1 e_1 ,..., $\exists$ (E_i e_i) ,..., E_n e_n, $\mathcal{P}$) = **! AND**(E_1 e_1 ,..., E_i e_i ,..., E_n e_n, $\mathcal{P}$)

Table 5.4: Rewriting Rules: NPDR(Negation Push Down Rule)

<table>
<tr><td><Query>::= PATTERN <generating exp>
WITHIN <window>
[RETURN <set of primitive events>]
<generating exp> ::=
<generating exp in SEQ>
| <generating exp in AND>
| <generating exp in OR></td></tr>
<tr><td>XO = <generating exp in SEQ> ::= SEQ(XI, [<qual>])
| OR((<generating exp in SEQ>)$^+$, [<qual>])
| (<primitive-event type>, [<var>], [<qual>])
XI ::= (boolean expression in SEQ,)*, generating exp in SEQ, query in SEQ*
query in SEQ ::= generating exp in SEQ | boolean exp in SEQ
boolean exp in SEQ :: =
! <generating exp in SEQ> if XO and outer expressions of XO are not of the form ! XO
| $\exists$ <generating exp in SEQ>
| boolean exp in SEQ$\vee$ boolean exp in SEQ</td></tr>
<tr><td><generating exp in AND> ::=
| AND(Y, [<qual>])
| AND(SEQ(XO, [<qual>]))
| OR((<generating exp in AND>)$^+$, [<qual>])
| (<primitive-event type>, [<var>], [<qual>])
Y ::= (boolean expression in AND,)*, generating exp in AND, query in AND*
query in AND ::= generating exp in AND | boolean exp in AND
boolean exp in AND :: = ! <generating exp in AND>
| $\exists$ <generating exp in AND>
| boolean exp in AND$\vee$ boolean exp in AND</td></tr>
<tr><td><generating exp in OR> ::=
SEQ(XI, [<qual>])
| AND(Y, [<qual>])
| OR((<generating exp in OR>)$^+$, [<qual>])
| (<primitive-event type>, [<var>], [<qual>])
Z ::= (boolean expression in OR,)*, generating exp in OR, query in OR*
query in OR ::= generating exp in OR | boolean exp in OR
boolean exp in OR :: = ! <generating exp in OR>
| $\exists$ <generating exp in OR>
| boolean exp in OR$\vee$ boolean exp in OR</td></tr>
<tr><td><primitive-event type> ::= E_1 | E_2 | ...
<var> ::= event variable e_i
<qual>::= (<elemqual> ;)*
<elemqual> ::= <var>.attr <op> constant
<op> ::= $<$ | $>$ | $\leq$ | $\geq$ | $=$ | ! $=$
<window>::= time duration w | tuple count c</td></tr>
</table>

Table 5.5: Event Expression for Class Lcons

5.2 NEEL Event Expression Rewriting

Our system can process all queries expressed by *NEEL* in Section 5.1.1 [LRR+10]. But only some subset satisfying our language constraints described in Section 5.2.2 can be optimized using our rewriting techniques presented below. NEEL logical query optimizer needs to analyze if optimization is applicable. By flattening a nested *NEEL* expression, we could avoid the problem of forced execution ordering described in Section 5.1.

5.2.1 Event Expression Rewriting Rules

Our proposed rewriting rules fall into three categories: flattening rules, distributive rules and negation push down rules. Tables 3.2, 3.3 and 3.4 list our proposed *NEEL* rewriting rules for nested CEP expressions. Two expressions connected by "=" generate the same results. Namely, generating expressions return the same event history under any possible event history input and boolean expressions evaluate to the same boolean value.

5.2.2 Language Constraints

The rewriting system is only defined over some Class Lcons of expressions defined in Table 3.5. Theorem 3 in Section 5.2.8 proves that Class LC is closed under rewriting.

Class Lcons Design Decision.

- When an outer expression is SEQ, SEQ(∃ AND) and SEQ(AND) don't belong to Class Lcons. When an outer expression is AND, AND(∃ SEQ) and AND(! SEQ) don't belong to Class Lcons. It is because AND operator can't always be expressed by SEQ operator. Namely, AND(Exp_1, Exp_2)[H_w] != SEQ(Exp_1, Exp_2) $\vee$ SEQ(Exp_2, Exp_1)[H_w]. The SEQ operator requires strict time ordering among Exp_1 and Exp_2 instances. Hence, it misses several cases such as overlapping intervals among Exp_1 and Exp_2 instances which are captured by AND operator. Class Lcons containing SEQ(SEQ), SEQ($\exists$ SEQ), AND(AND), AND($\exists$ AND) or OR(OR) can be rewritten by the flattening rules.

- Lcons doesn't contain double negation on SEQ, !SEQ(!). It is because under our nested CEP model, we don't have an operator to support "for all" semantics. For example, Given input $\{a_1, b_2, d_4, c_6, d_8, e_{10}\}$. Assume q_k = SEQ(A a, ! SEQ(B b, ! (C c), D d), E e). q_k will return $\{a_1, e_{10}\}$ if All $\{b_i, d_j\}$ pairs with 1< i< j < 10 have C instances in between. $\{b_2, d_4\}$ has no C instances in between. q_k will not return $\{a_1, e_{10}\}$.

5.2.3 Flattening rules

The inner SEQ, AND or OR subexpression is merged into the outer SEQ, AND or OR expression respectively.

Rule 1 *After applying FR1, the nested SEQ(SEQ()) is equivalent to SEQ().*

$$\begin{aligned} &SEQ(SEQ(E_1\ e_1, ..., E_i\ e_i, \mathcal{P}), E_j\ e_j, ..., E_n\ e_n) \\ &= SEQ(E_1\ e_1, ..., E_i\ e_i, ..., E_n\ e_n, \mathcal{P}). \end{aligned} \tag{5.10}$$

Proof:

Assume Exp_{inner} = SEQ(E_1 e_1 ,..., E_i e_i, $\mathcal{P}$) and the event instance matching Exp_{inner} is e. Using Equation 5.3 (Definition of SEQ), the left hand side of Equation 5.10 can be written as

$$\begin{aligned}
&SEQ(SEQ(E_1\ e_1,...,E_i\ e_i,\mathcal{P}),E_j\ e_j,...,E_n\ e_n)[H_w] \\
&= \{\{e,e_j,...,e_n\}|\{e,e_j,...,e_n\} \in SEQ(E_1\ e_1,...,E_i\ e_i,\mathcal{P})[H_w] \times E_j[H_w] \\
&\times ... \times E_n[H_w] \wedge (e_i.te < e_j.ts < ... < e_n.ts)\}
\end{aligned} \tag{5.11}$$

Let the inner expression be denoted as Exp_{inner} = SEQ(E_1 e_1 ,..., E_i e_i, $\mathcal{P}$). Using Equation 5.3 (Definition of SEQ), we can write Exp_{inner} as

$$\begin{aligned}
&SEQ(E_1\ e_1,...,E_i\ e_i,\mathcal{P})[H_w] \\
&= \{set_{of}(\overrightarrow{e_{1,i}})|(\overrightarrow{e_{1,i}} \in \Pi E_{1,i}[H_w]) \wedge (\mathcal{P} == true)\}.
\end{aligned} \tag{5.12}$$

According to Equation 5.12, the event instance e matching Exp_{inner} can be expressed by $\{e_1,...,e_i\}$ and these events are ordered ($\overrightarrow{e_{1,i}}$). Thus for the right hand of Equation 5.11, we have

$$\begin{aligned}
&\{\{e,e_j,...,e_n\}|\{e,e_j,...,e_n\} \in SEQ(E_1\ e_1,...,E_i\ e_i,\mathcal{P})[H_w] \times E_j[H_w] \\
&\times ... \times E_n[H_w] \wedge (e.te < e_j.ts... < e_n.ts)\} \\
&= \{\{e_1,...,e_i,e_j,...,e_n\}|\{e_1,...,e_i,e_j,...,e_n\} \in SEQ(E_1\ e_1,...,E_i\ e_i, \\
&\mathcal{P})[H_w] \times E_j[H_w] \times ... \times E_n[H_w] \wedge (e_i.te < e_j.ts... < e_n.ts)\}
\end{aligned} \tag{5.13}$$

The subexpression SEQ(E_1 e_1 ,..., E_i e_i, $\mathcal{P}$)[H_w] can be substituted by the right hand side of Equation 5.12. According to Equation 5.2 (Definition of $E_i[H_w]$),$E_i[H_w]$

can be substituted by $\{e_i|e_i \in E_i[H_w]\}$. For the right hand side of Equation 5.13, we have

$$\begin{aligned}
&\{\{e_1,...,e_i,e_j,...,e_n\}|\{e_1,...,e_i,e_j,...,e_n\} \in SEQ(E_1\ e_1,...,E_i\ e_i,\mathcal{P})[H_w] \\
&\times E_j[H_w] \times ... \times E_n[H_w] \wedge (e_i.te < e_j.ts... < e_n.ts)\} \\
&= \{\{e_1,...,e_i,e_j,...,e_n\}|\{e_1,...,e_i,e_j,...,e_n\} \in \{\{e_1,...,e_i\}|(\{e_1,...,e_i\} \in \Pi E_{1,i}[H_w])\wedge \\
&(\mathcal{P} == true)\} \times \{e_j|e_j \in E_j[H_w]\} \times ... \times \{e_n|e_n \in E_n[H_w]\} \wedge (e_i.te < e_j.ts... < e_n.ts)\}
\end{aligned} \tag{5.14}$$

According to Cross Product, for the right hand side of Equation 5.14, we have

$$\begin{aligned}
&\{set_{of}(\overrightarrow{e_{1,n}})|\overrightarrow{e_{1,n}} \in \{set_{of}(\overrightarrow{e_{1,i}})|(\overrightarrow{e_{1,i}} \in \Pi E_{1,i}[H_w]) \wedge (\mathcal{P} == true)\}\times \\
&\{e_j|e_j \in E_j[H_w]\} \times ... \times \{e_n|e_n \in E_n[H_w]\} \wedge (e_i.te < e_j.ts... < e_n.ts)\} \\
&= \{set_{of}(\overrightarrow{e_{1,n}})|\overrightarrow{e_{1,n}} \in \Pi E_{1,i}[H_w] \times E_j[H_w] \times ... \times E_n[H_w] \wedge (\mathcal{P} == true)\} \\
&= \{set_{of}(\overrightarrow{e_{1,n}})|\overrightarrow{e_{1,n}} \in \Pi E_{1,n}[H_w] \wedge (\mathcal{P} == true)\}
\end{aligned} \tag{5.15}$$

For the right hand side of Equation 5.10, using Equation 5.3 (Definition of SEQ), we can write it as

$$\begin{aligned}
&SEQ(E_1\ e_1,...,E_i\ e_i,E_j\ e_j,...,E_n\ e_n,\mathcal{P})[H_w] \\
&\quad = \{set_{of}(\overrightarrow{e_{1,n}})|\{\overrightarrow{e_{1,n}}\} \in \Pi E_{1,n}[H_w] \wedge (\mathcal{P} == true)\}.
\end{aligned} \tag{5.16}$$

So the expressions on the left side of Equation 5.10 as now defined in Equation 5.15 and the right side of Equation 5.10 as now defined in Equation 5.16 are equivalent. The position of the inner subexpression doesn't affect the application of the flattening rule FR1. □

Rule 2 *After applying FR2, SEQ(SEQ(!)) is equivalent to SEQ(!).*

$$\begin{aligned} &SEQ(SEQ(E_1\ e_1,...,!(E_i\ e_i,P_i(e_i)),E_j\ e_j),E_{j+1}\ e_{j+1},...,E_n\ e_n) \\ &\quad = SEQ(E_1\ e_1,...,!(E_i\ e_i,P_i(e_i)),E_j\ e_j,...,E_n\ e_n) \end{aligned} \tag{5.17}$$

We omit the proof for Rule FR2 as it is similar to FR1.

Discussion. Flattening Rule 2 still holds if ! E_i or $\exists$ E_i exists at the end of the inner sub-expression or if ! E_j $\exists$ E_i exists at the start of the outer sub-expression. For example, for SEQ(A a, SEQ(B b, C c, !D d), E e, F f), the inner subexpression SEQ(B b, C c, !D d) is bounded by A and E instances in the outer expression which is not changed after rewriting. Similarly, the D instance is bounded by C and E instances which is not changed after rewriting. Also for SEQ(A a, SEQ(B b, C c, D d), ! E e, F f), the inner subexpression SEQ(B b, C c, D d) is bounded by A and F instances in the outer expression which is not changed after rewriting. Similarly, E instance is bounded by D and F instances which is not changed after rewriting.

Rule 3 *After applying FR3, AND(AND) is equivalent to AND().*

$$\begin{aligned} &AND(AND(E_1\ e_1,...,E_j\ e_j,\mathcal{P}),E_{j+1}\ e_{j+1},...,E_n\ e_n) \\ &\quad = AND(E_1\ e_1,...,E_j\ e_j,...,E_n\ e_n,\mathcal{P}). \end{aligned} \tag{5.18}$$

Proof:

Assume Exp_{inner} = $AND(E_1\ e_1, ..., E_j\ e_j, \mathcal{P})$. By Equation 5.6 (Definition of AND), we can write Exp_{inner} as

$$\begin{aligned} & AND(E_1\ e_1, ..., E_j\ e_j, \mathcal{P})[H_w] \\ & \quad = \{set_{of}(e_{1,j}) | (set_{of}(e_{1,j}) \in \Pi E_{1,j}[H_w]) \wedge (\mathcal{P} == true)\}. \end{aligned} \tag{5.19}$$

According to Equation 5.19, the event instance matching Exp_{inner} can be expressed by $set_{of}(e_{1,j})$. Using AND operator Definition 16, we have

$$\begin{aligned} & AND(AND(E_1\ e_1, ..., E_j\ e_j, \mathcal{P}), E_{j+1}\ e_{j+1}, ..., E_n\ e_n)[H_w] \\ & \quad = \{set_{of}(e_{1,n}) | set_{of}(e_{1,n}) \in AND(E_1\ e_1, ..., E_j\ e_j, \mathcal{P})[H_w] \times \Pi E_{j+1,n}[H_w]\} \end{aligned} \tag{5.20}$$

By substituting AND(E_1 e_1 ,..., E_j e_j, $\mathcal{P}$)[H_w] with the right hand side in Equation 5.19, for the right hand side of Equation 5.20, we have

$$\begin{aligned} & \{set_{of}(e_{1,n}) | set_{of}(e_{1,n}) \in AND(E_1\ e_1, ..., E_j\ e_j, \mathcal{P})[H_w] \times \Pi E_{j+1,n}[H_w]\} \\ & = \{set_{of}(e_{1,n}) | set_{of}(e_{1,n}) \in \{set_{of}(e_{1,j}) | (set_{of}(e_{1,j}) \in \Pi E_{1,j}[H_w]) \\ & \wedge (\mathcal{P} == true)\} \times \Pi E_{j+1,n}[H_w]\} \end{aligned} \tag{5.21}$$

Using event history cross product, for the right hand side of Equation 5.21, we have

$$
\begin{aligned}
&\{set_{of}(e_{1,n})|set_{of}(e_{1,n}) \in \{\{e_1,...,e_j\}|(\{e_1,...,e_j,e_{j+1},...,e_n\} \in \Pi E_{1,j}[H_w]) \\
&\wedge (\mathcal{P} == true)\} \times \Pi E_{j+1,n}[H_w]\} \\
&= \{set_{of}(e_{1,n})|set_{of}(e_{1,n}) \in (\Pi E_{1,j}[H_w] \times E_{j+1}[H_w]) \times .... \times E_n[H_w] \wedge \mathcal{P} == true\} \\
&= \{set_{of}(e_{1,n})|(set_{of}(e_{1,n})) \in \Pi E_{1,n}[H_w] \wedge (\mathcal{P} == true)\}.
\end{aligned}
\tag{5.22}
$$

On the right side, by Equation 5.6 (Definition of AND)

$$
\begin{aligned}
&AND(E_1\ e_1,...,E_j\ e_j,E_{j+1}\ e_{j+1},...,E_n\ e_n,\mathcal{P})[H_w] \\
&\quad = \{set_{of}(e_{1,n})|(set_{of}(e_{1,n})) \in \Pi E_{1,n}[H_w] \wedge (\mathcal{P} == true)\}.
\end{aligned}
\tag{5.23}
$$

So the expressions on the left side defined in Equation 5.20 and the right side by Equation 5.23 are equivalent. By induction, we can prove the correctness of Flattening Rule 3. □

Rule 4 *After applying FR4, AND(AND(!)) is equivalent to AND(!).*

$$
\begin{aligned}
&AND(AND(E_1\ e_1,...,!(E_i\ e_i,P_i(e_i)),E_j\ e_j,\mathcal{P}),E_{j+1}\ e_{j+1},...,E_n\ e_n) \\
&\quad = AND(E_1\ e_1,...,!(E_i\ e_i,P_i(e_i)),E_j\ e_j,...,E_n\ e_n,\mathcal{P}).
\end{aligned}
\tag{5.24}
$$

Proof:

Suppose e refers to an event instance of the inner subsequence $Exp_{inner}[H_w]$ = AND(E_1 e_1 ,..., ! (E_i e_i, $P_i(e_i)$), E_j e_j, $\mathcal{P}$)[H_w]. On the left side of Equation 5.24, the semantics of the expression corresponds to:

$$\begin{aligned}&AND(AND(E_1\ e_1,...,!(E_i\ e_i,P_i(e_i)),E_j\ e_j,\mathcal{P}),E_{j+1}\ e_{j+1},...,E_n\ e_n)[H_w]\\&=\{\{e,e_{j+1},...,e_n\}|\{e,e_{j+1},...,e_n\}\in Exp_{inner}[H_w]\times\Pi E_{j+1,n}[H_w]\}.\end{aligned}\tag{5.25}$$

Further expanding the inner sub-expression $Exp_{inner}[H_w]$ by Equation 5.7 we get

$$\begin{aligned}&AND(E_1\ e_1,...,!(E_i\ e_i,P_i(e_i)),E_j\ e_j,\mathcal{P})[H_w]\\&=\{\{e_1,...,e_{i-1},e_j\}|\{e_1,...,e_{i-1},e_j\}\in(\Pi E_{1,i-1}[H_w]\times E_j[H_w]))\\&\wedge\mathcal{P}==true\wedge(\nexists e_i\ where(e_i\in E_i[H_w]\wedge P_i(e_i)==true))\}.\end{aligned}\tag{5.26}$$

By plugging Equation 5.26 into Equation 5.25, we get:

$$\begin{aligned}&AND(AND(E_1\ e_1,...,!(E_i\ e_i,P_i(e_i)),E_j\ e_j,\mathcal{P}),E_{j+1}\ e_{j+1},...,E_n\ e_n)[H_w]\\&=\{\{e_1,...,e_{i-1},e_j,e_{j+1},...,e_n\}|\{e_1,...,e_{i-1},e_{j+1},...,e_n\}\in\Pi E_{1,i-1}[H_w]\times\Pi E_{j+1,n}[H_w]\\&\wedge(\mathcal{P}==true)\wedge(\nexists e_i\ where(e_i\in E_i[H_w]\wedge P_i(e_i)==true))\}.\end{aligned}\tag{5.27}$$

On the right side, according to Equation 5.7,

$$\begin{aligned}&AND(E_1\ e_1,...,!(E_i\ e_i,P_i(e_i)),E_j\ e_j,...,E_n\ e_n,\mathcal{P})[H_w]\\&=\{\{e_1,...,e_{i-1},e_j,...,e_n\}|\{e_1,...,e_{i-1},e_j,...,e_n\}\in\Pi E_{1,i-1}[H_w]\times\Pi E_{j,n}[H_w]\\&\wedge(\mathcal{P}==true)\wedge(\nexists e_i\ where(e_i\in E_i[H_w]\wedge P_i(e_i)==true))\}.\end{aligned}\tag{5.28}$$

So the expressions of the left side (Equation 5.27) and the right side (Equation 5.28) are equivalent. □

By Equation 5.7 (Definition of AND with negation), AND has at least one generating expression. Hence the flattened AND also has at least one generating expression.

Rule 5 *After applying FR5, OR(OR) is equivalent to OR().*

$$\begin{aligned} &OR(OR(E_1\ e_1,...,E_i\ e_i),E_j\ e_j,...,E_n\ e_n) \\ &= OR(E_1\ e_1,...,E_i\ e_i,E_j\ e_j,...,E_n\ e_n). \end{aligned} \tag{5.29}$$

Proof:

On the left side of Equation 5.29, according to Equation 5.9,

$$\begin{aligned} &OR(OR(E_1\ e_1,...,E_i\ e_i),E_j\ e_j,...,E_n\ e_n)[H_w] \\ &= \{\{e\}|\{e\} \in OR(E_1\ e_1,...,E_i\ e_i)[H_w]\} \cup ... \cup \{\{e_j\}|\{e_j\} \in E_j[H_w]\}... \\ &\{\{e_n\}|\{e_n\} \in E_n[H_w]\} \end{aligned} \tag{5.30}$$

On the right side, according to Equation 5.9,

$$\begin{aligned} &OR(E_1\ e_1\ ,...,E_i\ e_i,...,E_n\ e_n)[H_w] \\ &= \{\{e_1\}|\{e_1\} \in E_1[H_w]\} \cup ...\{\{e_i\}|\{e_i\} \in E_i[H_w]\}... \cup \\ &\{\{e_n\}|\{e_n\} \in E_n[H_w]\} \end{aligned} \tag{5.31}$$

So the expressions of the left side (Equation 5.30) and the right side (Equation 5.31) are equivalent. Equation 5.29 is correct. □

Rule 6 *After applying FR6, SEQ($\exists$ SEQ) is equivalent to SEQ($\exists$ E_i).*

$$\begin{aligned}&SEQ(E_1\ e_1, \exists SEQ(E_2\ e_2, ..., !(E_i\ e_i, P_i(e_i)), E_j\ e_j), E_{j+1}\ e_{j+1}, ..., E_n\ e_n)\\ &\quad = SEQ(E_1\ e_1, \exists E_2\ e_2, ..., !(E_i\ e_i, P_i(e_i)), \exists E_j\ e_j, E_{j+1}\ e_{j+1}, ..., E_n\ e_n).\end{aligned} \tag{5.32}$$

Proof:

On the left side of Equation 5.32, according to Equation 5.4,

$$\begin{aligned}&SEQ(E_1\ e_1, \exists SEQ(E_2\ e_2, ..., !(E_i\ e_i, P_i(e_i)), E_j\ e_j), E_{j+1}\ e_{j+1}, ..., E_n\ e_n)[H_w]\\ &\quad = \{\{e_1, e_{j+1}, ..., e_n\} | \{e_1, e_{j+1}, ..., e_n\} \in (E_1[H_w] \times \Pi E_{j+1,n}[H_w]) \wedge\\ &\quad (SEQ(E_2\ e_2, ..., !(E_i\ e_i, P_i(e_i)), E_j\ e_j)[H[e_1.te, e_{j+1}.ts]]! = \emptyset)\}.\end{aligned} \tag{5.33}$$

According to Equation 5.4, SEQ(E_2 e_2 ,..., ! (E_i e_i, $P_i(e_i)$), E_j e_j)[H[e_1.te, e_{j+1}.ts]] != $\emptyset$ in the right side of Equation 5.33 implies E_2 [H[e_1.te, e_{j+1}.ts]] != $\emptyset$ $\wedge$ ... $\wedge$ E_{i-1} [H[e_{i-2}.te, e_{j+1}.ts]] != $\emptyset$ $\wedge$ E_j [H[e_{i-1}.te, e_{j+1}.ts]] != $\emptyset$ $\wedge$ E_i [H[e_{i-1}.te, e_j.ts]] = $\emptyset$. Thus we have:

$$\begin{aligned}&SEQ(E_1\ e_1, \exists E_2\ e_2, ..., \exists E_{i-1}\ e_{i-1}, !(E_i\ e_i, P_i(e_i)), \exists E_j\ e_j, E_{j+1}\ e_{j+1}, ..., E_n\ e_n)[H_w]\\ &\quad = \{\{e_1, e_{j+1}, ..., e_n\} | \{e_1, e_{j+1}, ..., e_n\} \in (E_1[H_w] \times \Pi E_{j+1,n}[H_w]) \wedge\\ &\quad E_2[H[e_1.te, e_{j+1}.ts]]! = \emptyset \wedge ... \wedge E_{i-1}[H[e_{i-2}.te, e_{j+1}.ts]]! = \emptyset \wedge\\ &\quad E_j[H[e_{i-1}.te, e_{j+1}.ts]]! = \emptyset \wedge E_i[H[e_{i-1}.te, e_j.ts]] = \emptyset\}.\end{aligned} \tag{5.34}$$

On the right side of Equation 5.32, according to Equation 5.4,

$$
\begin{aligned}
&SEQ(E_1\ e_1, \exists E_2\ e_2, ..., \exists E_{i-1}\ e_{i-1}, !(E_i\ e_i, P_i(e_i)), \exists E_j\ e_j, E_{j+1}\ e_{j+1}, ..., E_n\ e_n)[H_w] \\
&\quad = \{\{e_1, e_{j+1}, ..., e_n\} | \{e_1, e_{j+1}, ..., e_n\} \in (E_1[H_w] \times \Pi E_{j+1,n}[H_w]) \wedge \\
&\quad E_2[H[e_1.te, e_{j+1}.ts]] != \emptyset \wedge ... \wedge E_{i-1}[H[e_{i-2}.te, e_{j+1}.ts]] != \emptyset \wedge \\
&\quad E_j[H[e_{i-1}.te, e_{j+1}.ts]] != \emptyset \wedge E_i[H[e_{i-1}.te, e_j.ts]] = \emptyset\}.
\end{aligned}
\tag{5.35}
$$

So the expressions of the left side (Equation 5.34) and the right side (Equation 5.35) are equivalent. □

Rule 7 *After applying FR7, AND(∃ AND) is equivalent to AND(AND).*

$$
\begin{aligned}
&AND(\exists AND(E_1\ e_1, ..., !(E_i\ e_i, P_i(e_i)), E_j\ e_j, \mathcal{P}), E_{j+1}\ e_{j+1}, ..., E_n\ e_n) \\
&\quad = AND(\exists E_1\ e_1, ..., !(E_i\ e_i, P_i(e_i)), E_j\ e_j, ..., E_n\ e_n, \mathcal{P}).
\end{aligned}
\tag{5.36}
$$

The proof for Rule FR7 is similar to the proof for Rule FR6. Thus the details are omitted.

5.2.4 Distributive Law

Each event type in the inner OR expression is distributed into the outer SEQ and AND expressions.

Rule 8 *After applying DR1, SEQ(OR) is equivalent to OR(SEQ).*

$$
\begin{aligned}
& SEQ(E_1\ e_1, OR(E_2\ e_2, ..., E_i\ e_i, P_2(e_2), ..., P_i(e_i)), E_j\ e_j, ..., E_n\ e_n) \\
& \quad = OR(SEQ(E_1\ e_1, E_2\ e_2, E_j\ e_j, ..., E_n\ e_n, P_2(e_2)), ..., \\
& \quad SEQ(E_1\ e_1, E_i\ e_i, E_j\ e_j, ..., E_n\ e_n, P_i(e_i)))
\end{aligned}
\tag{5.37}
$$

Proof: Suppose e refers to an event instance of the inner subsequence $Exp_{inner}[H_w]$ = OR(E_2 e_2 ,..., E_i e_i, $P_2(e_2)$,..., $P_i(e_i)$)[H_w]. According to Equation 5.9 (Definition of OR), we get $Exp_{inner}[H_w]$:

$$
\begin{aligned}
& OR(E_2\ e_2\ , ..., E_i\ e_i, P_2(e_2), ..., P_i(e_i))[H_w] \\
& \quad = (E_2[H_w], P_2(e_2)) \cup ... \cup (E_i[H_w], P_i(e_i)) \\
& \quad = \{e_2 | e_2 \in E_2[H_w] \wedge P_2(e_2) == true\} \cup ... \cup \{e_i | e_i \in E_i[H_w] \wedge P_i(e_i) == true\}.
\end{aligned}
\tag{5.38}
$$

According to Equation 5.3 (Definition of SEQ), the semantics of the left side of Rule 8 (Equation 5.37) corresponds to:

$$
\begin{aligned}
& SEQ(E_1\ e_1, OR(E_2\ e_2, ..., E_i\ e_i, P_2(e_2), ..., P_i(e_i)), E_j\ e_j, ..., E_n\ e_n)[H_w] \\
& \quad = \{\{e_1, e, e_j, ..., e_n\} | (\{e_1, e, e_j, ..., e_n\}) \in (E_1[H_w] \times Exp_{inner}[H_w] \times \Pi E_{j,n}[H_w]))\}
\end{aligned}
\tag{5.39}
$$

By plugging Equation 5.38 into Equation 5.39, we get

$$
\begin{aligned}
&SEQ(E_1\ e_1, OR(E_2\ e_2, ..., E_i\ e_i, P_2(e_2), ..., P_i(e_i)), E_j\ e_j, ..., E_n\ e_n)[H_w] \\
&= \{\{e_1, e_2, e_j, ..., e_n\} | (\{e_1, e_2, e_j, ..., e_n\} \in E_1[H_w] \times E_2[H_w] \times E_j[H_w] \times ... \\
&\times E_n[H_w]) \wedge P_2(e_2) == true\} \cup ... \cup \{\{e_1, e_i, e_j, ..., e_n\} | (\{e_1, e_i, e_j, ..., e_n\} \\
&\in E_1[H_w] \times E_i[H_w] \times E_j[H_w] \times ... \times E_n[H_w]) \wedge P_i(e_i) == true\}
\end{aligned}
\tag{5.40}
$$

On the right side, according to SEQ operator semantics in Equation 5.3 we get:

$$
\begin{aligned}
&OR(SEQ(E_1\ e_1, E_2\ e_2, E_j\ e_j, ..., E_n\ e_n, P_2(e_2))\ ... \\
&SEQ(E_1\ e_1, E_i\ e_i, E_j\ e_j, ..., E_n\ e_n, P_i(e_i)))[H_w] \\
&= \{\{e_1, e_2, e_j, ..., e_n\} | (\{e_1, e_2, e_j, ..., e_n\} \in E_1[H_w] \times E_2[H_w] \times E_j[H_w] \times ... \times \\
&E_n[H_w]) \wedge P_2(e_2) == true\} \cup ... \cup \{\{e_1, e_i, e_j, ..., e_n\} | (\{e_1, e_i, e_j, ..., e_n\} \in E_1[H_w] \\
&\times E_i[H_w] \times E_j[H_w] \times ... \times E_n[H_w]) \wedge P_i(e_i) == true\}
\end{aligned}
\tag{5.41}
$$

So the left side of Equation 5.37 as defined in Equation 5.40 and the right side of Equation 5.37 as defined in Equation 5.41 are equivalent. □

Rule 9 *After applying DR2, AND(OR) is equivalent to OR(AND).*

$$\begin{aligned}
&AND(E_1\ e_1, OR(E_2\ e_2, ..., E_i\ e_i, P_2(e_2), ..., P_i(e_i)), E_j\ e_j, ..., E_n\ e_n) \\
&\quad = OR(AND(E_1\ e_1, E_2\ e_2, E_j\ e_j, ..., E_n\ e_n, P_2(e_2))\ ... \\
&\quad AND(E_1\ e_1, E_i\ e_i, E_j\ e_j, ..., E_n\ e_n, P_i(e_i)))
\end{aligned} \tag{5.42}$$

The proof for Rule 9 is similar to the proof for Rule 8. Thus the details are omitted.

Rule 10 *After applying DR3, SEQ($\vee$) is equivalent to OR(SEQ).*

$$\begin{aligned}
&SEQ(E_1\ e_1, \exists(E_2\ e_2, P_2(e_2)) \vee, ..., \vee \exists(E_i\ e_i, ..., P_i(e_i)), E_j\ e_j ... E_n\ e_n) \\
&\quad = OR(SEQ(E_1\ e_1, \exists(E_2\ e_2, P_2(e_2)), E_j\ e_j, ..., E_n\ e_n)\ ... \\
&\quad SEQ(E_1\ e_1, \exists(E_i\ e_i, P_i(e_i)), E_j\ e_j, ..., E_n\ e_n))
\end{aligned} \tag{5.43}$$

The proof for Rule 11 is similar to the proof for Rule 8. Thus the details are omitted.

Rule 11 *After applying DR4, AND($\vee$) is equivalent to OR(AND).*

$$\begin{aligned}
&AND(E_1\ e_1, \exists(E_2\ e_2, P_2(e_2)) \vee, ..., \vee \exists(E_i\ e_i, ..., P_i(e_i)), ..., E_n\ e_n) \\
&\quad = OR(AND(E_1\ e_1, \exists(E_2\ e_2, P_2(e_2)), ..., E_n\ e_n)\ ... \\
&\quad AND(E_1\ e_1, \exists(E_i\ e_i, P_i(e_i)), ..., E_n\ e_n))
\end{aligned} \tag{5.44}$$

The proof for Rule 5.44 is similar to the proof for Rule 8. Thus the details are omitted.

Rule 12 *After applying DR5, SEQ(∃ OR) is equivalent to OR(SEQ).*

$$
\begin{aligned}
&SEQ(E_1\ e_1, \exists OR(E_2\ e_2, ..., E_i\ e_i, P_2(e_2), ..., P_i(e_i)), E_j\ e_j, ..., E_n\ e_n) \\
&\quad = OR(SEQ(E_1\ e_1, \exists E_2\ e_2, E_j\ e_j, ..., E_n\ e_n, P_2(e_2))\ , ..., \\
&\quad SEQ(E_1\ e_1, E_i\ e_i, E_j\ e_j, ..., E_n\ e_n, P_i(e_i)))
\end{aligned}
\tag{5.45}
$$

The proof for Rule 5.45 is similar to the proof for Rule 8. Thus the details are omitted.

Rule 13 *After applying DR6, AND(∃ OR) is equivalent to OR(AND).*

$$
\begin{aligned}
&AND(E_1\ e_1, \exists OR(E_2\ e_2, ..., E_i\ e_i, P_2(e_2), ..., P_i(e_i)), E_j\ e_j, ..., E_n\ e_n) \\
&\quad = OR(AND(E_1\ e_1, \exists(E_2\ e_2, P_2(e_2)), E_j\ e_j, ..., E_n\ e_n)\ ... \\
&\quad AND(E_1\ e_1, \exists(E_i\ e_i, P_i(e_i)), E_j\ e_j, ..., E_n\ e_n))
\end{aligned}
\tag{5.46}
$$

The proof for Rule 5.46 is similar to the proof for Rule 8. Thus the details are omitted.

5.2.5 Negation Push Down Rules

For negation (!) in expressions satisfying our language constraint in Section 5.2.2, negation (!) is pushed into the inner AND, SEQ or OR subexpression so that ! is before each primitive even type.

Rule 14 *After applying NPDR1 (right-to-left unroll), ! SEQ is equivalent to pushing negation (!) into the inner SEQ subexpression. The preconditions of NPD1 are for 1 $\leq$ j $\leq$ i, E_j must be primitive and E_j is not a boolean expression ! E_j.*

$\overrightarrow{E_1, e_1, E_{i-1}, e_{i-1}}$ denotes E_1, e_1, E_2, e_2 ,..., E_{i-1}, e_{i-1}.

$$
\begin{aligned}
&!SEQ(\overrightarrow{E_1, e_1, E_{i-1}, e_{i-1}}, E_i\ e_i) \\
&\quad =!(E_i\ e_i) \vee \\
&\quad \exists SEQ(!(E_{i-1}\ e_{i-1}), E_i\ e_{i1}, !(E_i\ e_{i2})) \vee \\
&\quad \exists SEQ(!(E_{i-2}\ e_{i-2}), E_{i-1}\ e_{i-1}, !(E_{i-1}\ e_{i-1}), E_i\ e_{i1}, !(E_i\ e_{i2})) \vee \\
&\quad \ldots \\
&\quad \exists SEQ(!(E_1\ e_1), E_2\ e_{21}, !(E_2\ e_{22}), \ldots, E_i\ e_{i1}, !(E_i\ e_{i2})) \\
&\quad \text{Assume that for } 2 \leq \text{j} \leq \text{i, } E_j \text{ must be primitive and for } 1 \leq \text{k} \leq \text{i-1} \\
&\quad E_k \text{ is not a boolean expression ! } E_k.
\end{aligned}
\tag{5.47}
$$

Proof: Let us prove Equation 5.48 first. Equation 5.47 can be proven by applying Equation 5.48 i times.

$$
\begin{aligned}
&!SEQ(\overrightarrow{E_1, e_1, E_{i-1}, e_{i-1}}, E_i\ e_i)[H_w] \\
&\quad =!(E_i\ e_i)[H_w] \vee \\
&\quad \exists SEQ(!SEQ(\overrightarrow{E_1, e_1, E_{i-1}, e_{i-1}}), E_i\ e_{i1}, !(E_i\ e_{i2}))[H_w] \\
&\quad \text{Assume that the last event type } E_i \text{ must be primitive and } E_{i-1} \text{ is not ! E.}
\end{aligned}
\tag{5.48}
$$

[**Proof for** $\rightarrow$] First, let us prove if the left hand side is true, then the right hand side is true.

If the left hand side is true, then the one of the following must hold. (II) E_i instances are missing; (I) At least one instance of E_1 ,..., E_{i-1} is missing; or (III) while none is missing but event instance ordering in sequence query is not satisfied among events of types E_1 ... E_i.

Now for each case we will prove that the right hand ride is true.

Case I: E_i instances are missing; According to Definition 12, ! $(E_i e_i)[H_w]$ = true. Thus, the right side of Equation 5.48 is true.

Case II: E_1 ... or E_{i-1} instances are missing. SEQ(E_1, e_1, E_2, e_2 ,..., E_{i-1}, e_{i-1})[H_w] = $\emptyset$. Two cases exist for E_i: E_i is also missing or E_i exists. If no E_i events exist, the case falls into Case I. Thus, the right side of Equation 5.48 is true. Otherwise, the last e_i in H_w among these event instances of the type E_i matches E_i e_i, ! (E_i e_i) as no more e_i of type E_i exists after the last one. According to Equation 5.4 (Definition of SEQ with Negation), SEQ(! SEQ($\overrightarrow{E_1, e_1, E_{i-1}, e_{i-1}}$), E_i e_i, ! (E_i e_i))[H_w] returns an event history which contains the last $e_i \in E_i[H_w]$. Thus according to Definition 12 for boolean expressions, $\exists$ SEQ(! SEQ($\overrightarrow{E_1, e_1, E_{i-1}, e_{i-1}}$), E_i e_{i1}, ! (E_i e_{i2}))[H_w] = true. Thus, the right side of Equation 5.48 is true.

Case III: None is missing but ordering is not satisfied. If the ordering between e_1 ,..., e_{i-1} events is not satisfied, SEQ($\overrightarrow{E_1, e_1, E_{i-1}, e_{i-1}}$) = $\emptyset$. ! SEQ($\overrightarrow{E_1, e_1, E_{i-1}, e_{i-1}}$) = true. The result is the same as Case II. According to Case II above, the right side of Equation 5.48 is true. Otherwise, if the the ordering between e_{i-1} and e_i events is not satisfied, it mean no sequences e_1 ,..., e_{i-1} exist before e_i. According to Equation 5.4 (Definition of SEQ with Negation), SEQ(! SEQ($\overrightarrow{E_1, e_1, E_{i-1}, e_{i-1}}$), E_i e_i, ! (E_i e_i)) returns an event history contains the last E_i event instance in H_w. $\exists$

SEQ(! SEQ($\overrightarrow{E_1, e_1, E_{i-1}, e_{i-1}}$), E_i e_{i1}, ! (E_i e_{i2}))[H_w] = true. Thus, the right side of Equation 5.48 is true.

We require that E_{i-1} is not a boolean expression ! E. The non-existence semantics is changed otherwise. Please refer to Example 21 below. To guarantee SEQ(E_i e_i, ! E_i, e_i)[H_w] represent the last E_i (no E_i instances exist after a matching E_i instance) in the input stream, E_i must be primitive. Problems would occur otherwise (see Example 22).

[**Proof for** $\leftarrow$] Next, let us prove that if the right hand side is true, then the left hand side is also true.

For the expression on the right side of Equation 5.48, if it is evaluated to be true, either (I) !E_iH[ts, te] = true. No events of type E_i exist in H[ts, te] or (II) Before the last E_i event in H[ts, te], no sequence results for $SEQ(\overrightarrow{E_1, e_1, E_{i-1}, e_{i-1}})$ exist. The above two cases mean the event history for SEQ($\overrightarrow{E_1, e_1, E_{i-1}, e_{i-1}}$, E_i e_i))[H[ts, te]] is empty. Thus, the expression on the left side of Equation 5.48 is true. We prove Equation 5.48 is correct. □

Example 21 *Assume a query Q_4 = SEQ(Recycle r, ! SEQ(Sharpening s, ! (Disinfection d), Checking c), Operating o). After applying Rule NPD1 (right-to-left unroll), we get Q'_4 = SEQ(Recycle r, ! (Checking c) $\vee$ $\exists$ SEQ(! SEQ(Sharpening s, ! (Disinfection d)), Checking c, ! (Checking c), Operating o). Q_4 requires the existence of Disinfection instances between every Sharpening and Checking instance pair. However, Q'_4 requires the existence of Disinfection instances between every Sharpening and the last Checking instance pair (represented by Checking c, ! (Checking c)). The precondition requiring the event type before Checking is not a boolean expression ! E (! Disinfection here) is produced on purpose.*

Example 22 *SEQ(E_i e_{i1}, ! E_i e_{i2}) will return the last E_i instance in stream if E_i is a primitive event type. However, if E_i is a composite event type, the event instance returned by SEQ(E_i e_{i1}, ! E_i e_{i2}) may not be the last E_i in the stream. Assume E_i is a composite event type SEQ(Checking c, Operating o) with input events $\{c_2, c_3, o_4, o_5\}$. The last E_i event should be $\{c_3, o_5\}$. However, $\{c_2, o_4\}$ would also match SEQ(E_i, ! E_i) as we can't find another E_i event that occurs strictly after it. The reason is that $\{c_2, o_4\}$ and $\{c_3, o_5\}$ are overlapping with $\{c_2, o_4\}$.te = 4 and $\{c_3, o_5\}$.ts = 3.*

Basing on above proof, we have shown that Equation 5.48 is true. □

Rule 15 *After applying NPDR2 (left-to-right unroll), ! SEQ is equivalent to pushing negation (!) into the inner SEQ subexpression. The preconditions are for 1 ≤ j ≤ i type E_j must be primitive and E_j is not a boolean expression ! E_j.*

Below, $\overrightarrow{E_2, e_2, ..., E_i, e_i}$ denotes E_2, e_2 ,..., E_i, e_i.

$$
\begin{aligned}
&!SEQ(E_1\ e_1, \overrightarrow{E_2, e_2, E_i, e_i}) \\
&\quad = !(E_1\ e_1) \vee \\
&\quad \exists SEQ(!(E_1\ e_{11}),\ E_1\ e_{12},\ !(E_2, e_2)) \vee \\
&\quad \ldots \\
&\quad \exists SEQ(!(E_1\ e_{11}),\ E_1\ e_{12},\ \ldots\ !(E_{i-1}\ e_{1i-1}), E_{i-1}\ e_{2i-1}, !(E_i\ e_i)) \\
&\quad \text{Assume that for } 2 \leq \text{j} \leq \text{i type } E_j \text{ must be primitive and for } 1 \leq \text{k} \leq \text{i-1}, \\
&\quad E_k \text{ is not a boolean expression ! } E_k.
\end{aligned}
\tag{5.49}
$$

Proof: Let us prove Equation 5.50 first. Equation 5.49 can be proven by applying Equation 5.50 i times.

$$!SEQ(E_1\ e_1, \overrightarrow{E_2, e_2, E_i, e_i})$$
$$= !(E_1\ e_1) \vee$$
$$\exists SEQ(!(E_1\ e_{11}),\ E_1\ e_{12},\ !SEQ(\overrightarrow{E_2, e_2, E_i, e_i}))$$
given that the first event type E_1 must be primitive and E_2 is not a boolean expression ! E.
(5.50)

[**Proof for** $\rightarrow$]. First, let us prove if the left hand side of Equation 5.50 is true, then the right hand side is true.

If the left hand side is true, then the one of the following must hold. (I) If E_1 is missing; (II) At least one instance of E_2 ... E_i is missing; or (III) If none is missing, then their ordering is not satisfied.

Now for each case we will prove that the right hand ride is true.

Case I E_1 is missing; ! $(E_1\ e_1)[H_w]$ = true. Thus the right hand side of Equation 5.50 is true.

Case II If at least one instance of E_2 ... E_i is missing, SEQ(E_2, e_2 ,..., E_i, e_i) = $\emptyset$. SEQ(! $(E_1\ e_1)$, $E_1\ e_1$)[H_w] represents the first E_1 event in H_w as before a matching E_1 event instance, no more E_1 event instances exist in H_w. If E_1 instances exist in H_w, SEQ(! $(E_1\ e_1)$, $E_1\ e_1$, ! SEQ(E_2, e_2 ,..., E_i, e_i))[H_w] returns the first E_i instance. Thus the right hand side of Equation 5.50 is true. Otherwise, if E_1 instances do not exist in H_w, $(E_1\ e_1)[H_w] = \emptyset$. ! $(E_1\ e_1)[H_w]$ = true. Thus the right hand side of Equation 5.50 is true.

Case III If none is missing but their ordering in sequence query is not satisfied. If the orderng among e_2 ,..., e_i is not satisfied, SEQ(E_2, e_2 ,..., E_i, e_i) = $\emptyset$. The result is the same to Case II. Otherwise, if the ordering between e_1 and e_2 is not satisfied, it mean no sequences e_2 ,..., e_i exist after e_1 of type E_1. According to Equation 5.4 (Definition of SEQ with Negation), SEQ(! (E_1 e_1), E_1 e_1, ! SEQ($\overrightarrow{E_2, e_2, E_i, e_i}$)) returns an event history contains the first E_1 event in H_w. $\exists$ SEQ(! (E_1 e_{11}), E_1 e_{12}, ! SEQ($\overrightarrow{E_2, e_2, E_i, e_i}$))[$H_w$] = true. Thus, the right side of Equation 5.50 is true.

To guarantee SEQ(! E_1 e_1, E_1 e_1)[H_w] represent the first E_1 in the input stream, E_1 must be primitive. We also require that E_2 is not a boolean expression ! E. The reasons for these requirements are the same as the requirements for Rule NPD1.

[**Proof for** $\leftarrow$] For the expression on the right side of Equation 5.50, if it is evaluated to be true, either (I) No E_1 events exist in H_w or (II) After the first E_1 event in H_w, no sequence results for SEQ($\overrightarrow{E_2, e_2, E_i, e_i}$) exist. Thus it means the event history for SEQ($\overrightarrow{E_1, e_1, E_i, e_i}$)[$H_w$] is empty. Thus the boolean expression on the left side of Equation 5.50 is true. We thus have proven that Equation 5.50 is correct. □

Based on above proof, we have proven that Equation 5.50 is true. □

Rule 16 *After applying NPDR3, ! AND is equivalent to pushing negation (!) into the inner AND subexpression.*

$$!AND(E_1\ e_1, ..., E_i\ e_i, \mathcal{P})\ =\ !(E_1\ e_1, P_1(e_1))\ \vee ... \vee\ !(E_i\ e_i, P_i(e_i)) \tag{5.51}$$

Proof: We prove that if the left hand side holds true, then the right hand side also holds true and vice versa.

[Proof for $\rightarrow$] If the boolean expression on the left side of Equation 5.51 is evaluated to be true, AND(E_1 ,..., E_i, $\mathcal{P}$)[H_w] = $\emptyset$. It means $\exists$ E_i[H_w] = $\emptyset$. Thus at least one subexpression on the right side of Equation 5.51 holds true.

[Proof for $\leftarrow$] According to Equation 5.9, the right side of Equation 5.51 requires in [H_w], $\exists$ E_i, !(E_i e_i,$P_i(e_i)$)[H_w] = true. The implies not all E_1 ,...., E_i instances exist in [H_w]. So the left side of Equation 5.51 is true. So the Rule 17 is correct. □

Rule 17 *After applying NPDR4, ! AND is equivalent to pushing negation (!) into the inner AND subexpression with boolean expressions.*

$$\begin{aligned} &!AND(E_1\ e_1, ..., !(E_i\ e_i, P_i(e_i)), ..., \exists E_j\ e_j, \mathcal{P}) \\ &=!(E_1\ e_1, P_1(e_1)) \vee ... \vee \exists(E_i\ e_i, P_i(e_i)) ... \vee !(E_j\ e_j, P_j(e_j)) \end{aligned} \tag{5.52}$$

The proof for NPD4 is similar to the proof for NPD3. Thus the details are omitted.

Rule 18 *After applying NPDR5, ! OR is equivalent to pushing negation (!) into the inner OR subexpression. All E_1 ,..., E_i in OR are generating subexpressions.*

$$\begin{aligned} &!OR(E_1\ e_1, E_2\ e_2, ..., E_i\ e_i, \mathcal{P}) \\ &=!(E_1\ e_1, P_1(e_1)) \wedge !(E_2\ e_2, P_2(e_2)) \wedge ... \wedge !(E_i\ e_i, P_i(e_i)) \end{aligned} \tag{5.53}$$

Proof:

On the left side of Equation 5.53,

$$\begin{aligned} &!OR(E_1\, e_1,\, E_2\, e_2, ..., E_i\, e_i, \mathcal{P})[H_w] \\ &=!(\{\{e_1\} \in E_1[H_w] \wedge P_1(e_1)\} \cup ... \cup \{\{e_i\} \in E_i[H_w] \wedge P_i(e_i)\}) \end{aligned} \tag{5.54}$$

For the expression on the right hand side of Equation 5.54 to be true, $\{\{e_1\} \in E_1[H_w] \wedge P_1(e_1)\}$,..., $\{\{e_i\} \in E_i[H_w] \wedge P_i(e_i)\}$ all return empty. Thus for the right hand side of Equation 5.54, we have

$$= (\nexists e_1 \in E_1[H_w] \wedge P_1(e_1) == true) \wedge ... \wedge (\nexists e_i \in E_i[H_w] \wedge P_i(e_i) == true) \tag{5.55}$$

We now prove Equation 5.55 is true. If the left hand side of Equation 5.55 is true, $(e_1 \in E_1[H_w] \wedge P_1(e_1) \cup ... \cup e_i \in E_i[H_w] \wedge P_i(e_i)) = \emptyset$. Namely, $E_1[H_w] = ... = E_i[H_w] = \emptyset$. Thus the right hand side of Equation 5.55 is true. If the right hand side of Equation 5.55 is true, $(\nexists\ e_1 \in E_1[H_w] \wedge P_1(e_1)$ == true) = true ,..., $(\nexists\ e_i \in E_i[H_w] \wedge P_i(e_i)$ == true) = true. Thus $e_1 \in E_1[H_w] \wedge P_1(e_1) = \emptyset$,..., $e_i \in E_i[H_w] \wedge P_i(e_i) = \emptyset$. Thus the left hand side of Equation 5.55 is true.

On the right side of Equation 5.53,

$$\begin{aligned} &!(E_1\, e_1, P_1(e_1))[H_w] \wedge !(E_2\, e_2, P_2(e_2))[H_w]\ \wedge\ ...,!(E_i\, e_i, P_i(e_i))[H_w] \\ &= (\nexists e_1 \in E_1[H_w] \wedge P_1(e_1) == true) \wedge ... \wedge (\nexists e_i \in E_i[H_w] \wedge P_i(e_i) == true) \end{aligned} \tag{5.56}$$

The left side of Equation 5.54 is equal to the right side of Equation 5.56. So the Rule 18 is correct. □

Rule 19 *After applying NPDR6, pushing negation (!) into the inner SEQ expression with exist $\exists$ boolean subexpressions is equivalent to ! SEQ with all generating subexpressions.*

$$!SEQ(E_1\, e_1, ..., \exists E_i\, e_i, ..., E_n\, e_n, \mathcal{P}) = !SEQ(E_1\, e_1, ..., E_i\, e_i, ..., E_n\, e_n, \mathcal{P}) \quad (5.57)$$

Proof. We prove that if the left hand side holds true, then the right hand side also holds true and vice versa.

[Proof for $\rightarrow$] If the boolean expression on the left side of Equation 5.57 is evaluated to be true, SEQ(E_1 e_1 ,..., $\exists$ E_i e_i ,..., E_n e_n, $\mathcal{P}$) = $\emptyset$. Two cases are possible: (I) No results matching SEQ(E_1 e_1 ,..., E_{i-1} e_{i-1}, E_{i+1} e_{i+1} ,..., E_n e_n, $\mathcal{P}$). (II) No $e_i \in E_i$ exists with e_{i-1}.ts $\leq e_i$.ts $\leq e_{i+1}$.ts. Thus the expression on the right side of Equation 5.57 holds true.

[Proof for $\leftarrow$] If the boolean expression on the right side of Equation 5.57 is evaluated to be true, SEQ(E_1 e_1 ,..., E_i e_i ,..., E_n e_n, $\mathcal{P}$) = $\emptyset$. Two cases are possible: (I) No results matching SEQ(E_1 e_1 ,..., E_{i-1} e_{i-1}, E_{i+1} e_{i+1} ,..., E_n e_n, $\mathcal{P}$). (II) No $e_i \in E_i$ exists with e_{i-1}.ts $\leq e_i$.ts $\leq e_{i+1}$.ts. Thus the expression on the left side of Equation 5.57 holds true. So the Rule NPD6 is correct. □

Rule 20 *After applying NPD7, pushing negation (!) into the inner AND expression with exist $\exists$ boolean subexpressions is equivalent to ! AND with all generating subexpressions.*

$$!AND(E_1\ e_1, ..., \exists E_i\ e_i, ..., E_n\ e_n, \mathcal{P})\ = !AND(E_1\ e_1, ..., E_i\ e_i, ..., E_n\ e_n, \mathcal{P}) \quad (5.58)$$

The proof for NPD7 is similar to the proof for NPD6. Thus the details are omitted.

5.2.6 Normal Forms for CEP Expressions

Rewriting aims to flatten nested *NEEL* expressions as much as possible to overcome the two problems described in Section 5.1. In addition, sharable subexpressions would be easily identified in flattened expressions. We distinguish between two normal forms for *NEEL* expressions of Class Lcons defined in Table 3.5: disjunctive normal form (DNF) and conjunctive normal form (CNF).

Definition 20 *A NEEL event expression E is said to be in* **disjunctive normal form** *if it is of the form (E OR E OR ... OR E) with each query conjunct E a sequential pattern specified with one SEQ or AND formed by primitive event types.*

Example 23 *q = SEQ(Recycle r, Washing w) OR SEQ(Recycle r, ∃ Washing w, Sharpening) OR SEQ(Recycle r, ! Washing w, Sharpening s) is in disjunctive normal form as defined in Definition 20.*

Definition 21 *A NEEL event expression E is said to be in* **conjunctive normal form** *if it is of the form (E AND E AND ... AND E) with each query disjunct E a sequential pattern specified with one SEQ formed by primitive event types.*

Example 24 *q = SEQ(Recycle r, Washing w) AND SEQ(Recycle r, ∃ Washing w, Sharpening) AND SEQ(Recycle r, ! Washing w, Sharpening) is in conjunctive normal form.*

5.2.7 NEEL Expression Flattening Procedure

Not all expressions expressed by NEEL can be rewritten as described by our language constraints in Section 5.2.2. We can only rewrite expressions defined by Class Lcons in Table 3.5. We can't rewrite nested SEQ and AND (e.g., SEQ(AND), SEQ(∃ AND), AND(∃ SEQ), AND(SEQ), AND(!SEQ)) and double negation on SEQ (e.g., ! SEQ(!)). Double negation over AND and OR could be removed (see Section 5.2.2). After applying negation push down over AND and OR until no longer applicable, if double negation on SEQ (e.g., ! SEQ(!)), nested SEQ and AND (e.g., SEQ(AND), SEQ(∃ AND), AND(∃ SEQ), AND(SEQ) and AND(!SEQ)) still exist, such nested expressions can't be flattened under our current model.

Input: An event expression Exp_{in} which satisfies the language constraints in Section 5.2.2.

Output: A normalized expression Exp_{out} of expression type as in Definitions 20 and 21 (Section 5.2.6).

- *Step 1:* Apply Flattening Rules until they are no longer applicable (flattening rules 1-3).
- *Step 2:* Push ! into expressions recursively by applying the Negation Push Down Rules (NPDR 1-6).
  - *Step 2.1:* Apply Negation over OR/AND until they are no longer applicable.

- *Step 2.2:* Apply Negation over SEQ(left-to-right/right-to-left) rules until they are no longer applicable;

- *Step 3:* Apply Distributive Rules until they are no longer applicable (distributive rules 1-3).

- *Step 4:* If the rewritten expression is in one of the normal forms, stop the procedure. Otherwise, iterate to *Step 1*.

Example 25 *Given the NEEL expression Q_6 = SEQ(E_1, ! SEQ(E_2, E_3), E_4, SEQ(! AND(E_5, E_6), E_7))*

- *By step 1 applying flattening rule, we get Q_6 =*
  SEQ(E_1, ! SEQ(E_2, E_3), E_4, ! AND(E_5, E_6), E_7)

- *By step 2.1 applying the negation push down rule over AND, we get Q_6 =*
  SEQ(E_1, ! SEQ(E_2, E_3), E_4, ! E_5 $\vee$! E_6, E_7);

- *By step 2.2 applying the negation push down rule over SEQ, we get Q_6 =*
  SEQ(E_1, !E_2 $\vee$ $\exists$ SEQ(!E_2, E_2, !E_3), E_4, !E_5 $\vee$!E_6, E_7);

- *By step 3 applying distributive rule, we get Q_6 =*
  OR(SEQ(E_1, ! E_2, E_4, ! E_5, E_7),
  SEQ(E_1, ! E_2, E_4, ! E_6, E_7),
  SEQ(E_1, $\exists$SEQ(! E_2, E_2, ! E_3), E_4, ! E_5, E_7),
  SEQ(E_1, $\exists$SEQ(! E_2, E_2, ! E_3), E_4, ! E_6, E_7));

  As Q_6 is not in any of the normal forms, apply step 1 again iteratively:

- *By step 1 applying the flattening rule, we get Q_6 =*
  SEQ(E_1, ! E_2, E_4, ! E_5, E_7) OR

SEQ(E_1, ! E_2, E_4, ! E_6, E_7) OR

SEQ(E_1, ! E_2, $\exists E_2$, ! E_3, E_4, ! E_5, E_7) OR

SEQ(E_1, ! E_2, $\exists E_2$, ! E_3, E_4, ! E_6, E_7);

As Q_6 is in the disjunctive normal form as defined in Definition 20, the rewriting procedure is stopped.

5.2.8 Properties of the Rewriting System

Before we show the properties of our rewriting system, we quantify the complexity of a nested CEP expression by the nesting levels. For operators and boolean connectors, we have SEQ, AND, OR, !, $\exists$, $\vee$, $\wedge$. We have the following combinations which are covered by cases in Table 5.2.8 below with the operators SEQ, AND and OR as the outer operator respectively.

Inner Expression Cases Considered for an Outer SEQ operator	
primitive event type E_i	[1] E_i [2] $\exists\ E_i$ [3] ! E_i
SEQ operator	[4] SEQ(Exp_1 ,..., Exp_n)
OR operator	[5] OR(Exp_1 ,..., Exp_i ,...., Exp_n)
AND operator	[6] AND(Exp_1 ,..., Exp_n)
$\exists$ SEQ operator	[7] $\exists$ SEQ(Exp_1 ,..., ! Exp_i ,..., $\exists\ Exp_k$,...., Exp_n)
$\exists$ OR operator	[8] $\exists$ OR(Exp_1 ,..., Exp_i ,...., Exp_n)
$\exists$ AND operator	[9] $\exists$ AND(Exp_1 ,..., Exp_i ,...., Exp_n)
! SEQ operator	[10] ! SEQ(Exp_1 ,..., Exp_i ,...., Exp_n)
! OR operator	[11] ! OR(Exp_1 ,..., Exp_i ,...., Exp_n)
! AND operator	[12] ! AND(Exp_1 ,..., ! Exp_i ,...., Exp_n)
Boolean Connectors	[13] $\exists\ Exp_1 \wedge ... \wedge\ !\ Exp_n$ [14] $!\ Exp_1 \vee ... \vee \exists\ Exp_n$

Inner Expression Cases Considered for an Outer AND operator	
primitive event type E_i	[1] E_i [2] $\exists$ E_i [3] ! E_i
SEQ operator	[4] SEQ(Exp_1 ,..., Exp_i ,...., Exp_n)
OR operator	[5] OR(Exp_1 ,..., Exp_i ,...., Exp_n)
AND operator	[6] AND(Exp_1 ,..., Exp_n)
$\exists$ SEQ operator	[7] $\exists$ SEQ(Exp_1 ,..., Exp_i ,...., Exp_n)
$\exists$ OR operator	[8] $\exists$ OR(Exp_1 ,..., Exp_i ,...., Exp_n)
$\exists$ AND operator	[9] $\exists$ AND(Exp_1 ,..., Exp_i ,...., Exp_n)
! SEQ operator	[10] ! SEQ(Exp_1 ,..., ! Exp_i ,...., Exp_n)
! OR operator	[11] ! OR(Exp_1 ,..., Exp_i ,...., Exp_n)
! AND operator	[12] ! AND(Exp_1 ,..., ! Exp_i ,...., Exp_n)
Boolean Connectors	[13] $\exists$ $Exp_1 \wedge ... \wedge$! Exp_n [14] ! $Exp_1 \vee ... \vee \exists$ Exp_n

Inner Expression Cases Considered for an Outer OR operator	
primitive event type E_i	[1] E_i [2] $\exists$ E_i [3] ! E_i
AND operator	[4] AND(Exp_1 ,..., Exp_n)
SEQ operator	[5] SEQ(Exp_1 ,..., Exp_n)
OR operator	[6] OR(Exp_1 ,..., Exp_i ,...., Exp_n)

Definition 22 *For a query q, α represents the maximum operator nesting levels of q. α is designed such that for an expression Exp in one of our normal forms, α(Exp) = 0. For the cases shown in Table 5.2.8 with SEQ as the outer operator. α is computed according to the following equation:*

$$\alpha(Exp) = \begin{cases} 0 & \textit{if Exp = Case[1-3]} \\ MAX(\alpha(Exp_i)+1, 1 \leq i \leq n) & \textit{if Exp = Case[4-6]} \\ MAX(\alpha(Exp_i)+2, 1 \leq i \leq n) & \textit{if Exp = Case[7-9]} \\ MAX(\alpha(Exp_i)+3, 1 \leq i \leq n) & \textit{if Exp = Case[10-12]} \\ MAX(\beta(Exp_i), 1 \leq i \leq n) & \textit{if Exp = Case[13-14]} \\ & \textit{if } Exp_i \textit{ is primitive } E_i \\ & \beta(Exp_i) = alpha(Exp_i) \\ & \textit{if } Exp_i \textit{ is SEQ(), AND(), OR()} \\ & \beta(Exp_i) = alpha(Exp_i) + 2 \end{cases}$$

For the cases shown in Table 5.2.8 with AND as the outer operator. α *is computed according to the following equation:*

$$\alpha(Exp) = \begin{cases} 0 & \textit{if Exp = Case[1-3]} \\ MAX(\alpha(Exp_i), 1 \leq i \leq n) & \textit{if Exp = Case[4]} \\ MAX(\alpha(Exp_i)+1, 1 \leq i \leq n) & \textit{if Exp = Case[5-6]} \\ MAX(\alpha(Exp_i)+2, 1 \leq i \leq n) & \textit{if Exp = Case[7-9]} \\ MAX(\alpha(Exp_i)+3, 1 \leq i \leq n) & \textit{if Exp = Case[10-12]} \\ MAX(\beta(Exp_i), 1 \leq i \leq n) & \textit{if Exp = Case[13-14]} \\ & \textit{if } Exp_i \textit{ is primitive } E_i \\ & \beta(Exp_i) = alpha(Exp_i). \\ & \textit{if } Exp_i \textit{ is SEQ(), AND(), OR()} \\ & \beta(Exp_i) = alpha(Exp_i) + 2. \end{cases}$$

For the cases shown in Table 5.2.8 with OR as the outer operator, α *is computed according to the following equation:*

$$\alpha(Exp) = \begin{cases} 0 & \textit{if Exp = Case[1-3]} \\ MAX(\alpha(Exp_i), 1 \leq i \leq n) & \textit{if Exp = Case[4-5]} \\ MAX(\alpha(Exp_i)+1, 1 \leq i \leq n) & \textit{if Exp = Case[6]} \end{cases}$$

α(*Exp*) = 0 if *Exp* is in one of the normal forms in Section 5.2.6. Primitive event types doesn't increase the nesting level α. Subexpressions with ! AND, ! SEQ and ! OR increase the nesting level α by 3. We don't increase α for expressions with AND(SEQ()) as it exists in CNF. We don't increase α for expressions with OR(SEQ()) and OR(AND()) as they exist in DNF. We now explain Definition 22 by the following examples.

Example 26 *To compute* $\alpha(SEQ(SEQ(E_1, !AND(E_3, E_4)), AND(E_5, E_6)))$*, we have:*

- $\alpha(SEQ(SEQ(E_1, !AND(E_3, E_4)), AND(E_5, E_6)))$
  $= MAX(\alpha\ (SEQ(E_1,\ !\ AND(E_3,\ E_4))) + 1,\ MAX(\alpha(E_5) + 1,\ \alpha(E_6) + 1))$
  (cases 4, 6 with SEQ as the outer operator)
- $\alpha(E_5) = \alpha(E_6) = 0$ *(case 1 with AND as the outer operator);*
- $\alpha\ (SEQ(E_1,\ !\ AND(E_3,\ E_4))) = MAX(\alpha\ (E_1),\ MAX(\alpha(E_3) + 3,\ \alpha(E_4) + 3)) = 3$ *(cases 1 and 12 with SEQ as the outer operator);*
- $\alpha\ (E_1) = \alpha\ (E_3) = \alpha\ (E_4) = 0$ *(case 1)*

Thus $\alpha(SEQ(SEQ(E_1, !AND(E_3, E_4)), AND(E_5, E_6))) = 4$.

Definition 23 *For an event expression Exp,* $N_{nest}(Exp) = \alpha(Exp)$ *where* α *is defined in Definition 22*

Theorem 1 *An event expression q is in a normal form iff* $N_{nest}(q) = 0$.

Proof Sketch: If N_{nest}(q) = 0, then α(q) = 0. If a subexpression Exp_i in q is expressed by SEQ, Exp_i doesn't include subexpression in cases [4-12] which would make N_{nest}(q) > 0. In cases [13-14], for $1 \leq i \leq n$, Exp_i shouldn't be expressed by SEQ, AND or OR which would make N_{nest}(q) > 0. Thus Exp_i belongs to the cases [1-3] in Table 5.2.8 with SEQ as the outer operator. SEQ(E_1 ,..., ! E_i ,..., E_n) is in a normal form. If Exp_i is expressed by AND, Exp_i doesn't include subexpressions in cases [5-12] which would make N_{nest}(q) > 0. In cases [13-14], for $1 \leq i \leq n$, Exp_i shouldn't be expressed by SEQ, AND or OR which would make N_{nest}(q) $>$ 0. Thus Exp_i belongs to cases [1-4] in Table 5.2.8 with AND as the outer operator. Exp_i could be AND(E_1 ,..., ! E_i ,..., E_n) and AND(SEQ(E_1, E_2) ,..., E_i ,..., E_n) which are in a normal form. If Exp_i is expressed by OR, Exp_i doesn't include the subexpression OR(OR) which would make N_{nest}(q) > 0. Exp_i could be expressed by OR(AND) and OR(SEQ) but no other operators should exist inside AND and SEQ which would make N_{nest}(q) > 0. OR(AND(E_1 ,..., E_n), SEQ(E_1 ,..., E_n)) is in DNF. Thus the event expression q is in a normal form.

If q is in a normal form, namely, CNF (see Definition 21), DNF (see Definition 20), then α (q) = 0. Thus N_{nest}(q) = 0.

We have proven an event expression q is in a normal form iff N_{nest}(q) = 0. □

.

Theorem 2 *Rewriting decreases $N_{nest}(q)$.*

Proof Sketch: First, we show that N_{nest}(q) is decreased after each successfully applied rewriting step. In Table 5.1.4, for FR1 and FR2, α(q) is decreased by 1 as the inner SEQ is removed. Similarly, for FR3 and FR4, α(q) is decreased by 1 as the inner AND is removed. For FR5, α(q) is decreased by 1 as the inner OR is removed. For FR6, α(q) is decreased by 2 as the inner $\exists$ SEQ is removed. For FR7, α(q) is decreased by 2 as the inner $\exists$ AND is removed. For DR1 and DR2, α(q) is decreased by 1 as the inner OR is removed. Similarly, For DR3 and DR4, α(q) is decreased by 1 as the inner $\vee$ is removed. For DR5 and DR6, α(q) is decreased by 1 as the inner OR is removed. For NPDR1 and NPDR2, α(q) is decreased by 1 as ! SEQ is removed with $\exists$ SEQ introduced. For NPDR3 and NPDR4, α is decreased by 1 as ! AND is removed with $\vee$ introduced. For NPDR5, α is decreased by 1 as ! OR is removed with $\wedge$ introduced. □

Theorem 3 *For q satisfying our language constraints in Section 5.2.2, if N_{nest} (q) $>$ 0, then q can be rewritten.*

Proof Sketch: N_{nest}(q) = α(q). Given α(q) $>$ 0 with q satisfying our language constraints expressed by Class Lcons in Table 3.5, I will show q can be rewritten and the expression after rewriting q is still of Class Lcons. Table 5.2.8 covers all possible subexpression cases. If q is expressed by SEQ as the outer operator, q may contain the following expressions: ! SEQ(primitive event types) (rewritten by NPDR), !SEQ(OR) (rewritten by DR and NPDR), ! SEQ(SEQ) (rewritten by FR and NPDR), SEQ(SEQ) (rewritten by FR), SEQ(OR)(rewritten by DR), SEQ($\exists$ SEQ) (rewritten by FR), SEQ(! SEQ) (rewrite by NPDR, DR and FR), SEQ($\exists$ OR) (rewritten by DR), SEQ($\exists$ SEQ $\vee$ $\exists$ OR) (rewritten by DR and FR).

If q is expressed by AND as the outer operator, q may contain the following expressions: AND(AND) (rewritten by FR), AND(!AND) (rewritten by NPDR and DR), AND(SEQ) (in CNF), AND(OR)(rewritten by DR), AND($\exists$ OR) (rewritten by DR), AND($\exists$ AND) (rewritten by FR), AND(! OR) (rewritten by NPDR), AND(!AND) (rewritten by NPDR), AND($\exists$(!) AND $\vee$ $\exists$ (!) OR) (rewritten by DR, FR and NPDR)).

If q is expressed by OR as the outer operator, q may contain the following expressions: OR(SEQ)(in DNF), OR(AND)(in DNF), OR(OR)(rewritten by flattening rule).

For all the above expression rewriting, no ! SEQ(!) and SEQ(AND) are introduced in each rewriting step. Namely, after rewriting, the above expressions are still within scope of Class Lcons. □

Theorem 4 *If an event expression q satisfies our language constraint, q can be rewritten into a normal form.*

Proof Sketch: Given q let q_0, q is rewritten into q_0 by several steps and q_0 cannot be rewritten. By Theorem 3, we have N_{nest}(q) = 0. By Theorem 1, q is in a normal form q_0. □

5.3 Shared Optimized *NEEL* Pattern Execution

Once a normalized expression has been constructed by our rewriting procedure described in Section 5.2.7, multiple sharing opportunities among subexpressions have been exposed. Below, we introduce the strategies we have designed for subexpression sharing among query conjuncts, disjuncts and leaf components[2] in the normalized forms defined in Definitions 20 and 21.

5.3.1 Subexpression Sharing

Sharing with Prefix Caching. First, expressions with a common prefix can share the same cached prefix results. It is wasteful for sequence construction to traverse the same set of stacks repeatedly. Thus the prefix caching method is designed to cache such results in the *PreCache*. This enables future sequence construction involving the same set of stacks to reuse these cached results. The common prefix is computed first before computing each expression. The buffered result e can be deleted after an event e_i with e_i.ts - e.ts $>$ window w is received.

[2] In the query plan expressed by a nested AND/SEQ expression, we call the bottommost event expressions *leaf components*.

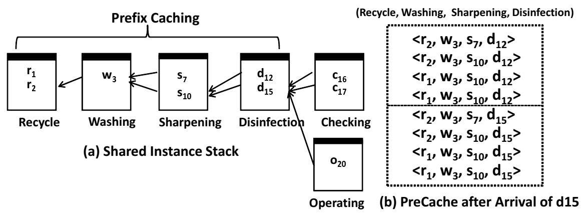


Figure 5.3: Prefix Caching Example

Example 27 *Assume we get a disjunctive normal form with two conjuncts E_1 = SEQ(Recycle, Washing, Sharpening, Disinfection, Checking) OR E_2 = SEQ(Recycle, Washing, Sharpening, Disinfection, Operating). Their common prefix is SEQ(Recycle, Washing, Sharpening, Disinfection). To avoid re-constructing results for the common prefix, such shared results (ordered by end timestamps) are stored in PreCache as shown in Figure 5.3. E_1 and E_2 results can then be computed simply by joining the results in the PreCache with events in Checking and Operating stacks respectively.*

Sharing with Suffix Clustering. Since event traversals for result construction typically start from events of the last event type in a pattern [WDR06, CHC$^+$06], shared suffices also eliminate redundant event traversals. Queries sharing the same suffices would then be evaluated concurrently by processing their shared suffices until the common part has been treated. Thereafter, each query is finished up by joining the suffix results with other events in the respective query to form final results.

Example 28 *Assume we get a conjunctive normal form with two disjuncts E_1 = SEQ(Recycle, Washing, Sharpening, Disinfection, Checking) AND E_2 = SEQ(Operating, Washing, Sharpening, Disinfection, Checking). Figure 5.4 shows the stacks shared*

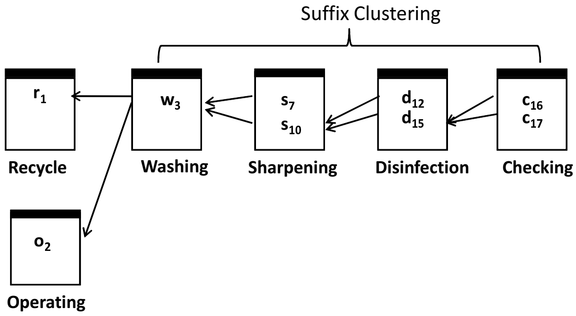


Figure 5.4: Suffix Clustering Example

among E_1 and E_2. Once the event c_{16} or c_{17} of type Checking arrives, the shared result construction for the suffix sub-pattern (Washing, Sharpening, Disinfection, Checking) is initiated.

Sharing among queries with **shared middle sub-expressions** can be similarly achieved. Results for such middle sub-expressions should be pre-computed and cached. Again, such cached results may need to be joined with other events that exist in the respective query to form final results.

5.3.2 Advanced Sub-expression Sharing with Different Negative Components

Beyond prior work [WDR06, BDG$^+$07, MM09], we now also tackle the case of sub-expression sharing with different negative components. Namely subpatterns contain the same projected positive event types while their negative event types may differ. Besides saving CPU resources, we achieve the added benefit that one sequence result may satisfy several such expressions. If we construct the results for such normalized event expressions of a nested query separately, we may inadvertently produce duplicate results namely one for each of these different event expressions. This then would not only waste CPU resources for re-computation but also incurs the costs associated with duplication removal.

We observe that such event expressions with common positive event types return the same results yet only apply different negation filters. The main idea is that we record the constraints of non-occurrence and non-projected occurrence for each expression at compile time. At run time, as we construct each sequence result, we keep track of which of the given constraints are satisfied (or, rather violated). We stop the evaluation early for unsatisfied event expressions.

Expression-vs-Negative Map (EMap). To facilitate the advanced sequence result generation, we design a data structure *EMap* that records the negative components and non-projected positive components of an expression with their positions. Columns in the map correspond to negative components and non-projected positive components with positions in the shared expressions while rows list the expression identifiers. If the same negative component or non-projected positive component exists in different positions in an expression, such negative component is listed multiple times in *EMap*. At compile time, a cell entry indicated by its row and column Map[i, j] is assigned a "1" if the negative event type as indicated by column j is listed in the specified position in an expression E_i and a "0" otherwise. Possibly one negative component may exist in more than one location in different queries.

Result Vector Indicator (RVI). For each partial sequence result, we maintain a *Result Vector Indicator (RVI)* which is represented by a bit array. The columns of RVI are the same as the ones in *EMap*. During query execution, a *RVI* is maintained to check if the current partial result is indeed a correct match. We mark the cell entry <i, j> for a column that corresponds to a negative component or a non-projected positive component as "1" if at run time the negative component or the non-projected positive component assigned with that column evaluates to true in the specified position in an event stream (not found for the negative component and found for the non-projected positive component).

Lemma 3 *We stop query evaluation early for one sub-expression E_i if logical AND-ing the bit vectors of the row for E_i in EMap with the RVI for the partial result is "0".*

Proof: When the logical AND-ing of the bit vectors of the row for E_i in *EMap* with the *RVI* for the partial result is "0", as the bits in *EMap* are all "1", it indicates at least one bit in *RVI* is "0". So we can conclude that at least either one negative component is evaluated to false (found) or one non-projected positive component is evaluated to false (not found). According to the semantics of SEQ operator with negation 5.4, such partial result is not satisfied. □

Example 29 *The normalization procedure rewrites Q_1 = SEQ(Recycle, Washing, ! SEQ(Sharpening, Disinfection, Checking), Operating) into the expression in Figure 5.5. Figure 5.6(a) shows the shared instance stacks for all three expressions. Figures 5.6(b) and 5.6(c) show the $EMap$ and RVI structures respectively. The negative component for E_1 is ! Checking, for E_2 (! Disinfection, Checking) (Checking is not a positive component as it is not listed in the projection list) and for E_3 (! Sharpening, Disinfection, Checking). When event instance o_{20} of type Operating arrives, the sequence construction is initiated. When evaluating the partial result $< w_5, o_{20} >$, we mark the cell "1" under (! S, D, C) in RVI as $< d_6, c_{16} >$ exists between w_5 and o_{20} and no Sharpening events s_i with $5 < i < 6$ exist. Similarly, the (! D, C) AND (! C) cells are marked with "0". The partial result $< w_5, o_{20} >$ can* *continue the result construction for E_3 because the AND of the bits in the result vector RVI in Figure 5.6 (c) with the row for E_3 in the EMAP in Figure 5.6 (b) is "1". Result computation for E_1 and E_2 stopped early by Lemma 3 because the AND of such bits is "0".*

SEQ(Recycle, Washing, ! Checking, Operating) OR
$Proj_{R, W, O}$ SEQ(Recycle, Washing, ! Disinfection, Checking, Operating) OR
$Proj_{R, W, O}$SEQ(Recycle, Washing, ! Sharpening, Disinfection, Checking, Operating)

Figure 5.5: Normalized Expression for Q1

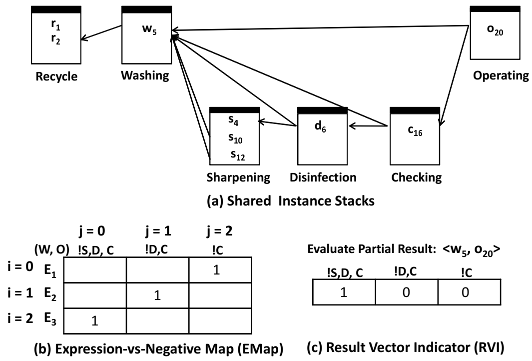


Figure 5.6: Bit-Marking Example

Lemma 4 *No duplicate results will be produced because we conduct sequence construction only once for all expressions in a group.*

Proof: We will output a sequence result for a group of shared expressions S if and only if $\exists$ E_i in S for which the logical bit by logical AND-ing the bit vectors of the row for the sub-expression E_i with the current result's *RVI* is "1". Each sequence result is only outputted once for a group of shared expressions. It implies that all the non-existence constraints in at least one of the clustered expressions are satisfied. □

The pseudo-code for the shared logic bit-marking based sequence construction strategy is presented in Figure 5.7. Given flattened event expressions (query disjuncts/conjuncts/leaf components) with the same positive components and one or more different negative components, *EMap* is first constructed. Then, we conduct the sequence construction process for every event instance e_j of the accepting state in the rightmost stack, traversing back along the event pointers. During sequence construction, *RVI* is filled for each partial sequence result to conduct the sequence validation process. We compare the *RVI* of each partial result with each row of *EMap* continuously after evaluating each negative component or each non-projected positive component. We stop or continue the sequence construction for each partial result based on Lemmas 3 and 4.

SequenceCompute Algorithm: output sequence results

Boolean *out* ← *true*;
while (*out* ∧ *stackIndex* != *0*) **do**
    Sequence s = Connect(SConstruction(), s); *// Recursively call sequence construction until the first stack is reached.*
    RVI rvi = BitMarking(); *// Mark jth cell "1" if RVI(j) holds true.*
    out = SequenceValidation(rvi); *// Check filled result vector with EMap.*
    stackIndex –;
end while

Figure 5.7: Sequence Compute with Run-Time Bit Marking

5.4 Plan-Finder

When a set of normalized CEP expressions S share some of the same positive components, several options may arise for grouping them to obtain better shared execution plans. Consider for example the normalized expression S = SEQ(A, B,

D) OR SEQ(A, B, ! C, D) OR $Proj_(A,B,D)$SEQ(A, B, ! E, C, D) OR SEQ(A, B, D, E, F) OR SEQ(A, B, D, E, G). The first three conjuncts share the same projected pattern SEQ(A, B, D). The bit-marking algorithm in Section 5.3.2 could be applied to them. Or, alternatively, the first and the last two conjuncts also share the common prefix SEQ(A, B, D). Prefix caching as in Section 5.3.1 could be applied to them. We must make a good choice among these options in the plan space.

5.4.1 Problem Definition of Finding Shared-Plans

Given a set of normalized CEP expressions S, an expression partition $P_i = \{g_1, g_2$,..., $g_i\}$ satisfies the following constraints:

- Full coverage: $\forall$ expression E_j in S, $\exists$ g_i that $E_j \in g_i$;
- Non-overlapping: $\forall$ g_i, g_j, $g_i \cap g_j = \emptyset$;
- Each group g_i is mapped to one shared physical operator in Section 5.3, i.e., each g_i is implementable.

A partition P_i is valid if it satisfies full coverage and non-overlapping constraints. We aim to find an expression partition P_i with the minimum execution cost among all possible partitions. Based on our cost analysis for nested and flattened execution plans [LRG+10c], the Plan-Finder constructs an optimized execution strategy for the normalized form as defined by Definitions 20 and 21 by selecting among possible alternatives.

5.4.2 Plan-Finder Search Space

We now analyze how many possible partitions the Plan-Finder would have to enumerate through to find the best one. To find an optimal solution requires us to enumerate all possible expression partitions. The *Bell number* [Kla03], or the number of different *partitions* P_i of a set S of n elements, describes the size of such a search space, i.e., the total number of all possible partitions for a set of expressions. The problem is challenging, as the complexity of the Plan-Finder *O(B_n)* is exponential as shown in Equation 5.59 where B_n represents the upper-bound of all possible multi-route configurations for the set T. The *Stirling number* $S(n,k)$ in B_n is the number of the partitions of n with exactly k blocks.

$$B_n = \sum_{k=1}^{n} S(n,k) = \sum_{k=1}^{n} \left(\frac{1}{k!} \sum_{j=1}^{k} (-1)^{k-j} \binom{n}{k} j^n \right) \tag{5.59}$$

5.4.3 Plan-Finder Search Algorithms

Due to the prohibitive exponential complexity of the search space, we adopt a cost-based heuristic for finding a good quality solution in reasonable time without enumerating the entire search space. While many heuristics are possible, below we sketch one using an iterative refinement methodology:

Selecting a Start Solution. We adopt the strategy to group all event subexpressions with the same projected event types into one group to achieve aggressive sharing; though other start heuristics are possible.

Search Strategy: We adopt the iterative improvement method due to its simplicity (see pseudocode in Figure 5.8). A single basic transformation (e.g., a split of a group or merge of two groups) would transition from a partition solution P_i to its neighbor P_j. g_i represents a group in the start partition solution. e.g., "$g_1g_2/g_3/g_4$" $\rightarrow$ "$g_1/g_2/g_3/g_4$" represents a split of two groups g_1 and g_2 while "$g_1/g_2/g_3/g_4$" $\rightarrow$ "$g_1g_2/g_3/g_4$" represents a merge of two groups g_1 and g_2.

Selecting a Stop Condition: In general, the search may stop when either k iterations have gone by, or the solution did not improve in the last several rounds, i.e., the search process reaches a plateau. Alternatively, the search can be bounded by resources such as time.

Plan-Finder Algorithm: output best plan

```
partition ← start solution; best-partition ← start solution;
while (not stop condition) do
   while (not local_minimum(partition)) do
      partition' ← find random solution in NEIGHBORS(partition)
      if (cost(partition') < cost(partition)) then
         partition ← partition'
      end if
   end while
   if (partition.cost < cost(best-partition)) then
      best-partition ← partition
   end if
end while
return best-partition;
```

Figure 5.8: Plan-Finder Algorithm

5.5 Performance Evaluation

The primary objective of our experimental evaluation is to study the accumulative CPU processing time of the traditional iterative nested execution [LRR+10] and our proposed optimized *NEEL* execution strategy with different workloads.

5.5.1 Experimental Setup

We have implemented all strategies within the HP stream management system CHAOS [GWA$^+$09b] using Java. We ran the experiments on Intel Pentium IV CPU 2.8GHz with 4GB RAM. We evaluated our techniques using the real stock trades data from [sto]. The data contained stock ticker, timestamp and price information. The portion of the trace we used contained 10,000 unique event instances. We used sliding windows with a size of 10ms. In our experiments, the y axis denotes the CPU processing time. CPU processing time means the wall clock time for processing an item e_i in stock trades measured by ($T_{end.ei}$ - $T_{start.ei}$) where $T_{start.ei}$ represents the system time when our processing engine starts processing the data item e_i and $T_{end.ei}$ represents the system time when the engine finishes processing the data item e_i. It is an atomic process, i.e., our processing engine won't stop processing that tuple until it is fully processed.

5.5.2 Experimental Design Query Plans

We first evaluate queries by varying three parameters as shown in Figures 5.9, 5.10 and 5.11. In Figures 5.9, the number of sub-queries is increased from 1 to 3. In Figure 5.10, we then keep the sub-query number as 1 and increase the sub-query length from 2 to 4. In addition, in Figure 5.11 we keep the number and the length of sub-queries the same and we change sub-query nesting levels from 1 to 3. Lastly, we evaluate our system with one complex workload in Figure 5.12.

We have implemented all strategies within the stream management system CHAOS [GWA$^+$09b] using Java. We ran the experiments on Intel Pentium IV CPU 2.8GHz with 4GB RAM. We evaluated our techniques using the real stock

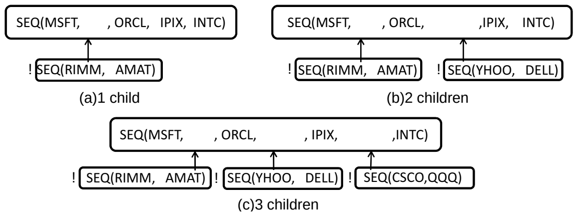


Figure 5.9: Sample Queries with Increased Children Number

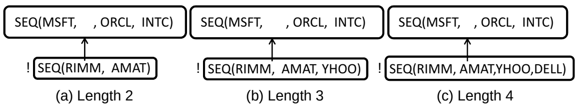


Figure 5.10: Sample Queries with Increased Query Length

trades data from [sto]. The data contained stock ticker, timestamp and price information. The portion of the trace we used contained 10,000 unique event instances. The arrival rate was set to 4,000 tuples/sec. We used sliding windows with a size of 10ms.

5.5.3 Varying the Number of Children Queries

The first experiment studied queries with increasing numbers of sub-queries as depicted in Figure 5.9. In Figure 5.14, we observe that our proposed optimized *NEEL* execution runs on average 5 fold faster than the more traditional nested execution. In the optimized *NEEL* execution, we don't need to compute results for SEQ(*RIMM*, *AMAT*), SEQ(*YHOO*, *DELL*) and SEQ(*CSCO*, *QQQ*). In Figure 5.15, we observe that in the nested execution, most of the time is used for computing children query results because for each outer partial result, we need to compute children results. This observation also holds true for queries used in

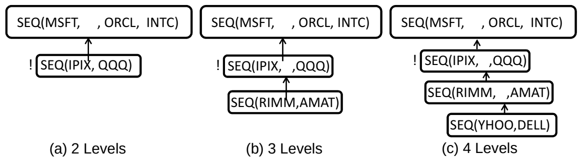


Figure 5.11: Sample Queries with Increased Nesting Levels

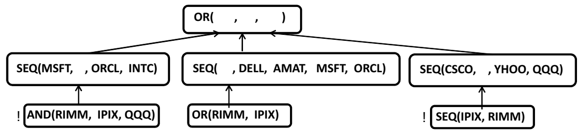


Figure 5.12: Complex Workload

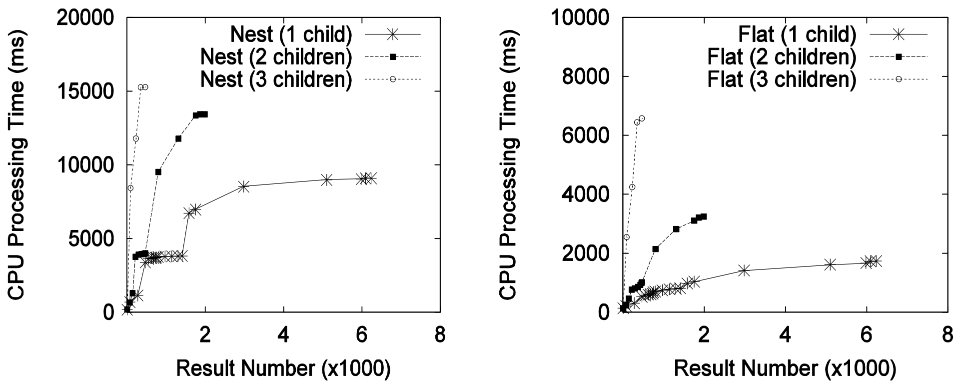
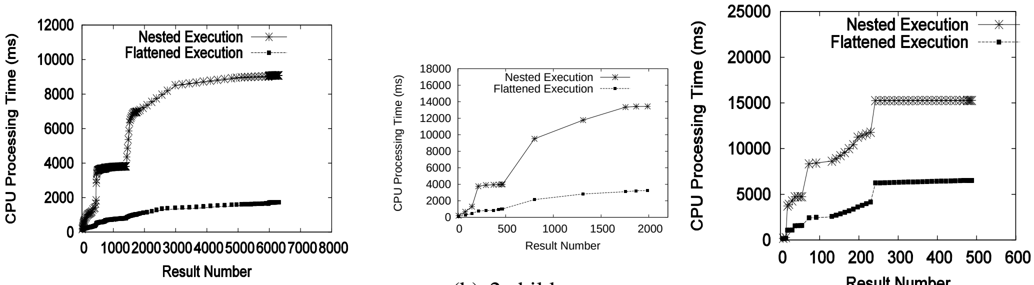


Figure 5.13: Nested and Flattened Execution with Increased Children Number

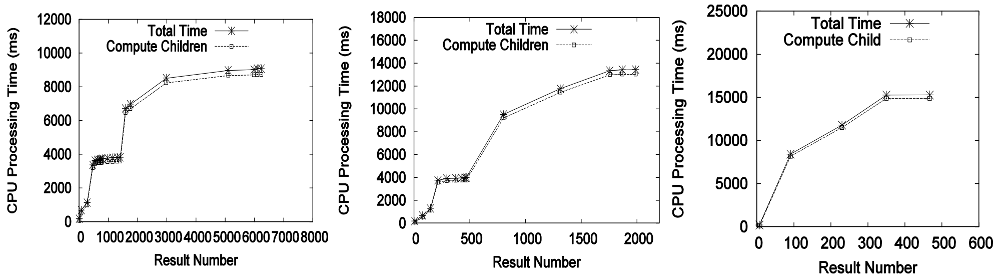
Figures 5.10 and 5.11.

Next, we compare the CPU processing times among the queries in Figure 5.9 with results shown in Figure 5.13. We observe that the query with 3 children generates the least number of results for both nested and flattened execution, because it has more constraints and more outer SEQ($MSFT$, $ORCL$, $IPIX$, $INTC$) results are filtered in the nested execution. In addition, the query with 3 children uses the most CPU processing time among the three queries because of processing



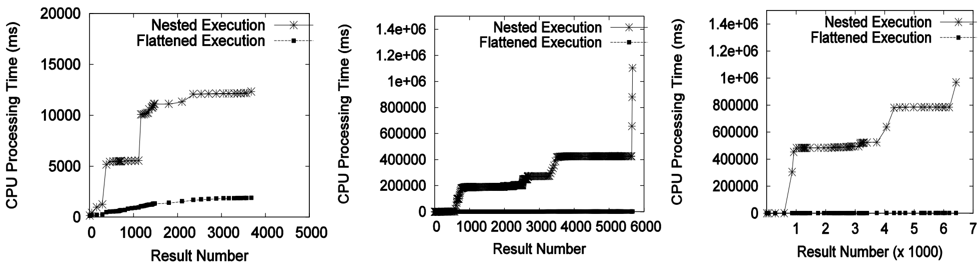
(a) 1 child (b) 2 children (c) 3 children

Figure 5.14: Varying the Number of Children Queries



(a) 1 child (b) 2 children (c) 3 children

Figure 5.15: Comparing Total Computation Time vs. Children Computation Time in Nested Execution with Increased Children Number



(a) Length 2 (b) Length 3 (c) Length 4

Figure 5.16: Varying the Length of Children Queries

more sub-queries. This consumes more CPU processing time. These results match our expectation as clearly the computation time increases with the number of sub-queries and also the probability of finding patterns decreases with an increasing number of event types, i.e., query constraints.

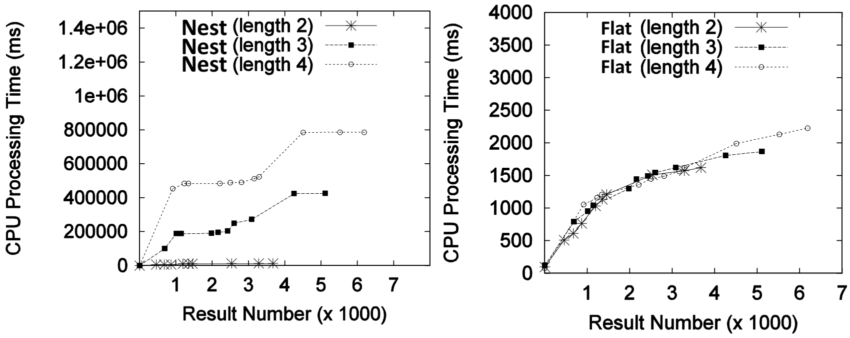
5.5.4 Varying the Length of Children Queries

This second experiment processes the queries depicted in Figure 5.10 with subquery lengths varying from 2 to 4. Results are shown in Figure 5.16. We observe that our proposed optimized *NEEL* execution runs on average several hundreds fold faster than the more traditional nested execution. In the flattened execution, we don't need to construct the children query results for SEQ($RIMM$, $AMAT$), SEQ($RIMM$, $AMAT$, $YHOO$) and SEQ($RIMM$, $AMAT$, $YHOO$, $DELL$).

Next, we compare the CPU processing time among queries in Figure 5.10 with results shown in Figure 5.17. The subquery with length 4 generates the largest number of results. As expected, it has less outer SEQ($MSFT$, $ORCL$, $INTC$) results filtered as the existence of a longer pattern is relatively less likely as compared to the other queries with shorter patterns. In addition, it uses the most CPU processing time among the three queries because it includes the sub-query with the longest length which consumes more computational processing resources.

5.5.5 Varying the Nesting Levels of Children Queries

The third experiment processes queries with varying sub-query nesting levels (Figure 5.11). Results are shown in Figure 5.18. Our proposed optimized *NEEL* execution consistently takes less time as compared to nested query execution. It is because the flattened execution doesn't need to construct the children query results for SEQ($IPIX$, QQQ), SEQ($RIMM$, $AMAT$) and SEQ($YHOO$, $DELL$). Thus significant CPU processing resources are saved.

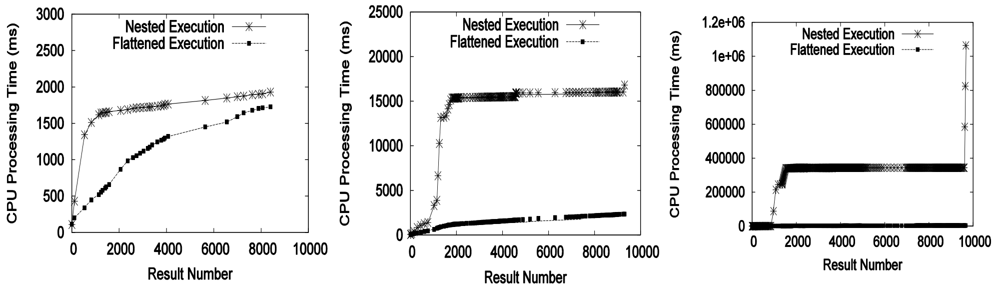


(a) Nested Execution

(b) Flat Execution

Figure 5.17: Varying the Length of Children Queries

Next, we compare the CPU processing time among queries in Figure 5.11 with results shown in Figure 5.19. The query with the largest nesting levels generates the most number of results and uses the most CPU processing time among the three queries for both nested and flattened execution. It is because the query includes the sub-query with the largest nesting levels which consumes more time to be computed. In the nested execution, less outer SEQ($MSFT$, $ORCL$, $INTC$) results are filtered as to filter one result, we need to at least find a sequence satisfying more constraint.

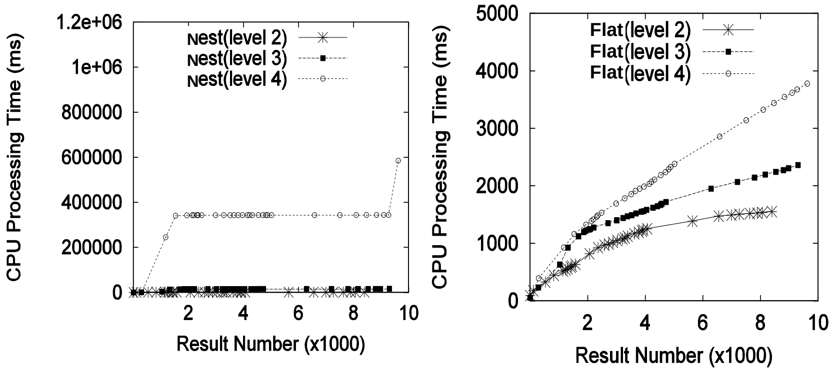


(a) Level 2

(b) Level 3

(c) Level 4

Figure 5.18: Varying the Levels of Children Queries



(a) Nested Execution

(b) Flat Execution

Figure 5.19: Varying the Levels of Children Queries

5.5.6 Complex Workload

The last experiment processes the complex query in Figure 5.12. The normalized expression $E = E_1$ (SEQ(MSFT, ! IPIX, ORCL, INTC)) OR E_2 (SEQ(MSFT, ! QQQ, ORCL, INTC)) OR E_3 (SEQ(MSFT, ! RIMM, ORCL, INTC)) OR E_4 (SEQ(RIMM, DELL, AMAT, MSFT, ORCL)) OR E_5 (SEQ(IPIX, DELL, AMAT, MSFT, ORCL)) OR E_6 ($Proj_{CSCO,YHOO,QQQ}$ SEQ(CSCO, ! RIMM, YHOO, QQQ)) OR E_7 ($Proj_{CSCO,YHOO,QQQ}$ SEQ(CSCO, ! IPIX, RIMM, YHOO, QQQ)). The partition returned by the planFinder is $\{[E_1, E_2, E_3], [E_4, E_5], [E_6, E_7]\}$. [$E_1$, E_2, E_3] is mapped to the operator in Section 5.3.2 as these subexpressions share the same positive event types (MSFT, ORCL, INTC) while the negative event types are different. Similarly, [E_6, E_7] is also mapped to the operator in Section 5.3.2. [E_4, E_5] is mapped to the operator in Section 5.3.1 as they share the same suffix (DELL, AMAT, MSFT, ORCL). As expected, our proposed *NEEL* execution takes less time as compared to iterative nested execution as shown in Figure 5.20.

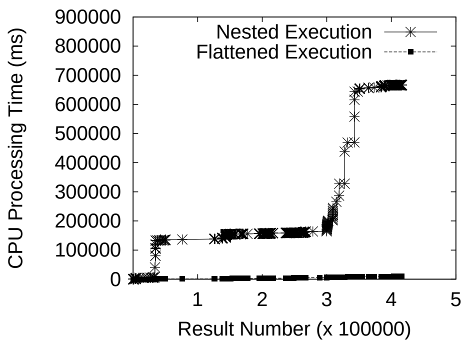


Figure 5.20: Complex Workload

5.6 Discussion: Query Decorrelation

Complex SQL queries used in decision support applications often include correlated subqueries. SQL queries may contain multiple correlated subqueries, possibly across several levels of nesting. Their efficient execution is important. In this section, we will review the state-of-the art in query optimization via decorrelation. And we will briefly discuss its applicability to nested CEP queries.

5.6.1 Correlated Query Example

The sample query Q_1 is an example of correlation based on the employees and departments. Q_1 finds young employees who are paid more than the average salary in their department. Each SELECT-FROM-WHERE component is a query block. The column E.did used inside the nested subquery block is drawn from the outer enclosing query block. A nested query block is correlated if it uses a value from an enclosing query block.

```
Q1 = SELECT *
```

```
FROM Emp E
WHERE E.age < 30
AND E.sal > (SELECT AVG(E1.sal)
             FROM Emp E1
             WHERE E1.did = E.did)
```

A subquery can be either aggregate or non-aggregate. An aggregate subquery has an aggregate function in its SELECT clause; it always returns a single value as the result. A non-aggregate subquery is linked to the outer query by one of the following operators: EXISTS, NOT EXISTS, IN, NOT IN, θ, SOME/ANY, and θ ALL, where $\theta \in \{<, \leq, >, \geq, =, \neq\}$; the result is either a set of values or empty.

5.6.2 Decorrelation

Due to the perceived inefficiencies in Nested Iteration, techniques have been proposed to avoid the tuple-at-a-time evaluation imposed by nested iteration [SPL96]. A correlated SQL query is transformed into an equivalent query that is no longer correlated. This process is called decorrelation. Significant research efforts have been devoted to the optimization of nested queries.

Logic of Decorrelation As pointed out in [SPL96] based on this decorrelation technique, any correlated subquery block can be modeled as a function CS(x) whose parameters x are the correlation values. In the sample query Q_2, the correlated subquery is a function that uses the value E.did as a parameter, and returns a table containing a single tuple, which holds the average salary in that department. The evaluation of the outer query block using Nested Iteration can be represented by the following pseudo-code.

```
precomputation...;
for each (x in X) {
          SubQueryResult = CS (x);
          Process(SubQueryResult);
          }
postcomputation ...;
```

where X represents the set of values with which the correlated subquery is invoked. The precomputation and postcomputation represent the portions of the evaluation before and after the region of interest to this discussion. The purpose of decorrelation is to overcome the drawbacks of Nested Iteration; to eliminate duplicate invocations of the subquery with identical correlation values and to reduce the redundant work done in each subquery invocation using set-oriented techniques, and to minimize the interference between the computation of the outer query block and the subquery block [Ses98]. Decorrelation can decouple the execution of CS from the execution of the outer query block. The following is described by the authors in [Ses98]:

"Consider some set X_1, such that $X \in X_1$. Obviously, $(x \in X)$ implies $(x \in X_1)$. Let us define a new table DS (i.e. "Decoupled Subquery) such that DS = $\{$(x,y) $|$ $x \in X_1 \wedge$ y $\in$ CS(x) $\}$. In other words, DS computes CS(x) for all values x in X_1. "

Now consider the following version of the pseudo-code of the outer block evaluation:

```
precomputation ...;
determine X1;
compute DS using X1;
```

```
for each (x in X) {
          SubQueryResult = {y1 |(x1, y1) in DS and x = x1 };
          Process(SubQueryResult);
          } The computation of DS is decoupled from that of
            the outer block.
postcomputation ...;
```

The condition x = x1 maintains the correlating relationship between the value of x in each pass through the loop, and the values selected from DS during that pass. It is easy to prove that the modified outer block produces the same answers as the original query block, as long as computing CS(x) and DS does not change any data in the rest of the system. This abstraction represents the basic idea behind all decorrelation algorithms. Compare this modification of the query evaluation with nested iteration [Ses98]:

- Since DS is computed using a set of X_1 of parameters of interest, there are no duplicate invocations, thereby resulting in a performance improvement.
- Since the entire set X_1 is available, the computation of DS can use efficient set-oriented techniques that reduce the amount of redundant work performed, thereby improving performance.
- The computation of DS is decoupled from that of the outer block. Consequently, there is no interference between the two.

5.6.3 Magic Decorrelation

The basic idea is to rewrite a correlated query in such a way that outer references no longer exist in the inner subquery. All the possible results from the sub-query are materialized. Later, the materialized results are joined with the outer query block on the outer reference values.

The result of applying Magic Decorrelation to the example query Q2 is shown as below. The steps are then explained in detail.

```
View Definitions
CREATE VIEW PreComputation AS
  (SELECT E.eid, E.sal, E.did
   FROM Emp E, Dept D
   WHERE E.did = D.did AND E.age < 30
   AND D.budget >100, 000)

CREATE VIEW FILTER_X1 AS
  (SELECT DISTINCT P.did
   FROM PreComputation P);

CREATE VIEW DecorrSubQuery_DS AS
  (SELECT F.did, AVG(E1.sal) as avgsal
  FROM Filter_X1 F, Emp E1
  WHERE E1.did = F.did
  GROUPBY F.did);
```

```
Outer Query Block
SELECT P.eid, P.sal
FROM PreComputation P, DecorrSubQuery_DS V
WHERE P.did = V.did
       AND P.sal > V.avgsal
```

The PreComputation table represents the computation in the outer query block until the point that the subquery invocations begin. The Filter-X1 table represents the (duplicate-free) set X1 of correlation values with which the subquery will be invoked. **SELECT DISTINCT** is used to **eliminate duplicates**. DecorrSubQuery-DS is the table generated by decorrelating the subquery using the Filter-X1 table. It contains one tuple per value of F.did (i.e., one tuple per correlation value). Note that the Filter-X1 table has been added to the FROM clause of the original subquery. That is, the nested dependencies has now been replaced by a join. Finally, in the outer query block, the preComputation table P is joined with the decorrelation subquery to form the rest of the post-computation, and produce the desired answers. The join predicate P.did = V.did enforces the correlating relationship.

The following Set(X), X1 and DS are described in Section 5.6.2.

1. Set(X) is computed and used as X1; obviously, there will be no unnecessary subquery computation.

2. DS is computed by adding X1 to the FROM clause of the original correlated subquery and converting the predicate using the correlation value to a join predicate.

3. The correlating relationship between the computation in the outer query block and the answers in DS is enforced by adding DS to the FROM clause of the outer query block and adding an equi-join predicate on the correlation values.

Query Graph Model. In IBM DB2, queries are internally represented in a Query Graph Model (QGM). The goal of QGM is to provide a conceptually more manageable representation of queries in order to reduce the complexity of query compilation and optimization.

Terms. A box B is **directly correlated** to box A, if B contains a correlation that references a column col from a table in the FROM clause of A. The column col is said to be the **correlation column**. A box C is (recursively) said to be correlated to box A, if C or one of Cs descendants is directly correlated to box A. For example, in Figure 5.21, Box (3) is directly correlated to Box (1) as it uses the input from (1). Box (3) and Box (2) are said to be correlated to Box (1) because at least one of the descendants of (3) and (2) are directly correlated to (1). q1.Building is the correlation column. We traverse the QGM in depth first order. For our example, visit the boxes in the order (1), (2), (3).

Each Box has a head and a body. **Head** is a declarative description of the output with schema (list of output columns) and property. **Body** specifies how to compute the output. The body of a box contains a graph. The vertices of this graph represent quantified tuple variables or quantifiers: F represents a regular tuple variable, e.g., FROM R AS r. E represents an existential quantifier, e.g., IN (subquery), or = ANY (subquery). SQL's predicate EXISTS, IN, ANY and SOME are true if at least one tuple of the subquery satisfies the predicate. The quantifiers associated with such subqueries have type E. A represents the universal quantifier, e.g., $>$ ALL (subquery) and S represents a scalar subquery, e.g., = (subquery). The body of every box has an attribute called *distinct* which has a value of ENFORCE, PRESERVE, or PERMIT. ENFORCE means that the operation must eliminate duplicates in order to enforce head.distinct = TRUE. PRESERVE means that the operation preserves the number of duplicates it generates. This could be because head.distinct = FALSE, or because head.distinct = TRUE and no duplicates could exist in the output of the operation even without duplicate elimination. PERMIT means that the operation is permitted to eliminate (or generate) duplicates arbitrarily.

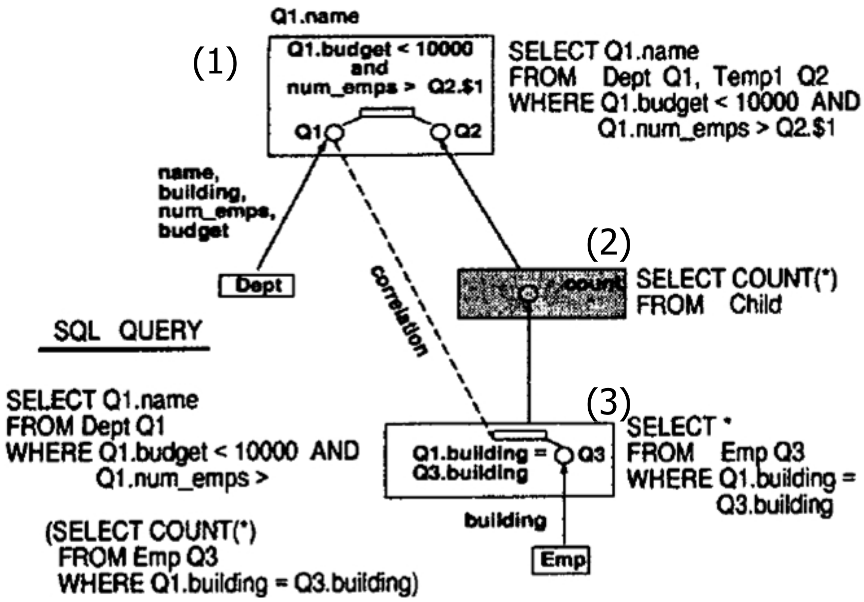


Figure 5.21: QGM Graph Example

Example 30 *We perform the decorrelation by a top-down traversal of the QGM tree as shown in Figure 5.21. For each box, it looks at its iterators (inputs to the box) in some order. It checks whether the iterator is correlated, and if so, whether it can be decorrelated. This decorrelation for the (box, iterator) is done in two steps. In the FEED step, a set of bindings that the subquery (iterator) needs are generated* *and these bindings are now used by the subquery. As pointed out in [SPL96], when the rewrite rule is applied to the subquery (i.e. when the subquery is treated as the CurBox), it decorrelates the subquery using the correlation values. This is called the ABSORB stage because the subquery absorbs the correlation bindings resulting in a decorrelated query.*

Removing Decorrelation. *We first visit box (1). It has a descendant box, that is correlated to it. So we perform the feed step on Box (1) is not correlated to an ancestor box, so there is no absorb. Let us see how feed for box (1) is performed.* **Feed for Box (1).** *Check if there is any condition on the "correlation" column in Box (1). If yes, push the selection condition before Box (1) (see Figure 5.46). Create another box, which removes duplicate values of the correlation column (see Figure 5.47). Create 2 boxes as in Figure 5.48. DCO (Decorrelated Output) box takes the above values as input while box (3) will now depend on this box. CI (Correlated Input) box takes output of DCO box, is correlated to Box (1) and performs the equi-join.* **Decorrelating Box (2).** *Box (3) is correlated to the parent DCO box of Box (2). So we perform the feed (see Figure 5.26). Push select conditions. In this case here, we have none. Next, we need to remove duplicates if any. In this case here, we have none. Create a DCO box and a CI box. Box (2) is correlated to its parent DCO box. So we perform the absorb (see Figure 5.27). For an aggregate operator, absorb includes a group by, followed by a LOJ. In this case, we end up with an unnecessary CI box. Remove it (see Figure 5.28).* **Decorrelating Box (3).** *There is no descendant box that is correlated to box (3) or its ancestor exists. Therefore, no feed. Box (3) is correlated to its parent DCO box. So we perform the absorb (see Figure 5.30). Absorb for SPJ box means just remove the correlation, and feed the box directly as input to the SPJ box. Remove unnecessary Q_8 input to*

DCO box (see Figure 5.31). Remove unnecessary DCO box (see Figure 5.32).

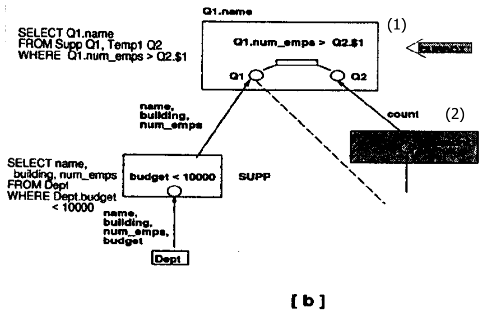


Figure 5.22: Pushing the Selection Condition

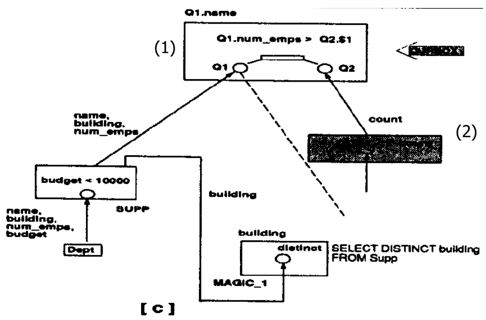


Figure 5.23: Removing Duplicates

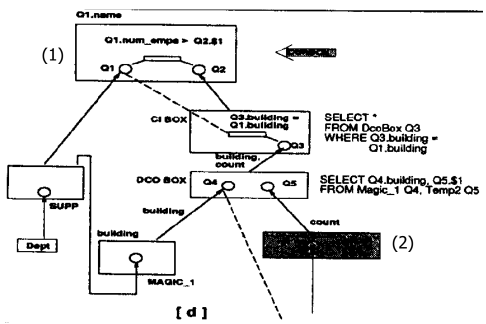


Figure 5.24: Removing the correlation between (1) and (3)

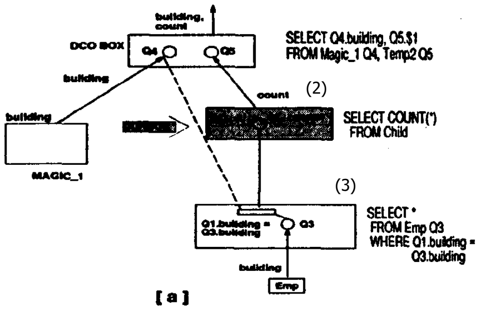


Figure 5.25: Starting point for box (2)

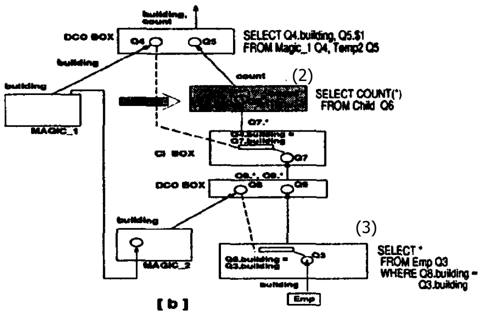


Figure 5.26: Feed for Box (2)

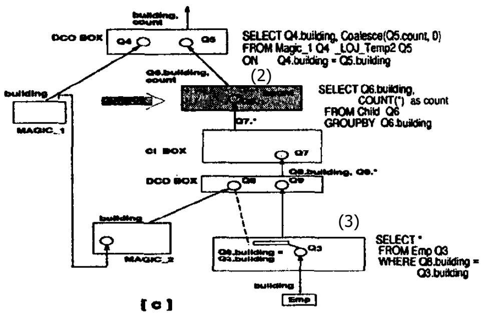


Figure 5.27: Absorb for box (2)

5.6.4 Application to CEP

Query decorrelation includes joining materialized results with an outer query block. In principle, such problem could also be applied to advanced CEP queries. In our

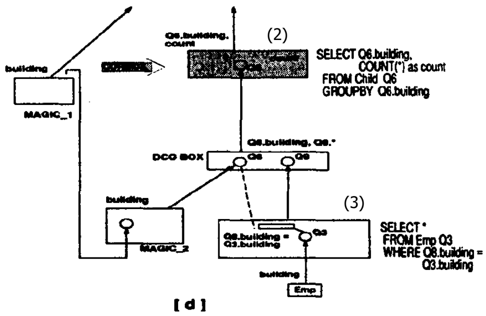


Figure 5.28: Remove unnecessary C1 box

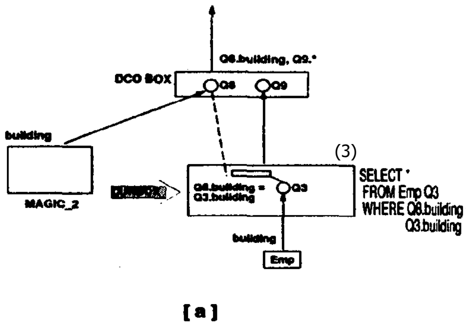


Figure 5.29: Starting point for box (3)

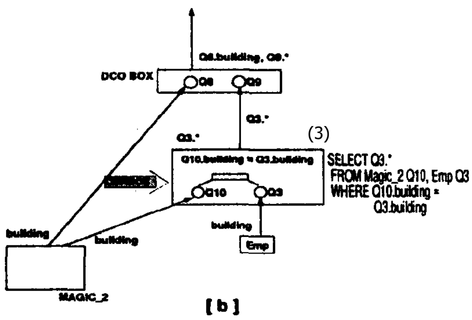


Figure 5.30: Absorb for box (3)

model, we do not consider "views/caches" and joins between separate views and a query. Hence in our work, we don't allow this path. Instead, we leave it for future work. In this section, we explore potential decorrelation techniques in the CEP

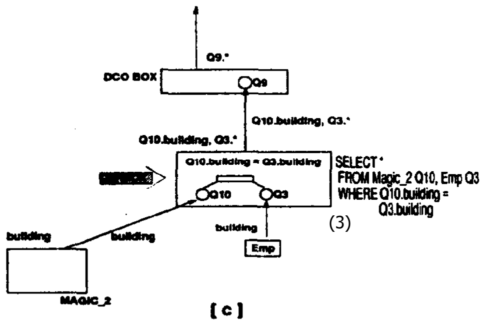


Figure 5.31: Remove unnecessary Q8 input to DCO box

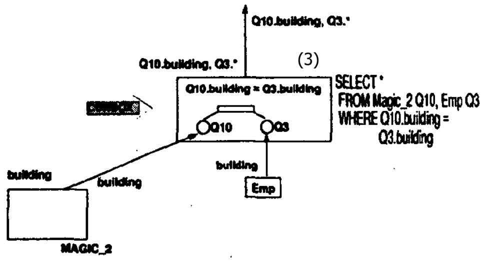


Figure 5.32: Remove unnecessary DCO box

context if we have to support joining between separate views.

Correlated CEP Query Examples. Q_7, Q_8 and Q_9 are sample correlated CEP queries. We will use them as running examples for the CEP query decorrelation.

```
Q7 = SEQ(R r, S s, T t, t.attr1 > 100, t.attr2 >
          (AGG(Count(*)
           SEQ(U u, V v, t.attr5 = u.attr5))
           ))
     WITHIN 1 hour

Q8 = SEQ(R r, S s, SEQ(U u, V v, u.attr3 = s.attr3),
     T t, t.attr1 > 100, s.attr2 < 50)
```

```
      WITHIN 1 hour

Q9 = SEQ(R r, S s, T t, t.attr1 > 100, s.attr2 < 50,
          EXIST (
                 SEQ(U u, V v, u.attr3 = s.attr3,
                      u.ts > s.ts and v.ts < t.ts)))
      WITHIN 1 hour

Q9 = SEQ(R r, S s, T t, t.attr1 > 100, t.attr2 >
          (AGG(Count(*)
           SEQ(U u, V v, t.attr5 = u.attr5))))
      WITHIN 1 hour

Q10 = SEQ(R r, S s, SEQ(U u, V v, u.attr3 = s.attr3),
           T t, t.attr1 > 100, s.attr2 < 50)
      WITHIN 1 hour

Q11 = SEQ(R r, S s, T t, t.attr1 > 100, s.attr2 < 50,
          EXIST (SEQ(U u, V v, u.attr3 = s.attr3,
                        u.ts > s.ts and v.ts < t.ts)
                  WITHIN 1 hour))
      WITHIN 1 hour

Q12 = SEQ(R r, S s, T t, t.attr1 > 100, s.attr2 < 50,
          NOT EXIST ( SEQ(U u, V v, u.attr3 = s.attr3,
                              u.ts > s.ts and v.ts < t.ts)
                        WITHIN 1 hour))
      WITHIN 1 hour
```

CEP Query with Aggregate subquery

NEEL Query Rewrite. The QGM construction method for *NEEL* is similar to the one for SQL. Namely, each event expression formed by a SEQ and an aggregate corresponds to a query block in the QGM. Window constraints are omitted in QGM.

Query Decorrelation Procedure. The magic decorrelation rewrite rule is applied to this CEP QGM in a top-down fashion, transforming one box at a time. CurBox corresponds to the box currently being processed.

For aggregate CEP query decorrelation, no CEP specific procedure needs to be designed. The reason is the correlated attributes between outer and inner query blocks are not pattern specific. We first describe the CEP query decorrelation procedure the same as the one described in Example 30 for SQL query decorrelation. And Feed and Absorb stages are explained further by Example 31.

Remove Correlation for CurBox.

- Traverse QGM in depth first order.
- For each current box A, check if a (descendant) box B is correlated to A/A's ancestor.
  - If yes, then *feed* the correlation to its child (if any). In the FEED stage, we determine if the child box is correlated. If so, it generates the set of correlation bindings that can be used to decorrelate the box.
  - If A is correlated to an (ancestor) box, then *Absorb* the correlation for box A Recall that Absorb will be different depending on whether the box is an aggregate box or an SPJ box. In the ABSORB stage, when

the rewrite rule is applied to the subquery, it decorrelates the subquery using the correlation values.

Feed Stage for CurBox.

- Check if there is any condition on the "correlation" attribute in CurBox. If yes, push the selection condition before CurBox (see 5.6.2).
- Create another box (corresponding to the "magic" expression), which removes duplicate values of the correlation attribute. A unique set of correlation bindings is projected into results for the "magic" expression for the child.
- The final step of the FEED stage is to decouple the CurBox from the child box. This is accomplished by creating 2 boxes DCO and CI:
  - DCO (Decorrelated Output) box: To decouple the CurBox from the child box, a DCO box is introduced immediately above the child, to produce a *decorrelated view* of the child to the parent. The DCO box has an iterator Q_m over the magic table of the child and an iterator Q_c over the child, and computes the cross product of the two.
  - CI (Correlated Input) box: A CurBox needs a correlated view of the subquery to retain the relationship between each correlation value and the corresponding answer from the decorrelated subquery. A Correlated Input (CI) box is introduced immediately above the DCO box, with a correlated predicate that provides this view to the CurBox. CI box takes output of DCO box. CI box is correlated to CurBox and performs the appropriate join method.

Absorb Stage for CurBox. It is usually possible to eliminate the Decorrelated Output (DCO) box entirely. This happens when rewrite rules are applied to the child box (which is now treated as the CurBox). There is a DCO box immediately above the CurBox with an iterator over its magic expression. During the ABSORB stage, the CurBox needs to absorb the correlation bindings that are available in the magic expression. In this Section, we only consider decorrelate aggregate CEP query. So if the CurBox is not SEQ box (e.g. it is an aggregate box), absorb includes adding the correlation attribute to the output, and a grouping by that attribute, followed by a left outer join (LOJ). Namely, for non-SEQ box, the actual correlation is usually contained in some descendant of the CurBox. Therefore, the correlation bindings in the magic expression should be fed to the children of the CurBox, so that they can be decorrelated. Once the children have been decorrelated, the CurBox can absorb the correlation bindings from the children.

Example 31 *Queries expressed by NEEL can be converted to SQL queries such as Q7SQL below. After converting NEEL with join predicates to SQL, we can apply existing query decorrelation technique to optimize the execution of NEEL expressions.*

```
Q7 = SEQ(R r, S s, T t, t.attr1 > 100, t.attr2 >
          (AGG(Count(*)
               SEQ(U u, V v, t.attr5 = u.attr5)
           )))
     WITHIN 1 hour

Q7SQL =
SELECT r, s, t
```

```
FROM R,S,T
WHERE t.te - r.ts < 1 hour and t.attr1 > 100 and t.attr2 >
        (SELECT count(*)
         FROM U, V
         WHERE U.ts < V.ts and t.attr5 = u.attr5)
```

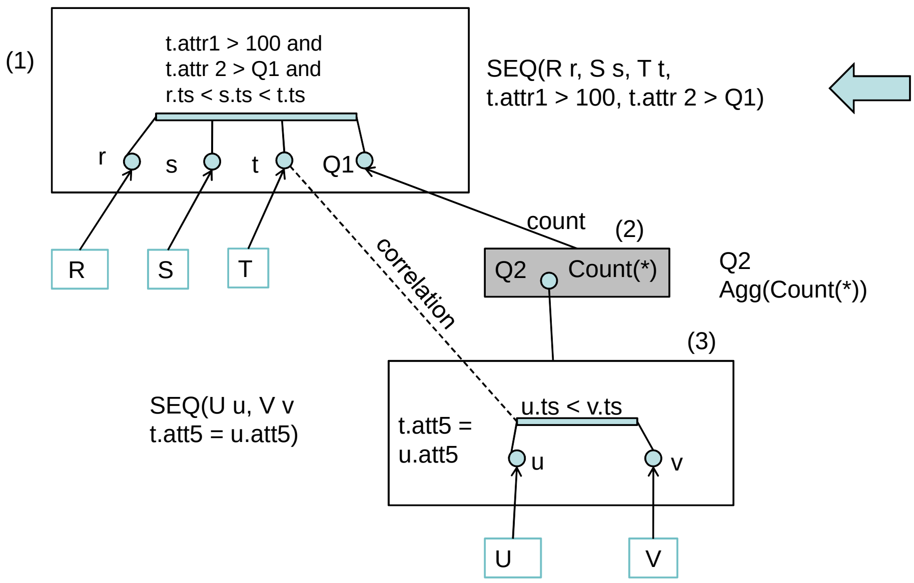


Figure 5.33: QGM for Q_7

QGM for Q_7 is shown in Figure 5.33. We first visit box (1). It has a descendant box, that is correlated to it. So we perform the feed. Box (1) is not correlated to an ancestor box, so there is no absorb. Let us see how feed for box (1) is performed. The predicate (t.attr1 > 100) is pushed before Box(1) (see Figure 5.34). We create another box magic$_1$ which removes duplicated t.attr2 (see Figure 5.35). DCO and CI boxes are created (see Figure 5.36). DCO box takes magic$_1$ and box 3 as input. Box (3) will now depend on this box. CI box takes output of

DCO box. CI box performs the equi-join. Next, let us decorrelate Box (2). The starting point for box (2) is shown in Figure 5.37. Box (3) is correlated to the parent DCO Box (2). So we perform the feed (see Figure 5.38). A DCO box and a CI box are created as before. Box (2) is correlated to its parent DCO box. We must perform the absorb (see Figure 5.39). We end up with an unnecessary CI box and we remove it (see Figure 5.40). Last, we decorrelate Box (3). The starting point for Box (3) is shown in Figure 5.41. There is no feed stage for Box (3) as no descendant box that is correlated to Box (3) or its ancestor exists. Thus, we can simply perform the absorb for Box (3) as it is correlated to its parent DCO box (see Figure 5.42). The iterator Q_7 *over the magic table in the DCO box is now redundant as the correlation bindings (Q10.att5) from the magic table iterator are added to the output of the CurBox and can be removed, leaving the CurBox decorrelated as in Figure (see Figure 5.43). Lastly the unnecessary DCO box is removed (see Figure 5.44). Figure 5.45 shows the final decorrelated query.*

Discussion. *Techniques to decorrelate SEQ queries could be applied to nested CEP queries with aggregate sub-queries. Decorrelation techniques for CEP queries are identical. Outer and inner CEP subexpressions are correlated involving event attributes. And we could treat a SEQ query as a special join. Decorrelation techniques help us improve performance. In our final decorrelated query, we only compute the aggregation result once for each distinct t.attr5.*

CEP Query with Non-aggregate subquery

Decorrelation techniques presented in [SPL96] mainly focus on CEP queries with aggregate subqueries. Let us re-consider the drawbacks that magic techniques

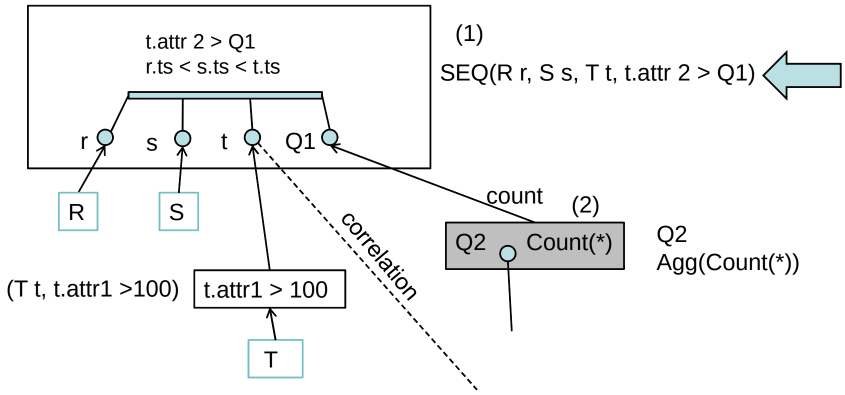


Figure 5.34: Push Predicates

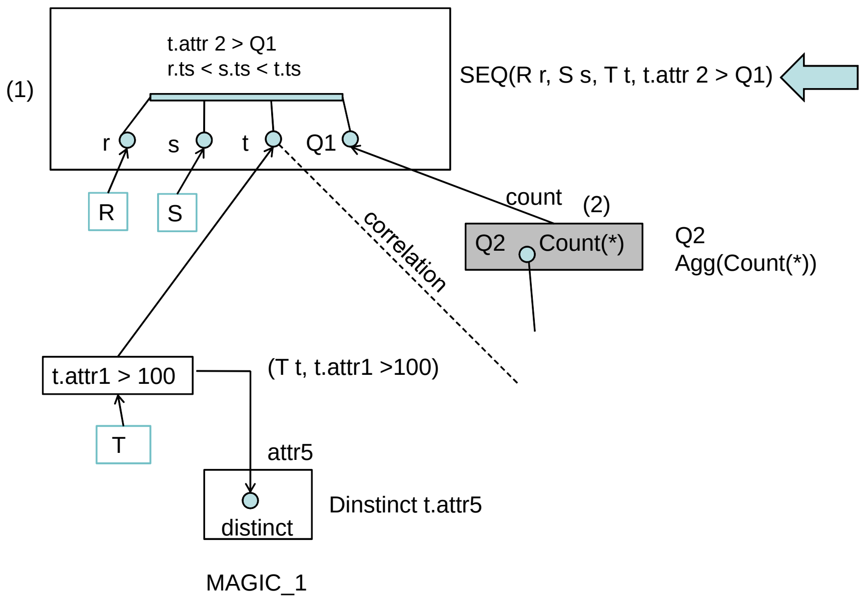


Figure 5.35: Create Magic Box

avoided. As mentioned earlier, the drawbacks of nested iteration are threefold: (1) duplicate invocations, (2) redundant work in each invocation, and (3) interference with processing in outer query block.

Non-aggregate CEP Query Optimization.

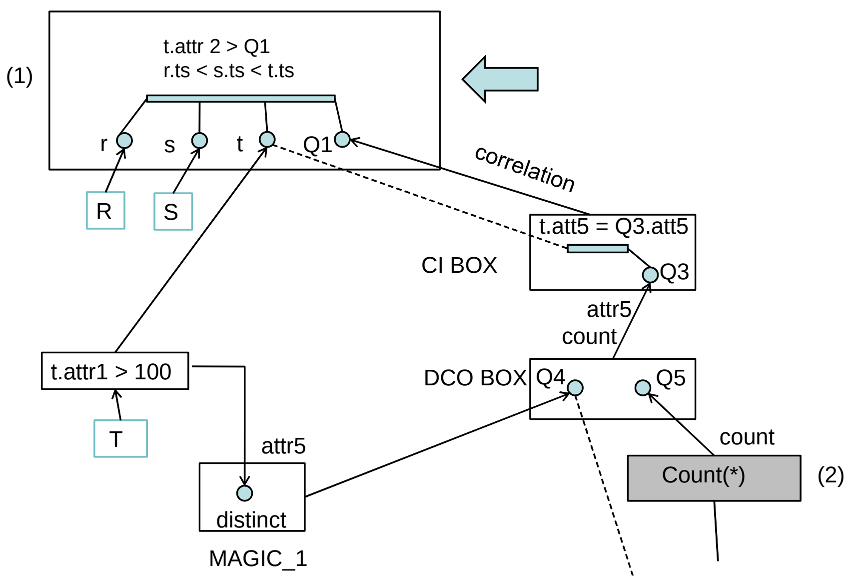


Figure 5.36: Creat DCO and CI Boxes

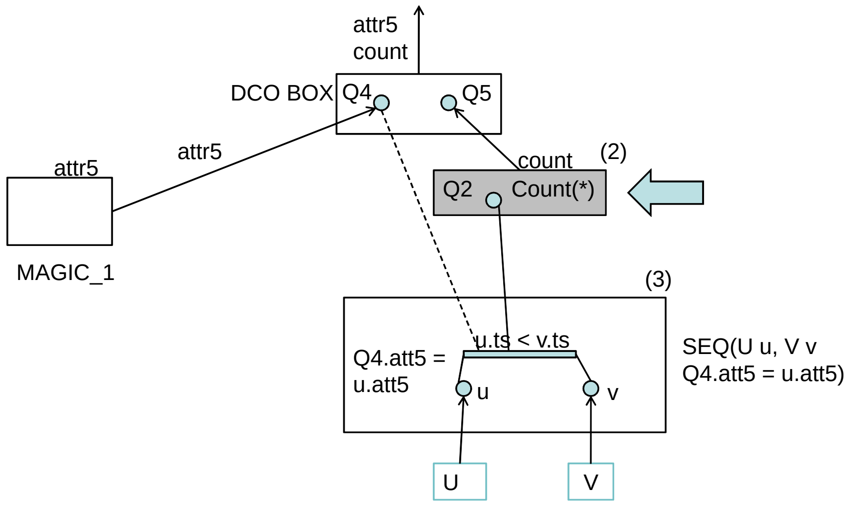


Figure 5.37: Starting Point for Box 2

First, we apply magic techniques for such queries. The steps are similar to Section 5.6.4. The differences are: (1) To capture the temporal subsequence context,

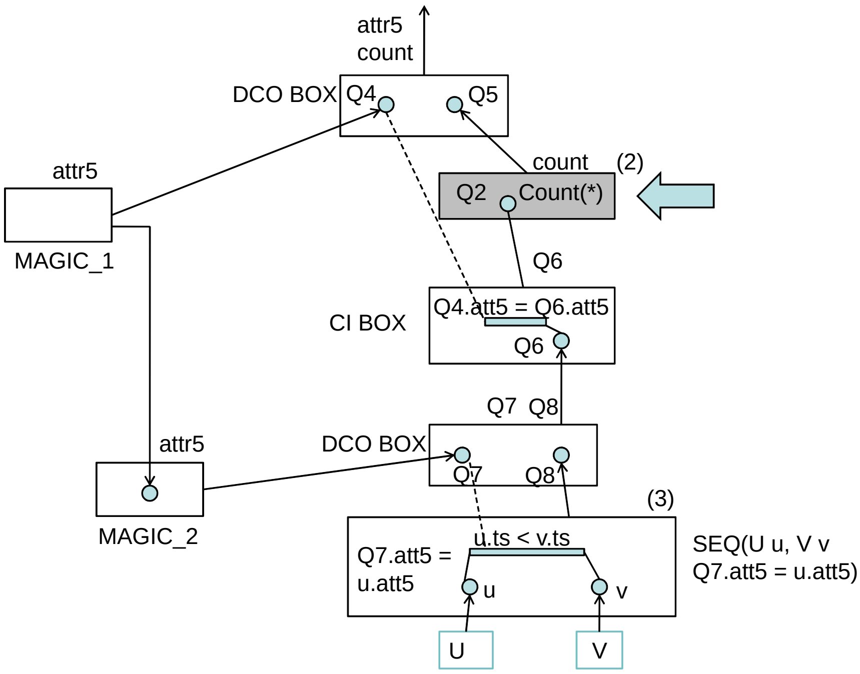


Figure 5.38: Feed Stage

the magic box contains distinct temporal pairs instead of distinct single values. (2) To minimize redundant work, we need to materialize results for each distinct temporal pair. (3) To eliminate duplicate invocations, we only compute results for correlated subqueries when answers were not materialized.

The *IntervalConstraints (distinct temporal pairs)* is computed for each sub-query given an outer query result triggered by an event *e*. It is given by the time-stamps of the events which bound the sub-queries. For each parent expression match, results of its subexpression are computed. The same triggering event *e* may generate multiple results for each subexpression with overlapping intervals. For example, assume one temporal pair $pair_1$ = [1, 5] and the other temporal pair $pair_2$ = [1, 10]. Cache results for $pair_2$ contain cache results for $pair_1$. We apply

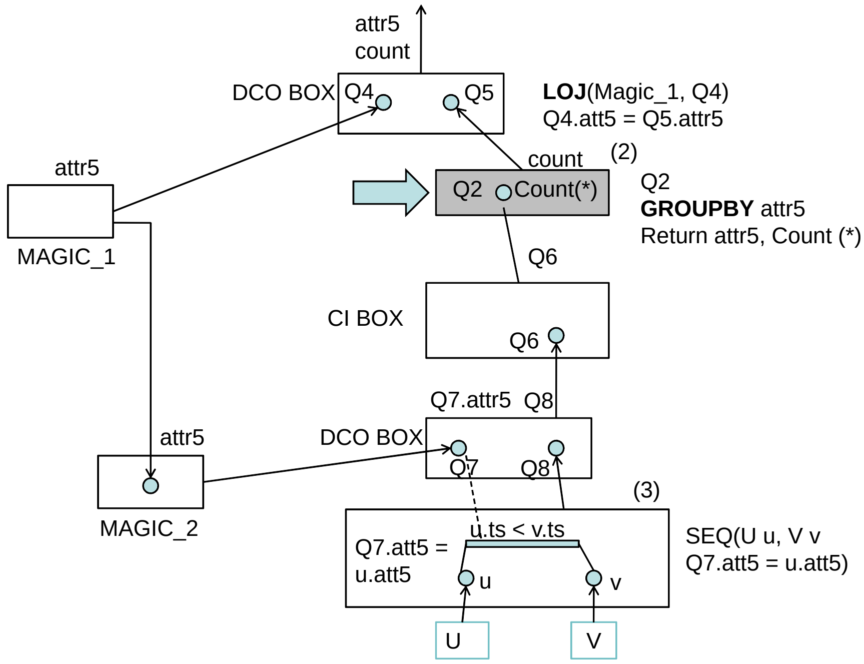


Figure 5.39: Absorb Stage

the set-based option to pre-compute table of magic decorrelation. To avoid re-computation of results occurring in the same interval, distinct temporal pairs are maintained in MAGIC box. Such meta-data "interval" is attached to the respective cache to indicate the time period for which its results are cached for. All possible results for each subexpression occurring within each interval (temporal pair) are stored in the respective cache.

As CEP queries work on sliding windows, it is easy to see that many intermediate results would continue to be valid from one sliding window to the next. Previously calculated results of the previous window should be cached and then be reused in the new window. We propose to cache and incrementally maintain the inner query results. So we could modify the cache maintenance method above for

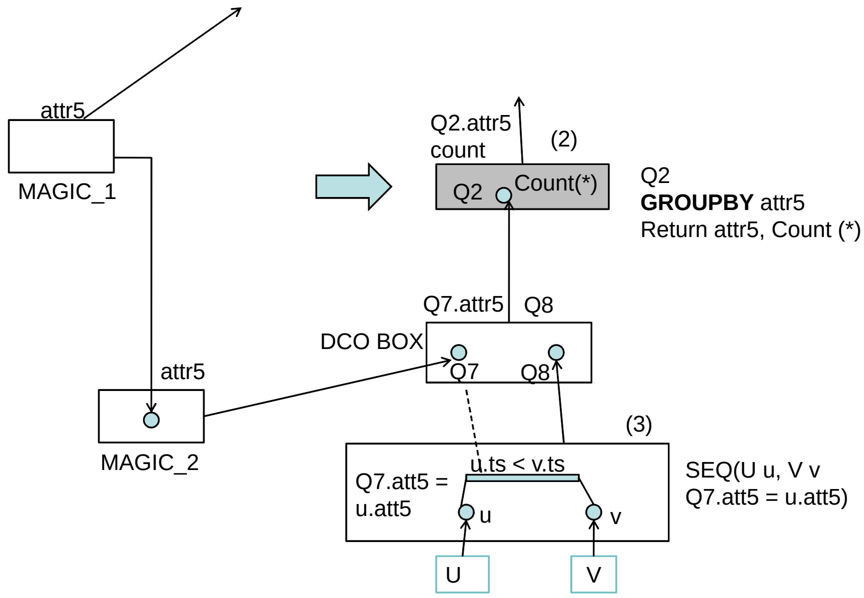


Figure 5.40: Remove Unnecessary CI Box

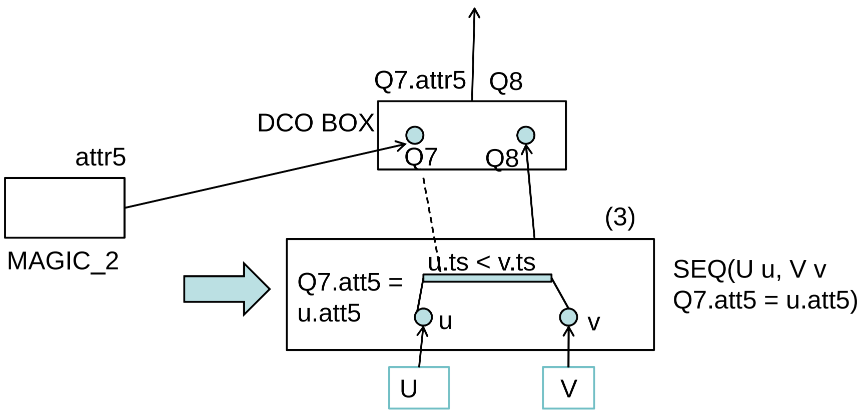


Figure 5.41: Starting Point for Box 3

more result reuse. In addition, we could consider batch processing.

Final Outer Result Generation. The generation of the final outer results depends on the type of the inner subqueries. Namely, if the subquery is a positive component in an event expression (e.g., Q_8), for each outer sequence result, it will join

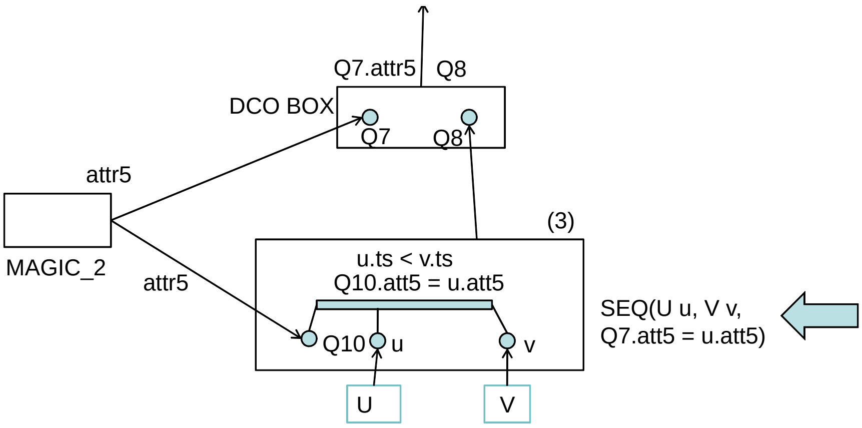


Figure 5.42: Absorb Stage

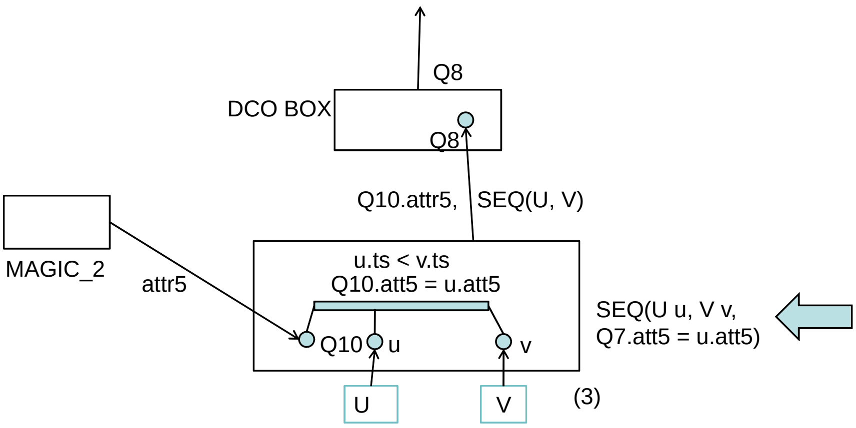


Figure 5.43: Remove Unnecessary Input

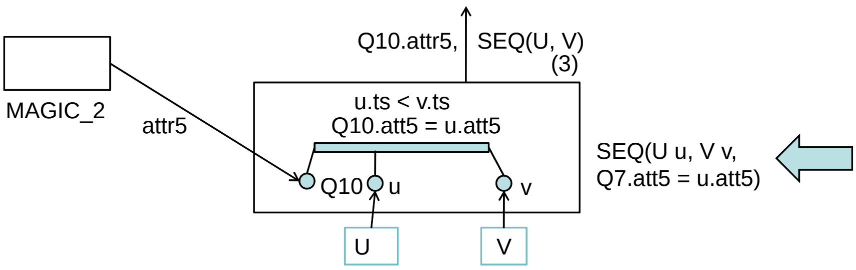


Figure 5.44: Remove Unnecessary DCO Box

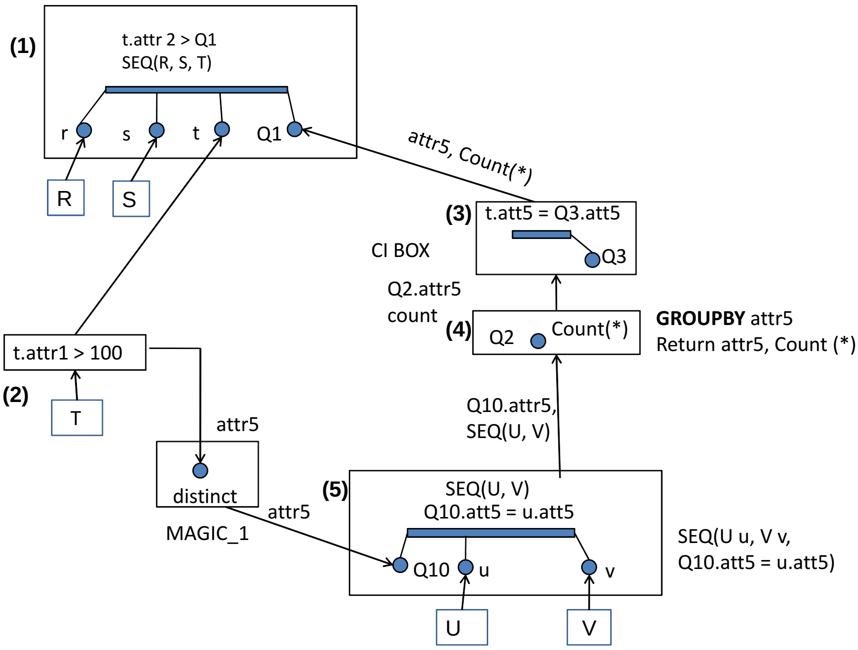


Figure 5.45: Final Decorrelated Graph

with cache results (if exist) using the most appropriate method (e.g., merge join). When the subquery is connected by <Eop> <Query> to the outer expression, then if Eop is "EXIST", for each outer sequence result, it is to be returned if the inner subquery result set is not empty (e.g., Q_9).

Example 32 *Queries expressed by NEEL can be converted to SQL queries such as Q8SQL below. After converting NEEL with join predicates to SQL, we can apply existing query decorrelation technique to optimize the execution of NEEL expressions.*

```
Q8 = SEQ(R r, S s, SEQ(U u, V v, u.attr3 = s.attr3), T t,
          t.attr1 > 100, s.attr2 < 50)
     WITHIN 1 hour
```

```
Q8SQL =
SELECT r, s, Qinner, t
FROM R, S, Qinner, T
WHERE R.ts < S.ts < T.ts and t.attr1 > 100 and s.attr2 < 50 and t.ts-r.ts < 1 hour
and Qinner IS IN SELECT u, v
                 FROM U, V
                 WHERE u.attr3 = s.attr3 and v.ts < t.ts and s.ts < u.ts
```

We create a $magic_1$ box which removes duplicates [s.ts, t.ts] pairs. DCO and CI boxes are created (see Figure 5.46). DCO box takes $magic_1$ and box (2) as input. CI box takes output of DCO box and it is correlated to box (1). Next, we decorrelate Box (2). There is no feed stage for Box (2) as no descendant box that is correlated to box (2) or its ancestor exists. We perform the absorb for Box (2) as it is correlated to its parent DCO box (see Figure 5.47). Box(2) adds the magic table as its input iterator. The source for correlation predicates is now the magic table iterator in Box(2). Unnecessary input from MAGIC-1 to DCO box is removed (see Figure 5.48) and unnecessary DCO box is removed (see Figure 5.49). The final graph after applying magic technique is shown in Figure 5.50. For each distinct [s.ts, t.ts] time pair, inner SEQ(U u, V v) results are materialized if predicates u.attr3 = s.attr3, u.ts $>$ s.ts and v.ts $<$ t.ts are satisfied. In streaming context, for every new constructed SEQ(R r, S s, T t) result, we check for [s.ts, t.ts] if the corresponding inner SEQ(U u, V v) results are computed before. If yes, we use materialized results. Otherwise, we compute it from scratch.

Example 33 `Q9 = SEQ(R r, S s, T t, t.attr1 > 100, s.attr2 < 50,`

```
        EXIST (
```

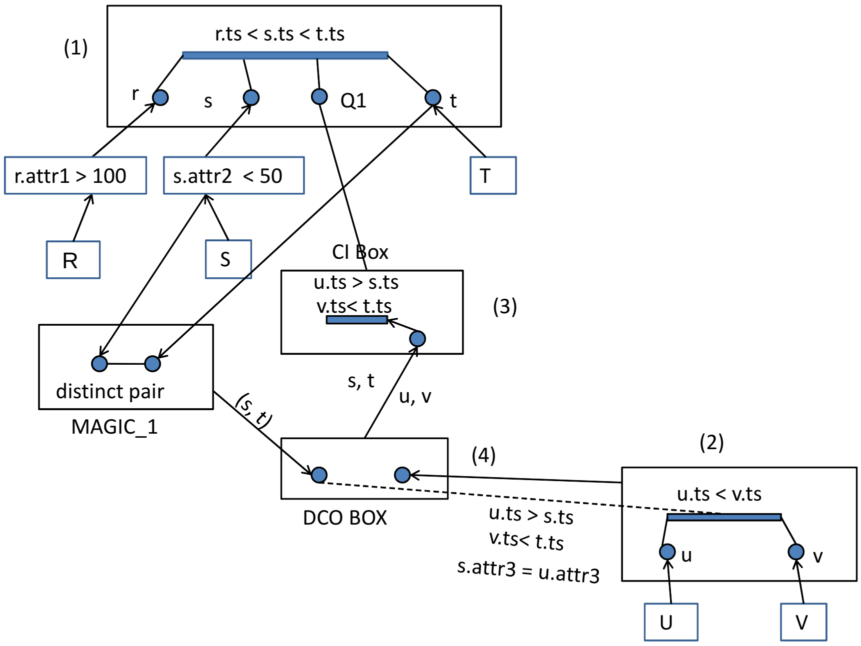


Figure 5.46: Push Predicates, Create Magic, DCO, CI Boxes

```
                SEQ(U u, V v, u.attr3 = s.attr3, u.ts > s.ts
                    and v.ts < t.ts)
                WITHIN 1 hour))
     WITHIN 1 hour

Q9SQL =
SELECT r, s, t
FROM R, S, T
WHERE R.ts < S.ts < T.ts and t.attr1 > 100 and s.attr2 < 50 and and t.ts-r.ts < 1 hour
      EXIST (SELECT u, v
             FROM U, V
             WHERE u.attr3 = s.attr3 and U.ts < V.ts and U.ts > S.ts
```

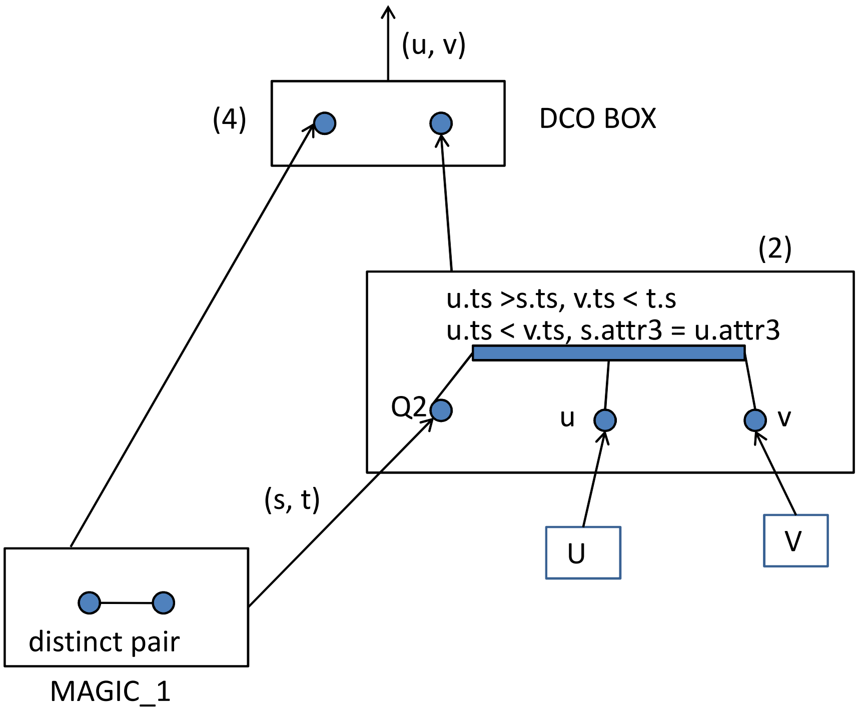


Figure 5.47: Absorb Stage

```
and V.ts < T.ts)
```

Consider the above correlated query Q_9 which can also be expressed by Q9SQL. The optimization steps are similar to Example 32. The final result construction is different. As the subquery SEQ(U u, V v, u.attr3 = s.attr3, u.ts > s.ts and v.ts < t.ts) WITHIN 1 hour is connected by "EXIST", for each outer sequence result <r, s, t>, it could be returned if the inner subquery result set is not empty.

Novel Issues of Decorrelation Technique in CEP Context. A few novel issues are explored as listed below.

- The magic table in magic decorrelation deals with distinct attribute values.

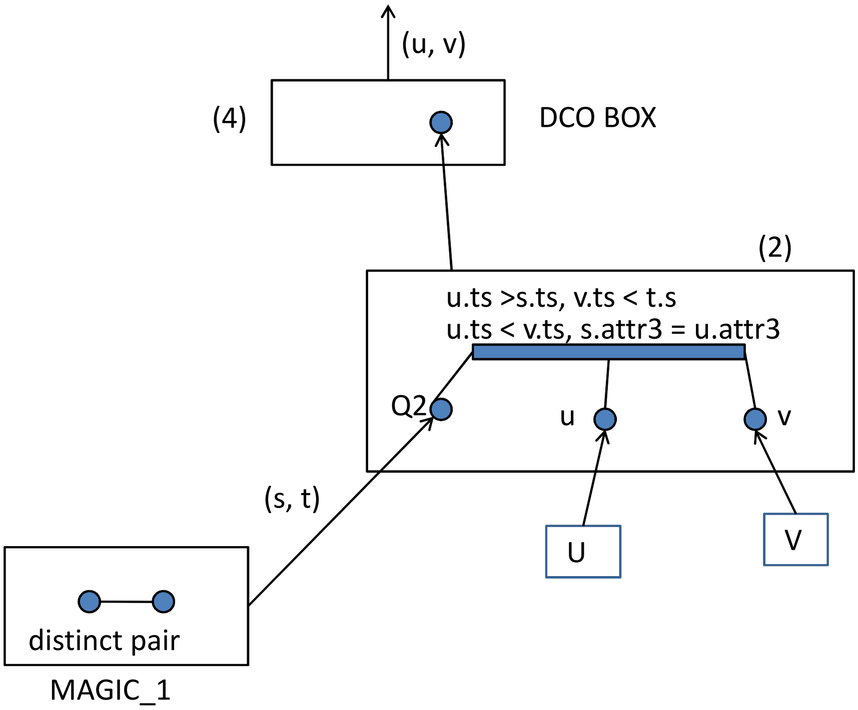


Figure 5.48: Remove Unnecessary DCO Input

However, in the nested CEP context, we need to extend it with distinct temporal pairs to capture stringent windows.

- The current decorrelation techniques only support static data. We consider streaming data for nested CEP queries.
- The Query Graph Model (QGM) is designed for SPJ queries. We have extended QGM for nested CEP queries with time correlation.
- We design optimization techniques for correlated CEP subqueries.

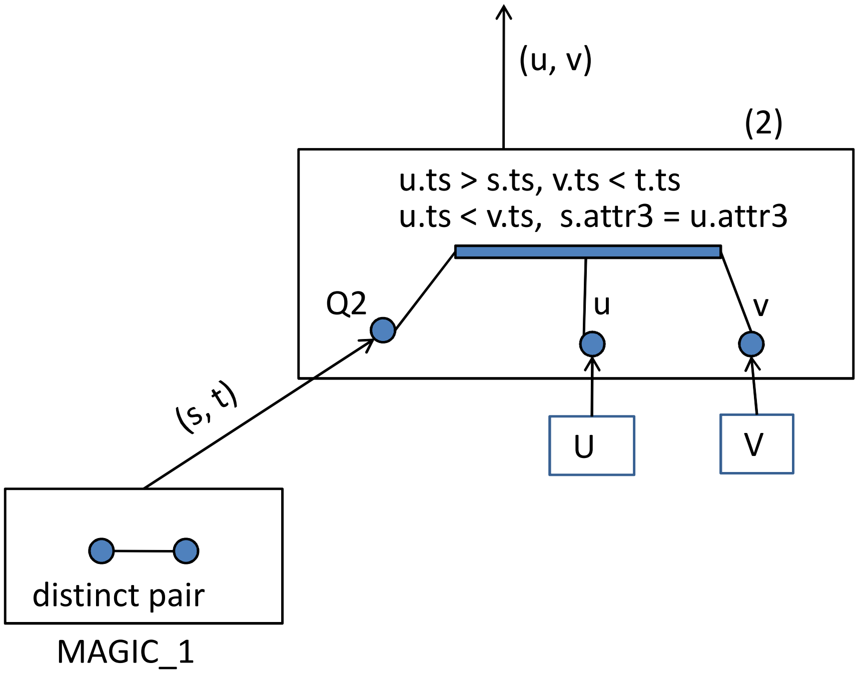


Figure 5.49: Remove Unnecessary DCO Box

5.7 Related Work

To the best of our knowledge, existing CEP systems [WDR06, BDG$^+$07, MM09, BGAH07, LLG$^+$09] mostly support the execution of only flat sequence queries. While CEDR [BGAH07] allows applying negation over composite event types within their proposed language, the execution strategy for such nested queries is not discussed. In addition, no work has been reported on tackling the performance deficiency when applying negation over composite event types.

SASE [WDR06, GADI08] supports novel language features such as negation, and demonstrates performance gain in processing complex event queries compared to traditional data stream processing system TelegraphCQ. We borrow from SASE

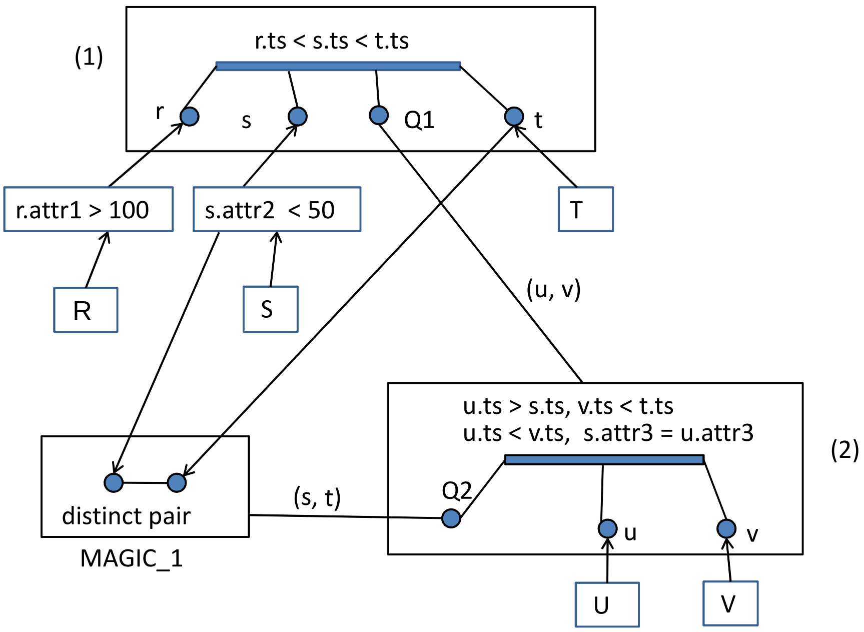


Figure 5.50: Final Decorrelated Graph

query syntax and algebra operators. However, the event (query) language of SASE is not composable, which restricts the set of queries expressible in the system. SASE [WDR06, GADI08] considers only flat queries and negation is applied as a final filtration step. Cayuga [BDG$^+$07] is able to inline one automaton into another automaton that reads the output of the former. For example, a Cayuga query (S1;S2);S3 can be naively implemented by two automata as follows. The first automaton A implements S1;S2, and produces an intermediate stream S'. The second automaton B implements S';S3. In this case, Cayuga can inline A into B, by replacing the forward edge of the start state of B with A, eliminating the need for producing the intermediate stream S'. This is supported in Cayuga as query plans are composable. However, Cayuga doesn't discuss applying negation over composite event types. ZStream [MM09] considers the ordering of execution for CEP queries using a tree-based query plan – similar to join ordering in traditional relational databases. It only supports negation over primitive event types. ZStream doesn't consider optimization over multiple expressions nor of nested CEP expressions. In short, no processing mechanisms nor optimization methods for CEP queries with nested complex negation have been proposed in the literature to date.

Complex pattern queries often contain common or similar sub-expressions within a single query or also among multiple distinct queries. Multiple-query optimization in databases [Sel88, RSSB00, Fin82] typically focus on static relational databases and identifies common subexpressions among queries such as common joins or filters. However, multiple expression sharing for stack-based pattern evaluation for CEP queries has not yet been studied. In particular, our work is the first to share the processing of CEP expressions with the same positive event types interleaved with different negative event types.

STREAM's CQL query language [ABW06] extends SQL with support for window queries. Like SQL itself, CQL is declarative. However, it is not clear whether CQL is suitable for realtime event detection and composition. Similar to SQL, the data model underlying these stream query languages is unordered, and so in order to pin-point the i-th tuple within a set of N tuples returned by a window operator, an N-way self-join with temporal constraints on these N tuples is required. In [LWZ04], it is shown that SQL lacks expressive power for continuous queries on data streams, and the authors in [WZL03] extend SQL with features to support data mining and data streams. CQL offers only little explicit support for queries that involve temporal relationships between events (or tuples). They don't support events occurring over time-intervals explicitly. In CQL, time is primarily treated in two ways: (I) it's an attribute and as such can be involved in any predicates such as x1.ts $<$ x2.ts, and (II) for time-based window.

Work on temporal and sequence database systems has emphasized static datasets instead of data streams [RDR$^+$98, SZZA01, SLR95]. As pointed by [ME04], there are several proposals to extend the database query languages with means to search for sequential patterns. The specifics of the event data such as the event instance selection and consumption policies are not considered.

Chapter 6

Discussion of Solution Integration

Recent years have witnessed a rapid increase in attention in CEP systems [WDR06, MM09, DGP^{+}07, GADI08, Jag08] that extract flat patterns from event streams and make informed decisions in real-time. Efficient, scalable and robust methods for in-memory multi-dimensional nested pattern analysis over high-speed event streams need to be designed for CEP engines. These research challenges tackled in my dissertation are categorized into the following: (I) Lack of Nested Pattern Query Language; (II) Lack of processing strategies and optimization methods for nested pattern queries; (III) Lack of event model for pattern queries over different abstraction levels; (IV) Lack of processing strategies and optimization methods for Pattern Queries over Different Abstraction Levels; (V) Lack of mechanisms for out-of-order event handling.

This dissertation focuses on extending event sequence processing with new models and optimization techniques by meeting the above research challenges. As mentioned earlier in Chapters 3), 4 and 5 respectively, the techniques proposed to tackle these research challenges have each been addressed in isolation. For example, the out-of-order event handling framework introduced in Chapter 3 includes K-slack, conservative and aggressive methods with limited query support (flat SEQ queries). In the proposed ECube framework (Chapter 4), assumptions of in-order events and flat SEQ queries are made. In the proposed nestedCEP framework (Chapter 5), we assume events arrive in order. Clearly, in a practical system, our proposed techniques need to work together within an integrated system to solve more complex scenarios. In the following we study the extensions for the proposed techniques which make an integrated system possible.

E-Cube with Out-of-Order Event Streams.

Again by the same arguments as above, the K-slack method would work correctly with the proposed ECube framework (Chapter 4). E-Cube concept hierarchy and event pattern query hierarchy are orthogonal to supporting out-of-order events as they are defined independently of event arrivals. To apply the conservative method to E-Cube, we need to extend metadata (Partial Order Guarantee (POG)) to support event types in an event concept hierarchy. For example, we could have a POG notification specifying no more event instances with event type USA will come. Similarly, we could only have POG specified for a particular state in USA. Since correct results are guaranteed to be generated even when events arrive out of order, we can still apply the existing conditional computation mechanism. To apply the aggressive method to E-Cube, a revision tuple propagation strategy should be taken care of between queries with conditional computation. For example, consider two queries q_i = SEQ(A, B, C) and q_j = SEQ(A, B, C, D, E) with pattern changes in E-Cube. Assume a c_k event of type C arrives out-of-order. Revision tuples such as $< a_i, b_j, c_k >$ are constructed for q_i for the general to specific method. Such revision tuple needs to further join with D and E events in q_j. Similarly, when a a_i event of type A or a b_j event of type B arrives out of order, revision tuples are constructed for q_i involving a_i or b_j and are propagated to q_j. When a d_j event of type D or a e_k event of type E arrives out of order, revision tuples are constructed for q_j by joining SEQ(D, E) results involving d_j or e_k with stored SEQ(A, B, C) results.

Nested CEP Query Processing for Out-of-Order Event Streams.

NEEL syntax, semantics of operators we defined, rewriting rules, optimization methods are orthogonal. They are all independent of out-of-order handling methods because the correctness of them is not impacted by out-of-order handling. They are defined independently of event arrivals. The only issue is related to execution itself. The nested CEP query processing framework introduced in Chapter 5 includes the iterative nested execution strategy and the shared optimized NEEL pattern execution. The K-slack method in literature works correctly with the nested complex CEP query processing framework without any changes. The reason is out-of-order events are sorted in the buffer and CEP systems process in order events as usual. To apply the conservative and aggressive methods, we first need to extend our out-of-order processing to also support AND and OR operators. The mechanism would be rather similar to SEQ. Essentially, the nested execution strategy computes flat subexpressions at each level. The conservative methods developed for flat CEP expressions can be directly applied to the subexpression at each nesting level. For the aggressive method, we need to take care of the revision result propagation between levels. For shared NEEL pattern execution, as queries are flattened, existing techniques to compute results for common subexpressions could be applied. For example, two expressions SEQ(A, B, C) and SEQ(A, B, C, D) share the common prefix SEQ(A, B, C). Assume we apply the aggressive method and the event b_{12} of type B arrives out of order. SEQ(A, B, C) results involving b_{12} such as $\{a_i, b_{12}, c_k\}$ are computed first using existing techniques. These results will be joined with D events in window to form revision tuples. As another example, two expressions SEQ(A, !B, C, D) and SEQ(A, C, !E, D) share the common generating expression SEQ(A, C, D). When an event c_{10} of type C arrives out of order, SEQ(A, C, D) results involving c_{10} such as $\{a_i, c_{10}, d_k\}$ are computed first using the existing techniques. The bit-marking method is the same. Namely, for each $\{a_i, c_{10}, d_k\}$ result, we check the existence of B (E) events between a_i and c_{10} (c_{10} and d_k).

E-Cube for Nested CEP Queries.

Similar to SEQ, we need to extend the current ECube model with additional query refinement and reuse support for queries containing AND, OR and boolean expressions. To process nested CEP queries over multiple abstraction levels, we first rewrite these nested CEP queries into a normal form [LRG+11a]. Then we could apply E-Cube techniques to normalized sub-expressions. For example, assume B is at a coarser level than b in a concept hierarchy and after rewriting, we get q_i = SEQ(A, B, D) OR SEQ(A, b, D) OR SEQ(A, b, D, ∃ E). SEQ(A, B, D) is at a coarser level as compared to SEQ(A, b, D) with concept changes. SEQ(A, b, D) should be coarser than SEQ(A, b, D, ∃ E) with pattern changes. We thus could apply reuse and optimization methods in E-Cube for these subexpressions. Reuse among queries would be for particular components of these queries.

Chapter 7

Conclusions

7.1 Conclusions

Objectives of the dissertation focus on extending event sequence processing with new models and optimization techniques by meeting the four research challenges motivated in Chapter 6. This dissertation innovates several techniques to achieve efficient, scalable and robust methods for in memory multi-dimensional nested pattern analysis over high-speed event streams. The dissertation research is as described below.

In part I, we address the problem of processing pattern queries on event streams with out-of-order data arrival in our *E-Analytic* system. We analyze the problems state-of-the-art event processing technology experiences when faced with out-of-order data arrival including blocking, resource overflow, and incorrect result generation. We propose two complimentary solutions that cover alternative ends of the spectrum from norm to exception for out of orderness. Our experimental study demonstrates the relative scope of effectiveness of our proposed approaches, and also compares them against state-of-art *K-slack* based methods. Most current event processing systems either assume in order data arrivals or employ a simple yet inflexible mechanism (*K-slack*) which as our experiments confirm will induce high latency. Our work is complementary to existing event systems. Thus they can employ our proposed conservative or aggressive solutions according to their targeted application preferences.

In part II, our proposed E-Cube combines OLAP and CEP functionalities. We apply E-Cube techniques in our *E-Analytic* system to allow users to efficiently query large amounts of event stream data in multiple dimensions and at multiple abstraction levels. To the best of our knowledge, no prior work combines CEP and OLAP techniques for multi-dimensional pattern analysis over event streams as described in this Chapter. Our E-Cube solution improves computational efficiency for multi-dimensional event pattern detection by sharing results among queries in a unified query plan. Based on this foundation, we design a cost-driven adaptive optimizer called Chase which delivers optimal results. In the Chase method, our E-Cube optimization problem is mapped into a well-known graph problem. Our Chase method in many cases performs ten fold faster than the state-of-art strategy. Interesting future work includes supporting additional query features like recursion and closure as well as deployment on the cloud. Combining OLAP and CEP technologies requires both theoretical and practical contributions. On the theoretical front, we develop the solid foundation of a combined concept and pattern hierarchy. On the practical front, we present a methodology to efficiently process queries on streaming data over this hierarchy.

In part III, we describe the first work on comprehensively supporting nested query specification and execution in the CEP context. The CEP query language

NEEL in our *E-Analytic* system allows users to specify fairly complex queries in a compact manner with both temporal relationships and negation well-supported. A query plan for the execution of nested CEP queries is designed. This nested query plan model permits a direct implementation of nested CEP queries following the principle of nested query execution for SQL queries. However, such direct query execution suffers from several performance deficiencies. We thus design a normalization procedure converting a nested event expression into a normal form. We propose prefix caching, suffix clustering and a customized "bit-marking" physical execution strategy that efficiently process a group of similar subexpressions. An optimizer that employs iterative improvement capturing the optimal shared execution method is also designed. As demonstrated by our experiments, in many cases our optimized *NEEL* execution performs 100 fold faster than the traditional iterative nested execution. Our goal is to design nested CEP processing and optimization strategies that overcome the above identified shortcomings – thus significantly saving CPU processing resources.

7.2 Future Work

7.2.1 Generalizing ECube to Support Windows, Predicates and Aggregates.

Queries can have different window sizes, predicates and aggregates. These are interesting, related, but orthogonal topics that have been addressed by previous research using sliced time windows and shared data fragments [WRGB06, KWF06, LMT^{+}05]. In this chapter, we focus on the combination of pattern and concept hierarchies, while below we briefly sketch the application and extension of these existing ideas on sharing windows, predicates and aggregates across our E-Cube model.

7.2.2 Different Window Constraints

Assume window slides one tuple at a time and we partition stacks based on different window sizes. Each stack is partitioned into a continuous sequence of hierarchical slices. Assume two pattern queries q_i = SEQ(E_i, E_j) with window size w_i and q_j = SEQ(E_i, E_j, E_k) with window size w_j. The corresponding stacks for the event types E_i and E_j that are shared across the queries are partitioned into two slices, from 0 to w_i, and from w_i to w_j, assuming $w_i \leq w_j$. Events in the first w_i partition are logically also contained in the w_j partition. The hierarchy of slices is implemented by simple reference pointers w_i and w_j to the appropriate positions in the E_i data structure, i.e., the E_i stack. These window reference pointers are incrementally adjusted when new events of type E_i arrive as part of the regular insertion and purging process.

To reuse q_i results for q_j in the general-to-specific evaluation, q_i results are passed down to q_j in an intermediate buffer. The state is sorted by the minimum timestamp $e.ts$ among all components of each result tuple e. By sorting on such minimum timestamp for intermediate result tuples, we can efficiently purge results and determine result window ranges. For other reuse-based pattern evaluation strategies in E-Cube, similar variations of this state-slice idea can be applied.

Predicate Evaluation. Clearly as in traditional SQL OLAP cubes, if the join and select predicates for all queries are the same in E-Cube, then predicates over single positive event types can be pushed down to the WinSeq operator, filtering irrelevant events and preventing them from being placed into the corresponding stacks.

However, if the predicates are not the same, for events within the same window state-slice, we observe that queries with the same event pattern construct the same sequence results yet are filtered by different predicates. We apply customized "bit-marking" method for predicate evaluation [MSHR02]. The main idea of our strategy is to record the predicates applicable for each query at compile time. Information about queries that accept or reject a sequence result is encoded in the sequence result itself. We allocate a bitmap, queriesCompleted, with one bit per query, and store it in the sequence result. If a query's bit is set, it indicates that this sequence result has already been output or rejected by the query. Then the sequence result does not need to be output to that query. A completionMask list contains a bit mask for each query. Each completionMask indicates which operators in the operators list need to process a sequence result before it can be output. At run time, as we construct each sequence result, we keep track of which of the given predicate filters are satisfied by a sequence result via a bit marking. Then the correct tuple results are sent to the corresponding queries or stored in the corresponding intermediate states for future reuse.

Aggregation Processing.

If the aggregation function is incrementally computable such as count, we avoid retaining and re-processing tuples by maintaining partial aggregates [LMT+05]. The aggregate operator needs to store partial aggregates for not expired bins. At the beginning, a special "init" bin is labeled with -∞. Each result sequence sets up new start and end bins. Then the appropriate bins are updated. If the aggregation function is not incrementally computable, we need to materialize the actual sequence results so to be able to process the aggregation results.

Following our E-Cube model, queries with the same event pattern even if differing in window sizes, predicates or aggregates are grouped together. For the Chase evaluation in Section 4.4, the weight of each edge in MST would now correspond to the group computation costs of all pattern queries modeled by the same E-cuboid based on the results of another group. Given our reuse-based pattern evaluation strategies sketched above, the ordering among query groups decided by Chase would continue to be optimal. It is the straightforward extension of our E-Cube model. While the above indicates the compatibility of handling alternate windows, predicates and aggregation as part of E-Cube, we leave the discussion of more sophisticated techniques for integration into E-Cube such as pipelining and partial aggregation push-in as future work.

7.2.3 E-Cube resource limitations

The core E-Cube work assumes we have enough memory and the computing resources typically become strained before the memory does. So for a query, we would select conditional computation over self computation if the requirements for the optimal execution ordering are satisfied. In conditional computation, we need extra memory to store results which may be reused for other queries. If the cache storage space is limited, we can completely eliminate the use of cache or can use cache replacement policies to keep an upperbound on the number of cached patterns, maximizing the utilization of the cache. In addition, we could explore the idea of pipelining results. For example, for q_i = SEQ(A, B, C) and q_j = SEQ(B, C), In the top-down evaluation, we don't need to store q_j results. Instead, q_j results (b_i, c_j) can be pipelined to q_i as all A events with timestamps less than b_i.ts are store in the system.

7.2.4 Supporting Join Predicates in NEEL Expression Rewriting.

Currently, only simple predicates are supported for NEEL expression rewriting. We need to extend the rewriting system to support join predicates in NEEL expressions. For join predicates on negation, there is ambiguity which subexpression join predicates belong to. Suppose there is an attribute f that takes integer values. query1 = SEQ((A x), !(B y), (C z), (x.f $\leq$ y.f) $\wedge$ (y.f $\leq$ z.f)). Consider the history H = $\{a_1, b_2, c_3, c_0\}$ with a_1.type = A, b_2.type = B, c_3.type = C, c_0.type = C, a_1.f = 1, b_2.f = 2, c_3.f = 3 and c_0.f = 0. query1 on this H returns $\{a_1, c_0\}$. But now let query2 = SEQ((A x), !(B y), (C z), (x.f $\leq$ y.f) $\wedge$ (y.f $\leq$ z.f) $\wedge$ (x.f $\leq$ z.f)). A consequence of the condition in query1 is added in query2. But query2 cannot return $\{a_1, c_0\}$. The problem is that in query1 we see that y is defined inside a "!" and so we understand the (x.f $\leq$ y.f) and (y.f $\leq$ z.f) formula to be in the context "not there exists y such that (x.f $\leq$ y.f) and (y.f $\leq$ z.f)". But in query2, the (x.f $\leq$ z.f) part doesn't mention y at all and so is interpreted naively. The syntax lets us split off the conditions, such as (x.f $\leq$ y.f) $\wedge$ (y.f $\leq$ z.f) from the place where the variables are declared !(B y).

7.2.5 Integration of Complex NEEL Queries within an Extended E-Cube Analytics Framework.

Currently, the E-Cube system only supports flattened SEQ queries. To extend E-Cube system to support nested queries composed of SEQ, AND, OR and Negation, we could flatten a nested query to a normalized flattened query using the techniques proposed in Chapter 5. We need to extend event pattern query hierarchy to support queries with AND and OR operators. Computation sharing is achieved between subexpressions in the normalized query.

7.2.6 Parallel and Distributive Processing for Normalized NEEL Subexpressions

To make a CEP system scale in handling complex queries, pattern queries across a set of machines or use the existing resources more efficiently. Through NEEL query rewriting, a complex query is rewritten into a normalized expression. Each subexpression of such a normal form could then be executed in a parallel and distributive manner.

7.2.7 Marrying SQL/CQL and NEEL

As Law et al. [LWZ04] show, SQL lacks expressive power for continuous queries on data streams. CQL [ABW06] extends SQL with operators that read or write streams. These operators work as adapters to convert streams into relations, and vice versa. Since CQL is based on SQL, a relation in CQL is an (unordered) set of tuples. During query processing, the temporal ordering of tuples in the input stream may be lost. It is not clear whether SQL based language with set semantics are suitable for real-time event detection and composition. As one of the potential next steps, we could study how to marry SQL/CQL and NEEL.

7.2.8 Decorrelation of NEEL

SQL queries may contain multiple correlated subqueries. When executing nested SQL queries using nested iteration, redundant work is performed largely because of duplicate invocation of the correlated subquery with identical correlation values.

SQL query decorrelation techniques have been proposed to avoid the tuple-at-a-time evaluation imposed by nested iteration. As the inefficiency of executing nested CEP queries is caused by similar reasons as nested SQL queries, we could borrow the state-of-art SQL query decorrelation for CEP queries.

7.2.9 Caching of NEEL

The iterative execution of nested CEP expressions often results in the repeated recomputation of the same or similar results for nested subexpressions as the window slides over the event stream. We can optimize NEEL execution performance by caching intermediate results.

7.2.10 Extend Algebra of NEEL with for-all Semantics

When rewriting double negation over SEQ such as ! SEQ(A, !B, C), we require for all (A, C) events during some time interval, B events must exist in between. Extending algebra of NEEL with for-all semantics may help us in rewriting queries with double negation.

Bibliography

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

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

[Ae09] Mohamed Ali and etc. Microsoft cep server and online behavioral targeting. In *VLDB*, pages 147–160, 2009.

[Ahm04] Ahmed Ayad et al. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD Conference*, pages 419–430, 2004.

[AK00] Balachander Krishnamurthy Att and Balachander Krishnamurthy. On network-aware clustering of web clients. In *ACM SIGCOMM*, pages 97–110, 2000.

[Arv03] Arvind Arasu et al. Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1), 2003.

[BDG+07] Lars Brenna, Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker M. White. Cayuga: a high-performance event processing engine. In *SIGMOD Conference*, pages 1100–1102, 2007.

[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.

[BKMH06] Matthias Brantner, Carl-Christian Kanne, Guido Moerkotte, and Sven Helmer. Algebraic optimization of nested xpath expressions. In *ICDE*, page 128, 2006.

[BP02] J. M. Boyce and D. Pittet. Guideline for hand hygiene in healthcare settings. *MMWR Recomm Rep.*, 51:1–45, 2002.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.

[CGM10] Badrish Chandramouli, Jonathan Goldstein, and David Maier. High-performance dynamic pattern matching over disordered streams. *PVLDB*, pages 220–231, 2010.

[Cha03] Charles D. Cranor et al. Gigascope: A stream database for network applications. In *SIGMOD Conference*, pages 647–651, 2003.

[CHC$^+$06] K. Selçuk Candan, Wang-Pin Hsiung, Songting Chen, Jun'ichi Tatemura, and Divyakant Agrawal. AFilter: Adaptable XML filtering with prefix-caching and suffix-clustering. In *VLDB*, pages 559–570, 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.

[Dan03] Daniel J. Abadi et al. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003.

[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.

[Dou92] Douglas B. Terry et al. Continuous queries over append-only databases. In *SIGMOD*, pages 321–330, 1992.

[Edm67] J. Edmonds. Optimum branchings. In *J. Research of the National Bureau of Standards*, pages 233–240., 1967.

[Est06] Esther Ryvkina et al. Revision processing in a stream processing engine: A high-level design. In *ICDE*, page 141, 2006.

[Fin82] Sheldon Finkelstein. Common expression analysis in database applications. In *SIGMOD*, 1982.

[FJK+05] Michael J. Franklin, Shawn R. Jeffery, Sailesh Krishnamurthy, Frederick Reiss, Shariq Rizvi, Eugene Wu 0002, Owen Cooper, Anil Edakkunni, and Wei Hong. Design considerations for high fan-in systems: The hifi approach. In *CIDR*, pages 290–304, 2005.

[FK08] Ellison R Fitzpatrick K. Compliance of healthcare workers with infection control contact precaution procedures. In *Annual meeting of the Society for Healthcare Epidemiology of America*, pages 1–45, 2008.

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

[GGST86] Harold N. Gabow, Zvi Galil, Thomas H. Spencer, and Robert Endre Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[GHL06] Hector Gonzalez, Jiawei Han, and Xiaolei Li. Flowcube: Constructuing RFID FlowCubes for multi-dimensional analysis of commodity flows. In *VLDB*, pages 834–845, 2006.

[GHQ95] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In *VLDB*, pages 358–369, 1995.

[GÖ05] Lukasz Golab and M. Tamer Özsu. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD Conference*, pages 658–669, 2005.

[GWA+09a] Chetan Gupta, Song Wang, Ismail Ari, Ming C. Hao, Umeshwar Dayal, Abhay Mehta, Manish Marwah, and Ratnesh K. Sharma. Chaos: A data stream analysis architecture for enterprise applications. In *CEC*, pages 33–40, 2009.

[GWA+09b] Chetan Gupta, Song Wang, Ismail Ari, Ming C. Hao, Umeshwar Dayal, Abhay Mehta, Manish Marwah, and Ratnesh K. Sharma. Chaos: A data stream analysis architecture for enterprise applications. In *CEC*, pages 33–40, 2009.

[GWYL05] Bugra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. Adaptive load shedding for windowed stream joins. In *CIKM*, pages 171–178, 2005.

[HCC92] Jiawei Han, Yandong Cai, and Nick Cercone. Knowledge discovery in databases: An attribute-oriented approach. In *VLDB*, pages 547–559, 1992.

[HCD+05] Jiawei Han, Yixin Chen, Guozhu Dong, Jian Pei, Benjamin W. Wah, Jianyong Wang, and Y. Dora Cai. Stream Cube: An architecture for multi-dimensional analysis of data streams. *Distributed and Parallel Databases*, 18(2):173–197, 2005.

[HRK+09] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan J. Demers. Rule-based multi-query optimization. In *EDBT*, pages 120–131, 2009.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, pages 205–216, 1996.

[Jag08] Jagrati Agrawal et al. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2008.

[JD02] Boyce JM and Pittet D. Guideline for hand hygiene in healthcare settings. In *MMWR Recomm Rep*, pages 1–45, 2002.

[Jin05] Jin Li et al. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD Conference*, pages 311–322, 2005.

[Kim82] Won Kim. On optimizing an sql-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, 1982.

[Kla03] Martin Klazar. Bell numbers, their relatives, and algebraic differential equations. *J. Comb. Theory, Ser. A*, pages 63–87, 2003.

[KWF06] Sailesh Krishnamurthy, Chung Wu, and Michael J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006.

[LKH+08] Eric Lo, Ben Kao, Wai-Shing Ho, Sau Dan Lee, Chun Kit Chui, and David W. Cheung. OLAP on sequence data. In *SIGMOD Conference*, pages 649–660, 2008.

[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.

[LMT+05] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, pages 311–322, 2005.

[LRG+10a] Mo Liu, Elke A. Rundensteiner, Kara Greenfield, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. E-cube: Multi-dimensional event sequence processing using concept and pattern hierarchies. In *ICDE*, pages 1097–1100, 2010.

[LRG+10b] Mo Liu, Elke A. Rundensteiner, Kara Greenfield, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. NEEL: The nested complex event language for real-time event analytics. *in BIRTE*, 2010.

[LRG+10c] Mo Liu, Elke A. Rundensteiner, Kara Greenfield, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. Nested complex sequence pattern query processing over event streams: Rewriting and compacting. Worcester Polytechnic Institute, Technical Report in progress, 2010.

[LRG+11a] Mo Liu, Elke Rundensteiner, Kara Greenfield, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. High-performance nested cep query processing over event streams. In *ICDE*, 2011.

[LRG+11b] Mo Liu, Elke A. Rundensteiner, Kara Greenfield, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. E-cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing. *SIGMOD*, 2011.

[LRG+11c] Mo Liu, Elke A. Rundensteiner, Kara Greenfield, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. High-performance nested cep query processing over event streams. *ICDE*, 2011.

[LRR+10] Mo Liu, Medhabi Ray, Elke A. Rundensteiner, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. Processing nested complex sequence pattern queries over event streams. In *DMSN*, 2010.

[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*, pages 274–288, 2008.

[Lup04] Luping Ding et al. Joining punctuated streams. In *EDBT*, pages 587–604, 2004.

[LWZ04] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Query languages and data models for database sequences and data streams. In *VLDB*, pages 492–503, 2004.

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

[Mar99] Marcos Kawazoe Aguilera et al. Matching events in a content-based subscription system. In *PODC*, pages 53–61, 1999.

[ME04] Joris Mihaeli and Opher Etzion. Event database processing. In *ADBIS (Local Proceedings)*, 2004.

[MHM04] Norman May, Sven Helmer, and Guido Moerkotte. Nested queries and quantifiers in an ordered context. In *ICDE*, pages 239–250, 2004.

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

[Mou03] Moustafa A. Hammad et al. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308, 2003.

[MSHR02] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, pages 49–60, 2002.

[MZZ10] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. Efficient pattern matching over event streams. In *VLDB*, 2010.

[Pet03] Peter A. Tucker et al. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng.*, 15(3):555–568, 2003.

[PR08] Paolo Pialorsi and Marco Russo. Programming microsoft linq. 2008.

[Pra94] Praveen Seshadri et al. Sequence query processing. In *SIGMOD*, pages 430–441, 1994.

[RDR$^+$98] Raghu Ramakrishnan, Donko Donjerkovic, Arvind Ranganathan, Kevin S. Beyer, and Muralidhar Krishnaprasad. Srql: Sorted relational query language. In *SSDBM*, pages 84–95, 1998.

[RMCZ06] Esther Ryvkina, Anurag Maskey, Mitch Cherniack, and Stanley B. Zdonik. Revision processing in a stream processing engine: A high-level design. In *ICDE*, page 141, 2006.

[RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pages 249–260, 2000.

[SB03] Heiko Schuldt and Gert Brettlecker. Sensor data stream processing in health monitoring. In *Mobilität und Informationssysteme*, 2003.

[SC75] John Miles Smith and Philip Yen-Tang Chang. Optimizing the performance of a relational algebra database interface. *Commun. ACM*, 18(10):568–579, 1975.

[Sel88] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[Ses98] Praveen Seshadri. Query processing techniques for correlated queries. Technical Report RJ 10129, 1998.

[Shi04] Shivnath Babu et al. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, 2004.

[SLR95] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Seq: A model for sequence databases. In *ICDE*, pages 232–239, 1995.

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

[SrCL+05] Victor Shnayder, Bor rong Chen, Konrad Lorincz, Thaddeus R. F. Fulford Jones, and Matt Welsh. Sensor networks for medical care. In *SenSys*, page 314, 2005.

[sto] I. inetats. stock trade traces. http://www.inetats.com/.

[SW04] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *PODS*, pages 263–274, 2004.

[SZZA01] Reza Sadri, Carlo Zaniolo, Amir M. Zarkesh, and Jafar Adibi. Optimization of sequence queries in database systems. In *PODS*, 2001.

[TFR$^+$09] Dante I. Tapia, Juan A. Fraile, Sara Rodríguez, Juan Francisco de Paz, and Javier Bajo. Wireless sensor networks in home care. In *IWANN (1)*, pages 1106–1112, 2009.

[Uni02] Stanford University. Stream query repository,http://www-db.stanford.edu/stream/sqr/. 2002.

[Vij99] Vijayshankar Raman et al. Online dynamic reordering for interactive data processing. In *VLDB*, pages 709–720, 1999.

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

[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.

[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.

[WZL03] Haixun Wang, Carlo Zaniolo, and Chang Luo. Atlas: A small but complete sql extension for data mining and data streams. In *VLDB*, pages 1113–1116, 2003.

[Yue95] Yue Zhuge et al. View maintenance in a warehousing environment. In *SIGMOD*, pages 316–327, 1995.

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

[ZS02] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *VLDB*, pages 358–369, 2002.

[ZW07] Qi Zhang and Wei Wang. A fast algorithm for approximate quantiles in high speed data streams. In *SSDBM*, page 29, 2007.