

NEEL+: Supporting Predicates for Nested Complex Event Processing

by

Dazhi Zhang

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

August 2012

APPROVED:

Professor Elke A. Rundensteiner, Thesis Advisor

Professor Dan Dougherty, Thesis Reader

Abstract

Complex event processing (CEP) has become increasingly important in modern applications, ranging from supply chain management for RFID tracking to real-time intrusion detection. These monitoring applications must detect complex event pattern sequences in event streams. However, the state-of-art in the CEP literature such as SASE, ZStream or Cayuga either do not support the specification of nesting for pattern queries altogether or they limit the nesting of non-occurrence expressions over composite event types. A recent work by Liu et al proposed a nested complex event pattern expression language, called NEEL (Nested Complex Event Language), that supports the specification of the non-occurrence over complex expressions. However, their work did not carefully consider predicate handling in these nested queries, especially in the context of complex negation. Yet it is well-known that predicate specification is a critical component of any query language. To overcome this gap, we now design a nested complex event pattern expression language called NEEL+, as an extension of the NEEL language, specifying nested CEP queries with predicates. We rigorously define the syntax and semantics of the NEEL+ language, with particular focus on predicate scoping and predicate placement. Accordingly, we introduce a top-down execution paradigm which recursively computes a nested NEEL+ query from the outermost query to the innermost one. We integrate predicate evaluation as part of the overall query evaluation process. Moreover, we design two optimization techniques that reduce the computation costs for processing NEEL+ queries. One, the intra-query method, called predicate push-in, optimizes each individual query component of a nested query by pushing the predicate evaluation into the process of computing the query rather than evaluating predicates at the end of the computation of that particular query. Two, the inter-query method, called predicate shortcutting, optimizes inter-query predicate evaluation. That is, it evaluates the predicates that correlate different query components within a nested query by exploiting a light weight predicate short cut. The NEEL+ system caches values of the equivalence attributes from the incoming data stream. When the computation starts, the system checks the existence of the attribute value of the outer query component in the cache and the predicate acts as a shortcut to early terminate the computation. Lastly, we conduct exper-

imental studies to evaluate the CPU processing resources of the NEEL+ System with and without optimization techniques using real-world stock trading data streams. Our results confirm that our optimization techniques when applied to NEEL+ in a rich variety of cases result in a 10 fold faster query processing performance than the NEEL+ system without optimization.

Acknowledgements

Most of all I would like to thank my research advisor, Professor Elke A. Rundensteiner, whose passion for research, teaching and scientific exploration inspired me in more ways than I can count. She taught me new ways to look at research and encouraged creativity, took the time to read and carefully comment on my papers, and gave me encouragement and practical advice when I needed it most. It was a great privilege to work with her on this thesis.

I also would like to thank the CS department for the two-year TA support, without which I couldn't have manage to finish my study in WPI.

I would like to thank Professor Dan Dougherty for his work on the NEEL+ language design and revision. Those meetings and discussions with Professor Dougherty inspired me to question things, think critically and keep moving on. I also would like to thank him as my thesis reader for his time and valuable suggestions.

I have learned a great deal from my fellow graduate students and friends in DSRG and WPI. I am especially grateful to Mo Liu and Medhabi Ray (PhD students in the Department of Computer Science in WPI), who introduced me to CEP research, gave me an opportunity to collaborate with them and helped me extend out on their work which turned into my thesis.

I want to thank my parents and friends including Chuan Lei and Lei Cao for their continuous presence and support.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problems to be Tackled	3
1.3	Task Accomplished	4
2	Syntax and Semantics of NEEL+ Language	5
2.1	Syntax of NEEL+	5
2.2	Semantics of NEEL+	7
3	Processing Model for NEEL+ Language	11
3.1	System Design	11
3.2	Processing Paradigm	16
3.2.1	Iterative Execution Paradigm	16
3.2.2	Processing Nested Queries with Negation	16
3.2.3	Predicate Testing	17
3.2.4	Query Processing Algorithm	18
3.2.5	Example for Illustrating the Process Algorithm	18
4	Optimization Strategies for Processing NEEL+ Queries	22
4.1	Intra-Query Approach: Predicate Push-in	23
4.2	Inter-query Approach: Shortcutting Inter-query Predicates	25

4.2.1	Discussion	27
5	Cost Analysis for NEEL+ Query Processing	29
6	Experimental Evaluation and Analysis	33
6.1	Experimental Setup	33
6.2	Data Description of Experiment Data Set	33
6.3	Experiments	34
6.3.1	Comparison of NEEL+ System with and without Predicate Pushdown Op- eration	34
6.3.2	Comparison of NEEL+ System with and without Shortcutting Inter-query Predicate Operation	39
7	Related Work	43
8	Conclusions	45
8.1	Summary And Contribution	45
8.2	Future Work	45
8.2.1	Improve the Current System	46
8.2.2	Join Ordering	46
8.2.3	Rewriting	46
8.2.4	Query Decorrelation	46
8.2.5	Memory Awareness Processing	47
8.2.6	Sharing Common Expression	47

List of Figures

1.1	Nested Query Q1	3
3.1	CHAOS Architecture	12
3.2	NEEL+ System Design	13
3.3	Major Class Diagram	13
3.4	Example Query 1	14
3.5	Example Query 2	17
3.6	Example Query 3	19
3.7	Query Plan with Negative Parts Masked	20
3.8	Query Plan with Unmasked Negative Parts	20
4.1	Example Query 4	23
4.2	Example Query 5	24
4.3	Inner Variable V.S. Outer Variable	25
4.4	Data Structure for Equivalence Attribute Value	26
4.5	Example Query 6	26
6.1	Sample of the Stock Trading Data	34
6.2	Varying Positions Predicates	34
6.3	Comparing Execution Times of Queries with Different Predicate Positions between the Basic System and the Basic System with Predicate Push-in	35
6.4	Varying Window Sizes	36

6.5	Comparing Execution Times of Queries Using Different Window Sizes between the Basic System and the Basic System with Predicate Push-in	37
6.6	Varying Query Types	37
6.7	Effect on Execution Times of the Basic System and the Basic System with Predicate Push-in Using Different Query Types	38
6.8	Varying the Attribute Domain Size of the RIMM Event	39
6.9	Comparing Execution Times of the Basic System and the System with SIP Technique by Varying the Attribute Domain Size	40
6.10	Varying the Inter-query Correlated Predicate Position	41
6.11	Comparing Execution Times of the Basic System and the System with SIP Technique by Varying Positions of "shortcut" predicates	42

List of Tables

5.1 Terminology Used in Cost Estimation 29

Chapter 1

Introduction

1.1 Motivation

Complex event processing (CEP) has become increasingly important in modern applications, ranging from supply chain management for RFID tracking to real-time intrusion detection [20, 2, 14]. These monitoring applications submit complex queries to track sequences of events that match a given pattern on real time event streams. These complex patterns often correspond to the arbitrary nesting of sequence operators and the flexible use of negation in such nested sequences. For example, consider reporting contaminated medical equipments in a hospital [4, 7, 6]. 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, of 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 approximate warnings such as “This tool must be disposed”. The query $Q_1 = SEQ(Recycle\ r, Washing\ w, NOT\ SEQ(Sharpening\ s, Disinfection\ d, Checking\ c), Operating\ op)$ Where ($[ID]$ (equality on ID) and $op.ins-type = “surgery”$) in Figure 1.1s 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 contain equality checks on variable attributes and complex negation specifying

the non-occurrence of composite event instances. In this example query, the predicates requires equality on all event IDs of both the positive events and the negative composite event of sharpened, disinfected and checked subsequences.

The state-of-art CEP systems (such as SASE [20] and ZStream [14]) do not support nested CEP queries. Cayuga [2] only allows sub-queries in the FROM clause (of standard SQL [10]). It also doesn't support applying negation over composite event types. While CEDR [3] allows applying negation over composite event types within their proposed language, the execution strategy for such nested queries is not discussed. In short, processing techniques and optimization mechanisms for nested CEP queries have not been proposed by these state-of-the-art solutions.

In a recent work by Liu et al [12], a nested CEP language NEEL(Nested Complex Event Language) was proposed, which supports the nesting of Sequence, AND, OR and Negation expressions. [13] introduces an iterative nested execution strategy for processing nested event queries expressed in *NEEL* [12]. However, their work did not carefully consider predicate handling in these nested queries in the context of negation (i.e. non-occurrence). Yet, most queries in practice indeed involve predicate specifications. For instance, in Figure 1.1, we are looking for operating tools that match the specified pattern. However, without specifying the equality on the ID attribute, the query would return results composed of event instances that might have different id values. For example, a match of the example query could be $\langle Recycle, Washing, Operating \rangle$, with the Recycle event on tool 1, the Washing event on tool 2 and the Operating event on tool 3. Such matchings are clearly not meaningful. The CEP system would trigger a lot of false alarms. Instead, we are only interested in whether a given operating tool is under a particular sequence of actions - meaning the equality predicates among all event instances are critical for the correct semantics in this application context. Due to the presence of non-existential operators (NOT queries) in NEEL, integrating predicates into NEEL queries is not straightforward. For example, in a nested query written in NEEL syntax: $q = SEQ(A, !B, C; A.id = B.id)$, if there is no B existing between A and C, the predicate makes no sense since it cannot be evaluated. Therefore, supporting predicates in nested CEP queries pose several subtle challenges as we will explain below.

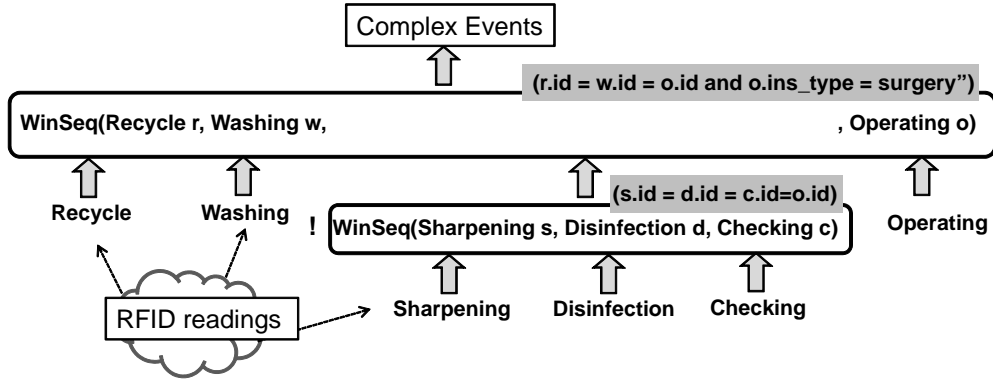


Figure 1.1: Nested Query Q1

1.2 Problems to be Tackled

As stated above, most queries in practice involve predicate specifications. However, there is no existing system that processes a nested CEP query with negative expressions and predicates.

Problem 1: There is no syntax and semantics defining the nested query that supports predicate specification, especially the meaning of predicates under the negation scope. As the example stated at the end of last chapter shows, integrating predicates into nested CEP queries is not straightforward. Therefore, we need to define correct syntax and semantics to address that.

Problem 2: There is no existing system supports processing nested queries with predicates. The existing systems either process flat queries with predicates or nested queries without queries. Therefore, there is also a need for developing a query processing model that correctly processes nested CEP queries with predicates. While the top-down iterative execution paradigm adopted from the NEEL system will be applied here to process the nested CEP queries, we now consider predicate evaluation processing integrated into the iterative process model.

Problem 3: The iterative process integrated with predicate evaluation contains two phases. It first computes the an individual query component and then run the predicate evaluation on the resulting matches. However, this might involve unnecessary computation as the predicates might be able to terminate the computation in an earlier stage without computing the whole matches. Therefore, optimization techniques are needed to reduce the computation costs.

1.3 Task Accomplished

In this thesis we work with a new language, called NEEL+, which extends the NEEL system to support predicates both syntactically and semantically. The language design is based on the work by Prof. Dougherty. To support the processing of the queries expressed in the new language, we design a processing framework that correctly handles these NEEL+ queries. Moreover, we investigate methods that avoid re-computations when dealing with queries with predicates. Lastly, we evaluate the NEEL+ system with optimization techniques against that without optimization techniques. In short, we accomplish the following tasks:

1. **Present Syntax and Semantics of NEEL+:** Define the syntax and semantics of NEEL+ for nested queries with predicates.
2. **Design the Processing Model of NEEL+:** Design a processing model for NEEL+ queries, such as iterative processing and predicate handling.
3. **Develop Optimization Strategies:** Develop optimization approaches for NEEL+ queries, such as intra-query predicate optimization and inter-query predicate optimization
4. **Develop the System:** Implement the system, including the processing engine and the optimization techniques.
5. **Experimental Evaluation:** Compare the performance of optimized NEEL+ against that without optimization.

Chapter 2

Syntax and Semantics of NEEL+ Language

This chapter gives an introduction of the NEEL+ language on which the work of this thesis is built. The language definition is based on the notes by Professor Dan Dougherty from the Computer Science Department of WPI.

2.1 Syntax of NEEL+

To fully understand the language description, one should have the intuition of the terms target variable and free variable. These terms are introduced in order to support meaningful predicates in a query. Target variables are the variables that we match against primitive events in the input stream in order to compute the result of the query. Free variables in a query is the set of variables bound to predicates. The target variables of a query are also free variables, but when looking at a sub-query, the target variables of the outer query might be free in the context of the sub-query.

The syntax of NEEL+ queries is defined as below:

- A simple query is an expression of the form $(A x) : \alpha(x, \vec{y})$ where A is a primitive event type, x is a variable of type A and α is a boolean combination of atomic formulas.

$$FV((A x) : \alpha(x, \vec{y})) = \{x, \vec{y}\} \text{ and } TV((A x) : \alpha(x, \vec{y})) = \{x\}$$

- If q is a query, then $!q$ is a query.

$$TV(!q) = \emptyset \text{ and } FV(!q) = FV(q) \setminus TV(q)$$

- Suppose that q_1, \dots, q_n are queries, then

- $AND(q_1, \dots, q_n)$ is a query, with

- $TV(AND(q_1, \dots, q_n)) = \bigcup \{TV(q_i) | 1 \leq i \leq n\}$

- $FV(AND(q_1, \dots, q_n)) = \bigcup \{FV(q_i) | 1 \leq i \leq n\}$

- Suppose that q_1, \dots, q_n are queries, then

- $OR(q_1, \dots, q_n)$ is a query, with

- $TV(OR(q_1, \dots, q_n)) = \bigcup \{TV(q_i) | 1 \leq i \leq n\}$

- $FV(OR(q_1, \dots, q_n)) = \bigcup \{FV(q_i) | 1 \leq i \leq n\}$

- If q is a query and x is a variable, then

- $(\exists x.q)$ is a query, with

- $TV(\exists x.q) = TV(q) \setminus \{x\}$

- $FV(\exists x.q) = FV(q) \setminus \{x\}$

- Suppose that for each $1 \leq i \leq n$, q_i is a query; then

- $SEQ(q_1, \dots, q_n)$ is a query. If q_i has a ! before it, q_i is called the negative part, otherwise q_i is called the positive part. For every query p in the positive part, and every r in the negative part, $TV(r) \cap FV(p) = \emptyset$.

This restriction amounts to the requirement that no query in the positive part can make reference to the target variable of a query in the negative part.

- $TV(SEQ(q_1, \dots, q_n)) = \bigcup \{TV(q_i) | 1 \leq i \leq k\}$, where q_i belongs to the positive part and k is the size of the positive part

- $FV(SEQ(q_1, \dots, q_n)) = \bigcup \{FV(q_i) | 1 \leq i \leq n\}$

Consider the query Q1 below:

$$Q1 = \exists c : \text{SEQ}((A a), !(B b : b.id = a.id), (C c))$$

When we see from the perspective of the whole query Q1, the only target variable is 'a' and the only free variable is also 'a'. From the perspective of the subquery ((B b) : b.id = a.id) the target variable is 'b', and both 'a' and 'b' are free. In the !-expression !((B b) : b.id = a.id) 'a' is free, but there are no target variables. In SEQ((A a); !((B b) : b.id = a.id); (C c)) the target variables are 'a' and 'c', and these are the free variables as well.

Take Query Q2 as example:

$$Q2 = \text{SEQ}((A a, !B b, !\text{SEQ}(C c, D d), E e))$$

Q2 is a nested query. The target variables and free variables of Q2 are both 'a' and 'e'. The positive part of Q2 are 'A a' and 'E e', while the negative part of Q2 are 'B b' and 'SEQ(C c, D d)'.

Take another query Q3 as example:

$$Q3 = \text{SEQ}((A a: a.id = b.id), !\text{SEQ}(B b, C c), D d)$$

Query Q3 is not a valid query, as we stated that no query in the positive part can make reference to the target variable of a query in the negative part. Since b is a free variable from the negative part, the predicate a.id = b.id must be attached to the variable in the !SEQ query.

2.2 Semantics of NEEL+

In order to introduce the semantics of NEEL+, we first introduce the concept of binding. Let H be an event history, which is an ordered set of primitive event instances. Then a binding is a mapping from the target variables to the event instances in the history, with timestamp of the events in increasing order. For example, let H be $[a_1 a_2 b_3 c_4]$, where the subscript stands for time

stamp. Let a query be $\text{SEQ}(A\ a, B\ b)$, where a and b are the target variables. Then $\langle a_1, b_3 \rangle$ is a binding and $\langle a_2, b_3 \rangle$ is another one. Both of them have event B 's timestamp later than event A 's timestamp. The meaning of a query q over a history is defined as a set of bindings. If the query only contains positive parts, the bindings will only include the target variables of the query. In the case of a query containing negative parts, the concept of extension of a binding will be introduced. Let q be a query and let β_0 be a binding whose domain is $\text{TV}(q)$. We define a binding β as an extension of binding β_0 if the domain of β is a superset of β_0 . By superset, we mean that β is a binding with domain equal to $\text{TV}(p) \cup \text{TV}(r)$, where p stands for the positive part of q and r stands for the negative part of q without the negation ("!"). If such extension is found, the binding β_0 will be rejected.

We can use a game between player P(os) and player N(eg) to better explain the semantics. Let the history be $H = [a_1, b_2, e_3, c_4, d_5]$, where the subscript stands for the timestamp. Let the query be $Q = \text{SEQ}(A\ a, \text{!SEQ}(B\ b, C\ c), D\ d)$. Moves by players are as below: 1. For player P(os), mask all negative expressions (only consider the positive part of Q). P(os) plays $(a- > a_1, d- > d_5)$ on $Q_1 = \text{SEQ}(A\ a, D\ d)$ 2. For play N(eg), mask the negation symbol ("!") only. N(eg) plays $(a- > a_1, b- > b_2, c- > c_4, d- > d_5)$ on the $Q_2 = \text{SEQ}(A\ a, \text{SEQ}(B\ b, C\ c), D\ d)$. The winning condition of this game is that the other player cannot move. In this example, N(eg) plays the last step $(w- > a_1, x- > b_2, y- > c_4, z- > d_5)$ while the P(os) cannot make another move, as the P(os) player has no more negation to mask in order to find a binding against N(eg)'s move. The P(os) player loses and therefore $(w- > a_1, z- > d_5)$ is not returned.

More precisely, let q be a query and let β be a binding over history H whose domain includes $\text{TV}(q)$. We will shortly define what it means for a binding β to satisfy a query q in history H , denoted $H, \beta \models q$. Anticipating this definition we define the meaning of a query as follows.

Definition 1 (Semantics of Queries) *Let q be a query and let H be a history. The meaning of q on H , written $q[H]$, is the set of those bindings β_0 such that*

- *the domain of β_0 is a subset of $\text{TV}(q)$, and*

- there is an extension β of β_0 such that $H, \beta \models q$.

It remains to define $H, \beta \models q$. Since this will be an inductive definition, we need to define $H, \beta \models q$ even when the domain of β includes free variables of q that are not target variables.

Definition 2 Let H be a history, let q be a query, and let β be a binding over H . We define

$$H, \beta \models q$$

by induction on the level of nesting of negations, with a sub-induction on the size of q .

- Case: q is $(A x) : \alpha(x, \vec{y})$

$H, \beta \models q$ if $\alpha(x, \vec{y})$ is true in H under β .

- Case: q is $AND(q_1, \dots, q_n)$

$H, \beta \models q$ if for each i , $H, \beta \models q_i$

- Case: q is $OR(q_1, \dots, q_n)$

$H, \beta \models q$ if for some i , $H, \beta \models q_i$

(This OR case is the fact that the domain of β doesn't have to include all the TV q_i , since as long as there is a q_i that has matches, the OR would have return value.)

- Case: q is $(\exists x.p)$

Let β^{-x} be the binding obtained by removing x from the domain of β . Then $H, \beta \models q$ if there is an extension β^+ of β^{-x} such that $H, \beta^+ \models p$.

- Case: q is $SEQ(q_1, \dots, q_n)$

– For each i , let β_i be the binding for q_i . Let C_i be the range of β_i , which is a sequence of timestamps in increasing order. $C_i.ts$ and $C_i.te$ are defined as the beginning timestamp and ending timestamp.

Now we say that $H, \beta \models SEQ(q_1, \dots, q_n)$ if for each $1 \leq i < n$:

1. $H, \beta \models Q_i$ for each i ¹, $C_i.te \prec C_{i+1}.ts$
- Let the positive part of q be $\langle p_1, \dots, p_k \rangle$. Then we say that $H, \beta \models SEQ(q_1, \dots, q_n)$ if
1. $H, \beta \models SEQ(p_1, \dots, p_k)$, and
 2. for no extension β^+ of β do we have $H, \beta^+ \models \mathbf{masknegation}(SEQ(q_1, \dots, q_n))$,
where **masknegation()** masks the negation ("!") in the negative parts of q .

For example, consider reporting contaminated medical equipments in a hospital [4]. 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). The query in Fig 2 expresses the 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. When written in NEEL+ syntax, the query is as below: $SEQ((Recycle\ r:r.id = w.id), (Washing\ w:w.id = o.id), !SEQ((Sharpening\ s:s.id = d.id), (Disinfection\ d:d.id = c.id), (Checking\ c:c.id = o.id)), Operating\ o)$. The target variable binding of the above query is $\langle r, w, o \rangle$. Suppose we have found the binding β_0 , the extension β of β_0 is binding of the query formed by masking the negation. In this case, the target variable of the new formed query would be $\langle r, w, s, d, c, o \rangle$. Let the history be $H = [r_1, w_2, s_3, d_4, c_5, o_{10}]$. Therefore we would have binding β_0 as $\langle r_1, w_2, o_{10} \rangle$ and β as $\langle r_1, w_2, s_3, d_4, c_5, o_{10} \rangle$. When such β is found, the binding β_0 would be rejected since the negation indicates that such an extension β is not supposed to be found.

¹yes, we mean β , not β_i

Chapter 3

Processing Model for NEEL+ Language

3.1 System Design

We first briefly introduce the architecture of CHAOS [8], upon which we built our system. Figure 3.1 shows the CHAOS platform and its integration with the enterprise applications. When data streams from sensors are read into the input stream queues of the CHAOS platform, the first unit provides data reduction functionality to summarize or significantly reduce the data volume to be processed by the following units. The Complex Event Processing component processes the refined data stream and searches for patterns that match the user defined query.

Our focus is on the Complex Event Processing component. The NEEL+ system is built upon the CHAOS platform with the capability of processing nested CEP queries with predicates. The design of the NEEL+ system can be seen in Fig 3.2, which fits in the Complex Event Processing component in CHAOS architecture.

Query Parser parses the user input query from its textual format into the system's data structures.

Physical Plan Generator contains two sub-components: Plan Optimizer and Execution Plan Generator. The Execution Plan Generator component takes in the parsed query and generates the

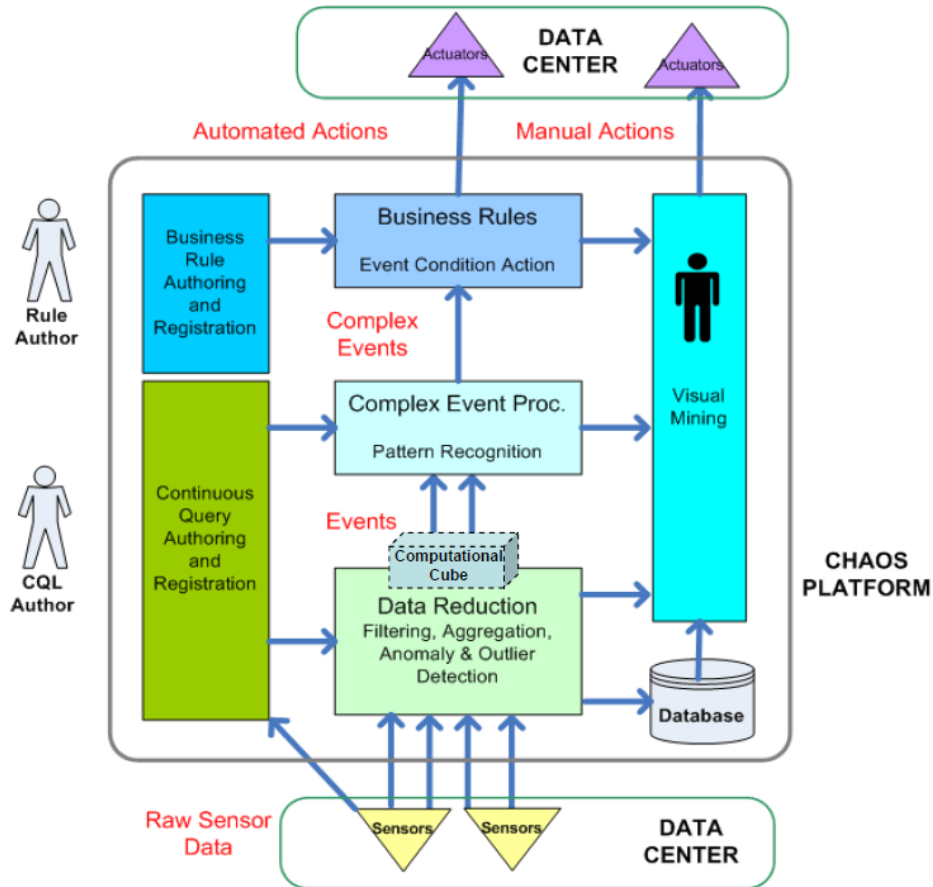


Figure 3.1: CHAOS Architecture

processing plan for the query, including the query processing order, predicate evaluation strategy from the Plan Optimizer and the related data structures to support the process. The Plan Optimizer takes in the statistics gathered by the Query Executor and decides what predicate process strategy to use. The main goal of the optimization techniques is to reduce computation. This is achieved by deciding where to place the predicates, when to evaluate them and what kind of data to cache to obtain more efficient processing.

Query Executor contains three sub-components. The Physical Operator processes the query on the input stream using an iterative paradigm, which will be introduced below. The Predicate Evaluator is a subcomponent of the Query Executor and is responsible for processing the predicates. The Statistic Gatherer continuously gathers statistics from the input data stream or the computational

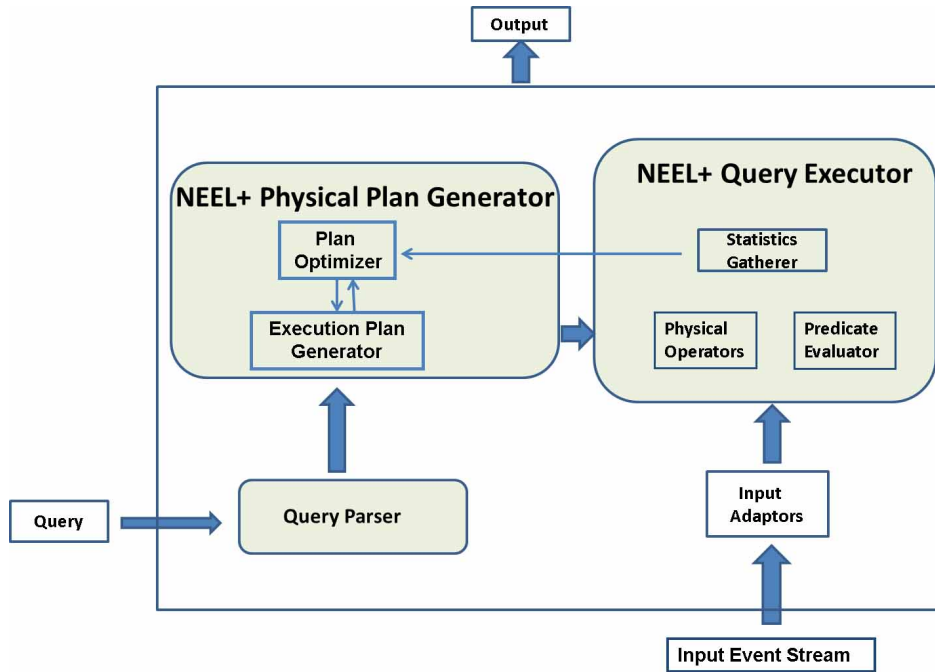


Figure 3.2: NEEL+ System Design

results and feeds them back to the plan optimizer to generate or update optimization strategies.

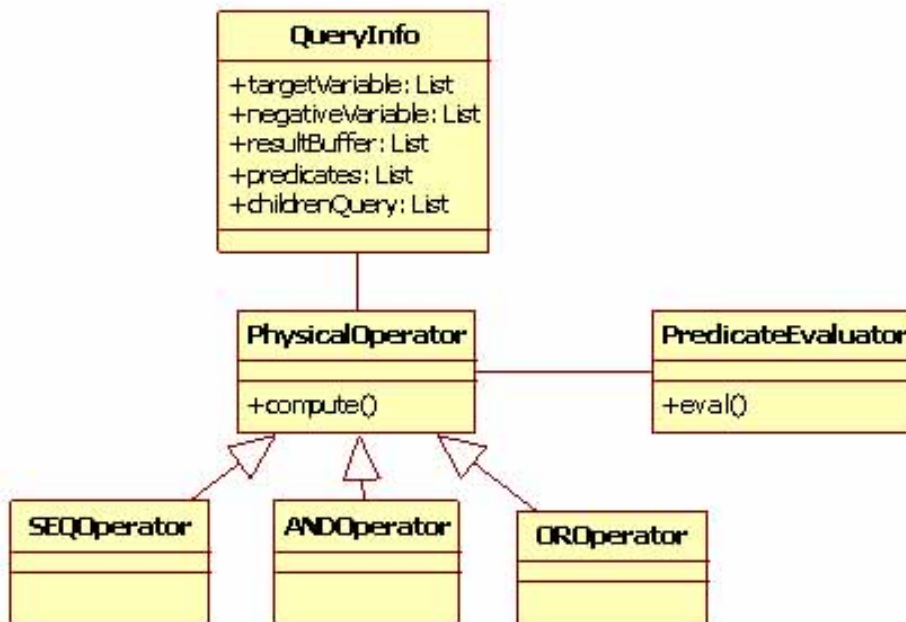


Figure 3.3: Major Class Diagram

```
SEQ (DELL d, !SEQ((YHOO y: y.id = d.id), RIMM ii), !AMAT a, MSFT s,
      SEQ((IPIX p: p.id = i.id), CSCO c), (INTC i: i.id = d.id))
```

Figure 3.4: Example Query 1

Figure 3.3 shows the major class diagram of the *NEEL+* system. We will use the query in Figure 3.4 to illustrate the interactions among the classes. Figure 3.4 shows a nested query, with a `!SEQ` sub-query nested between variable `d` and `a` and a `SEQ` sub-query nested between variable `s` and `i`. The target variables of the outer `SEQ` query are `d`, `a`, `s` and `i`. The target variables of the first `SEQ` sub-query (i.e. `!SEQ`) are `y` and `ii`. The target variables of the second `SEQ` sub-query are `p` and `c`.

QueryInfo: The input textual query will be parsed into the `QueryInfo` data structure by the `Query Parser` component introduced in the system design. Each `Physical Operator` and the information in it is corresponding to a `QueryInfo` object, with information like target variables, negative variables, children queries, predicates and result buffer for this query. In Figure 3.4, three `QueryInfo` objects will be created and stored since there are three `SEQ` queries in total.

- **targetVariable:** The target variables of a query are the variables that we match against primitive events in the input stream in order to compute the result of the query. They are the variables without a `!"` in front of them. In Figure 3.4, the target variables of the outer `SEQ` query are `d`, `s`, and `i`. The target variables of the first `SEQ` sub-query (when considered without `!"`) are `y` and `ii`. The target variables of the second `SEQ` sub-query are `p` and `c`.
- **negativeVariables:** The negative variables are those variables that are not being returned, but used to filter out the results of the target variable matching. They are the variables with a `!"` in front of them. In Figure 3.4, only the first query has a negative variable, namely, `a`.
- **childrenQuery:** It stores the `QueryInfo` objects that are nested under the current `QueryInfo` object. In Figure 3.4, the `childrenQuery` data structure of the first `QueryInfo` object contains

two objects, which are the QueryInfo objects of !SEQ and SEQ sub-queries. The children-Query data structures of the two sub-queries are both empty since the two sub-queries don't have any nested queries.

- **predicates:** It stores the predicates attached to the variables of the current queryInfo object. In Figure 3.4, the outer SEQ query has one predicate $i.id = d.id$ attached to the variable i . The first SEQ sub-query has a correlated predicate $y.id = d.id$ attached to the variable y . Lastly, the second SEQ sub-query also has a correlated predicate $p.id = i.id$ attached to the variable p
- **resultBuffer:** It stores the matching results from the target variables of the current queryInfo object to the input stream. In Figure 3.4, the result buffer of the outer SEQ query stores the results when computing those $\langle d, s, i \rangle$ sequences. The result buffer of the first SEQ sub-query stores the result when computing those $\langle y, ii \rangle$ sequences. The result buffer of the third query stores those $\langle p, c \rangle$ sequences. The query executor will manage these buffer based on the query plan. It will terminate the computation and clear the buffer or keep the buffer for joining purpose based on the status of the negative query's resultBuffer

PhysicalOperator: Each PhysicalOperator object is associated with one single QueryInfo object. The PhysicalOperator is the generalization of three concrete classes: SEQOperator, ANDOperator and OROperator. In the example, the nested sub-queries are all SEQ expressions. Therefore, each QueryInfo is associated with a SEQOperator object. The concrete operator is responsible for processing its corresponding query.

PredicateEvaluator: It is utilized by the PhysicalOperator object during the computation of a valid binding. It takes in the predicate objects and the computed bindings and returns a boolean value indicating whether the predicate evaluation passes or not.

3.2 Processing Paradigm

3.2.1 Iterative Execution Paradigm

Following the principle of nested query execution for SQL queries [17, 16, 19, 18], in NEEL+ we execute nested CEP queries using an iterative execution strategy. First we maintain run time stacks that keep track of incoming events. We define the last event of the outermost query expression as the *trigger event*, which will trigger the computation to start. The computation starts when the trigger event arrives and finds all possible matches within the predefined window size. The main processing flow is to evaluate the outer query first followed by its inner sub-queries. After computing an outer query, the predicates associated with this query will be evaluated. When computing an inner query, the computed results from the outer query will be passed down for the correlated predicate evaluation. Also, more stringent window constraints from outer queries will be passed down to inner queries. These sub-queries compute results involving events within the substream constrained by the constraint window. The results of positive inner queries are passed up and joined with the results of the outer query. The outer sequence result is filtered if the result set of any of its negative sub-queries is not empty.

3.2.2 Processing Nested Queries with Negation

Negative queries merely behave as complex filters, filtering out the positive components if a binding for the negative part is found within a stipulated time interval. For example, a flat SEQ query has a negative !A between positive B and C event types, b and c are stored in the targetVariable object while a is stored in the negativeVariable object of the current queryInfo object. We first evaluate the query without the negative expression, i.e., compute the bindings for $\langle b, c \rangle$. We store the resulting bindings in the resultBuffer object of the current queryInfo object. Then we iterate the bindings in this resultBuffer to check if an A event occurred between the qualified B and C events. If such an occurrence exists, such bindings for $\langle b, c \rangle$ are discarded.

A sub-query as a whole could also be negated. For example, $\text{SEQ}(A a, ! \text{SEQ}(B b, C c), D$

d). For each binding of outer query SEQ(A a, D d), we pass it down for the search of SEQ(B b, C c) bindings occurring between such A and D events. If none exists, then the passed down $\langle a, d \rangle$ binding is returned, otherwise it is filtered out.

3.2.3 Predicate Testing

We have three types of predicates: unary predicates, binary predicates related to the variables within the same query component and binary predicates across different query components, which we will refer to as correlated predicates. For the first and second type of predicates, predicate testing will be done when the target variable bindings of the query component have been found. The constructed binding will be kept if it satisfies the predicate conditions. Later it would join the bindings passed up by sub-queries to form the final result for the whole query. Otherwise the binding will be filtered out. For the last type, namely correlated predicates, information from the corresponding variables on the upper query component is required. In order to evaluate these predicates, the bindings from the corresponding query component are stored and passed down to the computation of the children queries. Therefore, when the children queries are being computed, they can access the binding information from the upper query components.

```

SEQ(Recycle r, (Washing w: w.id = r.id),
! SEQ((Sharpening s: s.id = o.id),
      (Disinfection d : d.id = s.id),
      (Checking c: c.id = s.id)),
(Operating o: o.id = w.id, o.ins-type = "surgery"))

```

Figure 3.5: Example Query 2

Consider the query in Figure 3.5. We will illustrate how to evaluate the three types of predicates using Figure 3.5. Let $H = [r_1 w_1 s_1 d_1 c_1 o_1 r_2]$, where the subscript stands for the id value and their timestamp is in increasing order. When trigger event o_1 arrives, we start constructing the binding for the outermost query component. Each event contains several attributes such as event

name, id, timestamp and so on. When the sequence $\langle r_1, w_1, o_1 \rangle$ is constructed, it will be stored in the resultBuffer object of the QueryInfo object corresponding to the outermost SEQ query. The predicate $o.ins - type = \text{"surgery"}$ will be first tested by the PredicateEvaluator. If the ins-type of o_1 is not equal to surgery, the constructed binding will be filtered. If the unary predicate is satisfied, then the binary predicate on this query component such $o.id = w.id$ and $w.id = r.id$ will be tested. They are all satisfied as their id values are all 1. Therefore, the binding $\langle r_1, w_1, o_1 \rangle$ will be stored in the resultBuffer object of the current QueryInfo object. Since the binding of the outer query is found, it will be passed down to the nested subquery for further computation. The subquery is computed under the time constraint bound by the w_1 and o_1 event pair. We start with the last event c_1 in the inner query and repeat the same sequence construction process. When the binding $\langle s_1, d_1, c_1 \rangle$ is found, we start to evaluate the predicates of the current query component. For $c.id = s.id$ and $d.id = s.id$, it's the same as described above. For evaluating $s.id = r.id$, we require the information from the binding passed down by the outer query. They are all satisfied since their id values are all 1. Since the sub-query is a negative query, the found binding will not be stored in the resultBuffer object. However, binding $\langle r_1, w_1, o_1 \rangle$ will be rejected since we have found a match in the nested negative subquery and the resultBuffer object of the outermost SEQ query component will be cleared.

3.2.4 Query Processing Algorithm

The process algorithm is listed in Algorithm 1.

3.2.5 Example for Illustrating the Process Algorithm

We will use the example query in Figure 3.4 to illustrate the processing algorithm for better understanding of how the processing algorithm works.

The query plan of the nested query with three SEQ operators is shown in Figure 3.6. As defined in the previous chapter, the trigger event is the incoming event that starts the computation of the query. The trigger event of this query is INTC. The computation starts when the trigger event

Algorithm 1 processQuery

Require: QueryInfo q { t is the trigger tuple, left and right are the bounding timestamps}

```
1: ResultBuffer parentResult, finalResult
2: parentResult = computeSingleQuery( $q$ ) {computeSingleQuery() computes the bindings of a
   individual query component}
3: parentResult = evaluatePredicate( $q$ , parentResult)
   {if there is no valid binding, return null}
4: if parentResult.size = 0 then
5:   return null;
6: end if
7: if  $q$ .subQuery.size()  $\neq$  0 then
8:   HashMap<childId, ResultBuffer> childResult
   {recursively call the compute processs}
9:   for each  $r_i$  in parentResult do
10:    for each  $q_i$  in  $q$ .SubQuery do
11:      childResult.put( $q_i$ .Id, processQuery( $q_i$ ))
12:    end for
13:    finalResult.add(join(childResult,  $r_i$ )) {join the current binding with the list of chil-
   dResults, and add it to the finalResult list}
14:  end for
15: else
16:   return parentResult
17: end if
18: return finalResult
```

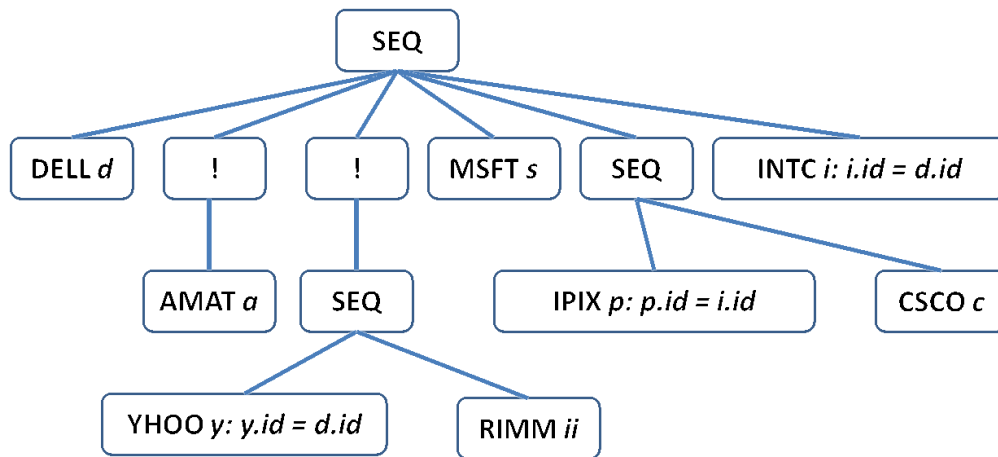


Figure 3.6: Example Query 3

comes.

The steps can be enumerated as follows:

1. Compute positive parts of the query. Mask all the negative parts of the current query under

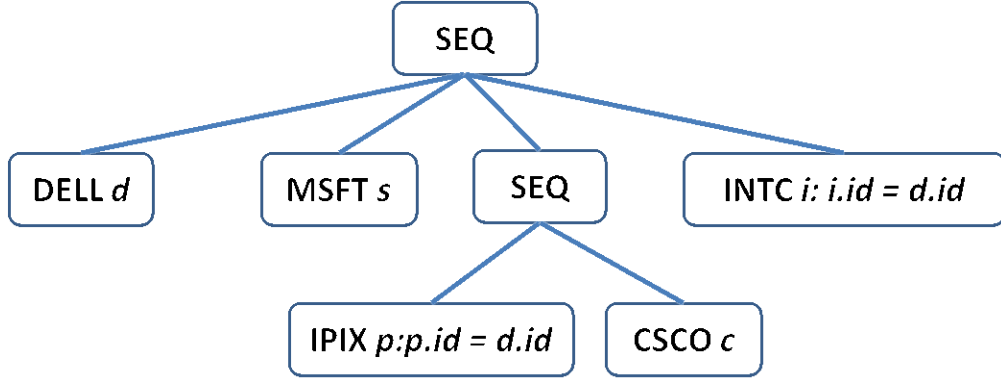


Figure 3.7: Query Plan with Negative Parts Masked

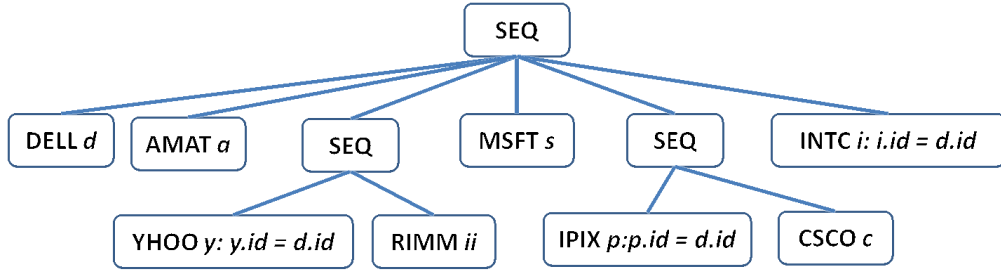


Figure 3.8: Query Plan with Unmasked Negative Parts

computation and construct the new query plan. Compute the query according to the new formed query plan. In Query 3 (Figure 3.6), when taking away all the negative parts, namely $\neg AMAT a$ and $\neg SEQ((YHOO y : y.id = d.id), RIMM ii)$, the query plan is shown in Figure 3.7. Then we compute the bindings according to the query plan above, we get a list of bindings for the target variables $\langle d, s, p, c, i \rangle$. When computing the positive parts, namely $SEQ(DELL d, MSFT s, SEQ((IPIX p : p.id = d.id), CSCO c), (INTC i : i.id = d.id))$, we will have the following situations:

- Flat query. For example, $SEQ(DELL d, MSFT s, INTC i)$. In this case, we use the processing method introduced in SASE [20] to compute the bindings for this query when the trigger event of type INTC arrives.
- A query with nested positive queries. For example, $SEQ(DELL d, MSFT s, SEQ(IPIX p, CSCO c), INTC i)$. In this case, we first compute the sequences $\langle d, s, i \rangle$. For each result obtained, we compute the nested query, which is $SEQ(IPIX p, CSCO c)$,

using the same method we discuss in the flat query situation. Then we join both result sets by the time constraints from the upper query, in this case, timestamps of variable d and r are the constraints for the nested SEQ sub-query $SEQ(IPIX\ p, CSCO\ c)$. If the sub-queries are further nested, we recursively repeat the same process until the lowest sub-query has been computed.

2. Evaluate predicates for the computed bindings and filter out bindings that fail the predicate evaluation. In Figure 3.7, we have two predicates, $i.id = d.id$ and $p.id = d.id$. If both predicate tests return true, the binding will be kept, otherwise it will be filtered.
3. Now only mask the negations ("!") from Query 3(Figure 3.6), we get a new query plan as shown in Figure 3.8. Then we compute bindings for this query plan, which are the extensions of the bindings computed based on Figure 3.7. For each of the bindings computed in step 2, we compute the extensions and we will get a set of extensions that match the input stream to the target variables in query plan 2, i.e. $\langle d, a, y, ii, s, p, c, i \rangle$. Those bindings computed in step 2 that have extensions are the ones potentially to be filtered. Now it remains to run the predicate tests for the extensions. In this case, there is only one predicate to be evaluated in the new computed extensions, namely $y.id = d.id$.
4. Evaluate predicates for the computed extensions. If the extensions pass the predicate evaluation, the binding from Figure 3.7 should be filtered out, because the predicates are under the negation("!") condition. In Figure 3.8, the extensions are computed based on the bindings passed down from the the computational result of Figure 3.7. Therefore, when we evaluate the predicate $y.id = d.id$, we can access the information of the variable d to perform the evaluation. If the predicate test returns true, the binding will be filtered since we have found the extensions with all predicate conditions satisfied.

Chapter 4

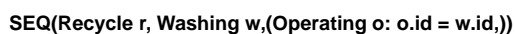
Optimization Strategies for Processing

NEEL+ Queries

The throughput of a given query is mainly decided by the data distribution of the data stream and the resulting selectivity of predicates. In ZStream [14], they use the tree-based query plans for both the logical and physical representation of the squery patterns. Several equivalent physical tree plans with different evaluation costs can be constructed for one logical CEP query. They capture the runtime behavior of a physical plan. They then choose the optimal one with lowest cost.s In this thesis, since we focus on supporting predicates for nested complex event processing, our optimization techniques fall on optimizing the predicate processing in nested CEP context. Also, due to the high volume of high speed-data streams and the large window sizes in many large deployments of applications, many results will be generated in a sliding window. Therefore, the main factor that affects the throughput of the CEP queries is effectively filtering via the predicates. This is where we develop optimization techniques to improve the system performance. We accordingly do not focus on the issue of join ordering tackled in ZStreem [14], but instead use the NFA-based approach—a fixed order evaluation as done in NEEL [12] and SASE [20]. Our goal is to reduce the computation costs and to avoid re-computation. To achieve this, two techniques are proposed.

4.1 Intra-Query Approach: Predicate Push-in

The processing algorithm in Chapter 3 evaluates predicates for each query after the bindings of that query have been computed. However, it is not necessary to always compute the whole bindings first since the predicate might be related to variables in the early computation stage. If we evaluate the predicates at an early stage, we can avoid further computation if the predicate evaluation were to fail and thus filter the partial binding. Hence, instead of evaluating predicates after the computation of the whole binding, we optimize the query processing by pushing the predicate evaluation into the computation process. Consider the example in Figure 4.1:



```
SEQ(Recycle r, Washing w,(Operating o: o.id = w.id,))
```

Figure 4.1: Example Query 4

The default process is to compute bindings for $\langle r, w, o \rangle$ first, and then to test the predicate $o.id = w.id$. As we have noticed, the predicate only includes Washing and Operating events. According to the computation paradigm introduced in Chapter 3, we start computing backwards when we meet the trigger event “Operating”. When the computation reaches the “Washing”, we will have the information for running the predicate $o.id = w.id$. If this condition is not satisfied, we can filter out this result without further computing the remainder of this binding. The idea is to push the predicate evaluation into the computation process and thus have the predicates evaluated as soon as the computation stage has the data the predicates need. This approach saves the computation costs by early terminating the computation process.

In this example, the predicate is attached to the event type “Operating”. Here we cannot evaluate the predicate $o.id = w.id$ because the “Washing” event is not yet been identified in our partial binding. This means we need to check if there is any predicate ready to run whenever the computation reaches a new event type. To avoid repeatedly checking available predicates, we now introduce the **predicate relocation** procedure.

There are two kinds of predicates in NEEL+, unary predicates and binary predicates. When

relocating predicates, we only consider binary predicates because a unary predicate is only related to one single variable and thus no relocation is needed. Since a binary predicate is related to two variables, the predicate can be attached to either variable by our syntax (see Section 2). To support predicate relocation, the optimizer introduced in the Chapter 3 will determine where the predicate should be placed at compile time. It starts by checking each variable that has predicates attached. If the predicates attached to a variable has reference to another variable that is in the later computation stage, namely we cannot access the information of the that variable, the optimizer will will relocate the predicates to that variable. Therefore, we can have all predicates evaluated when the computation reaches the variable that the predicates are attached to.

Next we discuss predicates correlating different query components of a nested query. Consider a correlated predicate example:

SEQ(Recycle r, Washing w,
SEQ(Sharpening s, Disinfection d),
(Operating o: o.id = d.id))

Figure 4.2: Example Query 5

Figure 4.2 is a query with a positive sub-query. The predicate is valid as the $d.id$ in the outer query references to the variable d in the positive sub-query. When the predicate is attached to the variable in the outer query, it cannot be tested even if the results of the outer query are computed, as it requires data from the inner query. In order to be consistent with the idea: whenever we reach a variable that contains predicates, it has all the data required to evaluate the predicates, we need to relocate the predicate to the variable in the inner query. When the computation reaches variable d in the inner query, we can run the predicate test immediately. If this is not satisfied, we can stop the computation rather than going on to compute $\langle s, d \rangle$.

Lastly, we will discuss the case of a query with a negative sub-query. If a predicate is correlating the outer and inner queries, predicate relocation is unnecessary because the *NEEL+* syntax makes it necessary to attach such predicate to the variables in the negative query. As we have discussed in Chapter 2, no query in the positive part can make reference to the target variable of a query in

the negative part. Therefore, if there is such correlated predicate, it will always be attached to the variable in the negative part.

4.2 Inter-query Approach: Shortcutting Inter-query Predicates

In some cases [15], we might collect or estimate statistics about the incoming data stream. This provides foundation for an informed decision to change our execution plan in order to achieve a less time consuming execution.

In order to introduce our technique, we first introduce two terms-inner variable and outer variable.

Definition 3 *Inner Variable:* The variable in the predicate that references to the variable of a inner sub-query component.

Definition 4 *Outer Variable:* The variable in the predicate that references to the variable of a outer query.

For example, in Figure 4.3, "ii" is the inner variable while "d" is the outer variable.

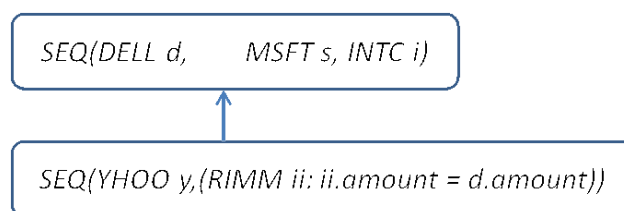


Figure 4.3: Inner Variable V.S. Outer Variable

Now we introduce our shortcutting inter-query predicate (SIP) optimization technique. For a nested query with inter-query predicates, suppose we have collected statistics about the CEP query that the domain size of the inner variable attribute (DSI) is much smaller than domain size of the outer variable attribute (DSO), for example, $\frac{DSI}{DSO} < 20\%$. When the inner variable's corresponding event comes, we buffer the different attribute values of the event in the data structure shown

in Figure 4.4. During the computation started from the trigger event, when the outer variable's corresponding event comes, we first check if the attribute value exists in the cache. If not, we can terminate the computation without completing the whole nested query. If the incoming attribute value exists in the cache, we need to keep on computing as the basic system does. Therefore, in this technique, the worst case is to compute like the basic system does plus spending extra costs on maintaining the cache. However, given certain statistics, we might greatly reduce computation.

Variable	<Value, Timestamp>

Figure 4.4: Data Structure for Equivalence Attribute Value

As the window keeps sliding, we need to maintain the data structure that stores the attribute values of the variables that the predicate references to. The maintenance of the data structure consists of three terms: insertion, update and purge.

(1) **Insertion:** When the inner variable's corresponding event comes and if the attribute value is not yet in the data structure in Figure 4.4, we insert the value with the corresponding event's timestamp into the hashtable as $\langle Value, Timestamp \rangle$ entity.

(2) **Update:** When the inner variable's corresponding event comes and if the attribute value exists in the data structure in Figure 4.4, retrieve the $\langle Value, Timestamp \rangle$ pair based on that attribute value and update the "Timestamp" with that of the incoming event.

(3) **Purge:** Whenever a new event comes, we will purge out those attribute values with expired timestamps in the the data structure in Figure 4.4.

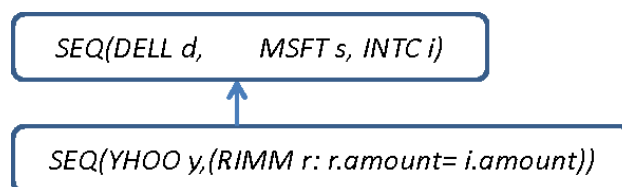


Figure 4.5: Example Query 6

We illustrate this technique using the example in Figure 4.5. Let the data stream be $[DELL_1 DELL_2 YHOO_3 IPIX_4 CSCO_5 RIMM_6 INTC_7 INTC_8 INTC_9]$, where the subscript stands for the timestamp. Suppose these tuples all have the attribute transaction amount. As shown in Figure 4.5, the predicate is $r.amount = i.amount$. Let the attribute value of $INTC_7$, $INTC_8$ and $INTC_9$ be 100, 200 and 300, and the attribute value of $RIMM_6$ be 100. Therefore, when $RIMM_6$ comes, the attribute value 100 will be cached. When we are computing the matching pattern $\langle d, s, i \rangle$ for the outer query and when we reach $INTC$, we first check if the attribute value of $INTC$ exists in the buffer. For $INTC_8$ and $INTC_9$, since the attribute values are 200 and 300, we can immediately terminate the computation since further computation of the inner query won't get any matching due to the inequality of the attribute values. For $INTC_7$, since it matches the value in the buffer, we need to further compute the inner query. Overall, if the inner query is selective enough, this optimization can greatly reduce the computation.

4.2.1 Discussion

The effectiveness of the SIP technique is based on the statistics of the incoming data streams. We will discuss the Pros and Cons as below:

Pros: When the attribute's domain size of the inner variable is much smaller than that of the outer variable, our technique would be able to shortcut more, namely it can terminate more computation at an earlier stage instead of finishing the computation then filter out the resulting bindings. Therefore, it is worthwhile taking this technique under the condition that the attribute's domain size of the inner variable is much smaller than that of the outer variable.

Cons: When the attribute's domain size of the inner variable is not much different from that of the outer variable, our technique might not be as effective. We spend extra costs maintaining the cache for those attribute values, yet most of the time the system cannot shortcut the predicates since the inner variable and outer variable have more or less the same attribute domain sizes. Therefore, for most of the time the system with the SIP will perform as the NEEL+ system without the optimization technique does, yet it costs time maintain the cache. Maintain the data structure that stores the

attribute values would be the main overhead of this technique.

Chapter 5

Cost Analysis for NEEL+ Query Processing

This chapter shows the algorithmic costs of the basic system and the optimization techniques. It assumes that average statistics of the data have been collected. It will provide a theoretical understanding of the efficiency of each technique based on the underlying data statistics.

Table 5.1: Terminology Used in Cost Estimation

Term	Definition
$C_{compute(q_i)}$	The cost of computing results for a query q_i independently
$NumE$	Number of total events received so far
$NumRE$	Number of relevant events received of the types in query set Q
P_{e_i}	Selectivity of predicates attached to variable. If there is no predicates attached to e_i , the value is set to 1.
Pt_{E_i, E_j}	Selectivity of the implicit time predicate of subsequence (E_i, E_j) . The default value is set to 1/2.
$ S_i $	Number of tuples of type E_i that are in time window TW_P . This can be estimated as arrival rate * TW_P * P_{e_i}
$ S_{q_i} $	The number of results for a query q_i
PC_{e_i}	The cost of evaluating predicates attached to e_i . If there is no predicates attached to e_i , the value is set to 1.
SC_{e_i}	The selectivity of the "shortcut" correlated predicate. If there is no such predicate attached to e_i , the value is set to 1.

Table 5.1 shows the cost factors in pattern evaluation. The CPU processing costs for an event pattern are composed of three main terms: the cost to insert the input data (C_{insert}), the cost to generate results ($C_{compute}$), and the cost to purge (C_{purge}) (Equation 5.1).

$$C_{q_i} = C_{insert(q_i)} + C_{compute(q_i)} + C_{purge(q_i)} \quad (5.1)$$

(1) Cost of insert (C_{insert}): The cost of insert C_{insert} remains unchanged independent of the chosen query evaluation method. Thus it can henceforth be ignored.

(2) Cost of compute ($C_{compute}$): Computation costs depend on the actual pattern evaluation strategy in use and is considered in depth below.

(3) Cost of purge (C_{purge}): The cost of purge C_{purge} remains unchanged independent of the chosen query evaluation method. Thus it can henceforth be ignored.

Compute Cost for the Basic System. For a flat query $q_i = \text{SEQ}(E_1 e_1, E_2 e_2, \dots, E_i e_i, \dots, E_n e_n)$, using stack-based pattern evaluation, $C_{compute(q_i)}$ is formulated in Equation 5.2. The equation is composed of two parts. The first part models the costs needed for computing the results of the query and the second part models the costs needed for evaluating the predicates attached to each variables.

$$\begin{aligned} C_{compute(q_i)} &= (|S_n| * |S_{n-1}| * Pt_{E_n, E_{n-1}} + |S_n| * |S_{n-1}| * Pt_{E_n, E_{n-1}} * |S_{n-2}| * Pt_{E_{n-1}, E_{n-2}} + \dots) \\ &\quad + |S_{q_i}| * \left[\prod_{i=1}^n PC_{e_i} \right] * \left[\prod_{i=1}^n P_{e_i} \right] \\ &= \left(\sum_{i=n}^2 |S_{i-1}| * \left[\prod_{j=i}^n |S_j| * Pt_{E_j, E_{j-1}} \right] \right) + |S_{q_i}| * \left[\prod_{i=1}^n PC_{e_i} \right] * \left[\prod_{i=1}^n P_{e_i} \right] \end{aligned} \quad (5.2)$$

Iterative Nested Execution. For a nested query q_i , $q_i.root$ represents the outermost query and $q_i.child_j$ represents its j th child. $C_{compute_{q_i}}$ consists of computation costs for q_i 's outermost query $q_i.root$, computation costs for $q_i.child_j$ $j=1\dots n$ and costs for joining each child query with its parent query.

$$C_{compute(q_i)}^{nested} = C_{compute(q_i.root)} + |S_{q_i.root}| * \left(\sum_{j=1}^{number\ of\ q_i.child} C_{compute(q_i.child_j)}^{nested} \right) \quad (5.3)$$

Basic System with Predicate Push-in Technique. For a flat query q_i , $C_{compute_{q_i}}$ consists of computation costs for q_i with the predicate evaluation pushed into the computation process. The equation is shown as below:

$$\begin{aligned} C_{compute(q_i)}^{Push-in} &= |S_n| * PC_{e_n} * P_{e_n} * |S_{n-1}| * PC_{e_{n-1}} * P_{e_{n-1}} * Pt_{E_n, E_{n-1}} + |S_n| * PC_{e_n} * P_{e_n} * |S_{n-1}| * \\ &\quad PC_{e_{n-1}} * P_{e_{n-1}} * Pt_{E_n, E_{n-1}} * |S_{n-2}| * PC_{e_{n-2}} * P_{e_{n-2}} * Pt_{E_{n-1}, E_{n-2}} + \dots \\ &= \left(\sum_{i=n}^2 |S_{i-1}| * PC_{e_{i-1}} * P_{e_{i-1}} * \left[\prod_{j=i}^n |S_j| * PC_{e_j} * P_{e_j} * Pt_{E_j, E_{j-1}} \right] \right) \end{aligned} \quad (5.4)$$

For a nested query q_i , the computation costs of q_i would be similar to Equation 5.3. The equation is shown as in Equation 5.5

$$C_{compute(q_i)}^{nested} = C_{compute(q_i.root)}^{Push-in} + |S_{q_i.root}| * \left(\sum_{j=1}^{number\ of\ q_i.child} C_{compute(q_i.child_j)}^{nested} \right) \quad (5.5)$$

Basic System with SIP Technique

For flat query $q_i = \text{SEQ}(E_1 e_1, E_2 e_2, \dots, E_i e_i, \dots, E_n e_n)$, using stack-based pattern evaluation, $C_{compute(q_i)}$ is formulated in Equation 5.6. The equation is composed of three parts, the first part is the cost of computing the results of the query, the second part is the cost of evaluating the predicates attached to each variables and the third part denotes the maintenance of the cache for

the equivalence attribute value for the "shortcut" predicates'.

$$\begin{aligned}
C_{compute(q_i)}^{SIP} &= (|S_n| * SC_{e_n} * |S_{n-1}| * SC_{e_{n-1}} * Pt_{E_n, E_{n-1}} + |S_n| * SC_{e_n} * |S_{n-1}| * SC_{e_{n-1}} * Pt_{E_n, E_{n-1}} \\
&\quad * |S_{n-2}| * SC_{e_{n-2}} * Pt_{E_{n-1}, E_{n-2}} + \dots) + |S_{q_i}| * \left[\prod_{i=1}^n PC_{e_i} \right] * \left[\prod_{i=1}^n P_{e_i} \right] + C_{Maintenance} \\
&= \left(\sum_{i=n}^2 |S_{i-1}| * SC_{e_{i-1}} * \left[\prod_{j=i}^n |S_j| * Pt_{E_j, E_{j-1}} \right] \right) + |S_{q_i}| * \left[\prod_{i=1}^n PC_{e_i} \right] * \left[\prod_{i=1}^n P_{e_i} \right] + C_{Maintenance}
\end{aligned} \tag{5.6}$$

(1) Cost of Maintaining the Equivalence Attribute Value ($C_{Maintenance}$): is maintained whenever the event that the inner variable of an inter-query predicate references to arrives. The maintenance cost of the cache consists of two parts: the cost to insert the equivalence attribute value (C_{insert}), and the cost to purge (C_{purge}). The cost of insert C_{insert} takes constant time as it only checks the existence of the attribute value and insert or update the value and timestamp accordingly. The purge process will purge each attribute value in the cache with an expired timestamp. An iteration is needed in order to purge all expired timestamps. Therefore, the cost of purge C_{purge} is related to the number of equivalence attribute value in the cache.

For a nested query q_i , the computation costs of q_i would be similar to Equation 5.3.

$$C_{compute(q_i)}^{nested} = C_{compute(q_i.root)}^{SIP} + |S_{q_i.root}| * \left(\sum_{j=1}^{number\ of\ q_i.child} C_{compute(q_i.child_j)}^{nested} \right) \tag{5.7}$$

Chapter 6

Experimental Evaluation and Analysis

6.1 Experimental Setup

The NEEL+ framework is implemented based on the HP stream management system CHAOS [8] using Java. We run the experiments on Intel Pentium IV CPU 2.8GHz with 4GB RAM with Microsoft Windows 7 operating system. We evaluated our techniques using the real stock trades data from [1]. Comparisons are made between the basic processing technique and the processing with predicate pushdown and between the basic processing technique and the processing with inter-query predicate optimization, based on the overall cumulative execution time. That is we will run the query over the stock data and plot the cumulative execution time against the cumulative number of results produced. The queries used in the experiments are shown in the following sections. The window size is specified when the user submits the query. By default it is kept to 100 seconds unless specified otherwise.

6.2 Data Description of Experiment Data Set

The data contains stock ticker, timestamp and trade amount information [1]. The data contained stock ticker, timestamp and trade amount information. The portion of the trace we used contained 10,000 event instances. It is in the form of a text file which is read by a separate thread to simulate

the incoming data stream. A snapshot of a portion of the data is shown in Figure 6.1, where the first column stands for the stock ticker, the second column stands for the timestamp measured in millisecond and the last column stands for the trade amount.

Event Type	Timestamp	Transaction Amount
DELL	25259.326	130
YHOO	25259.430	110
YHOO	25260.130	200
IPIX	25260.132	250
CSCO	25260.134	400
RIMM	25261.135	120
MSFT	25261.200	150
AMAT	25261.250	500
MSFT	25261.300	810
INTC	25261.426	210
	⋮	

Figure 6.1: Sample of the Stock Trading Data

6.3 Experiments

6.3.1 Comparison of NEEL+ System with and without Predicate Pushdown Operation

Effect of Different Predicate Positions

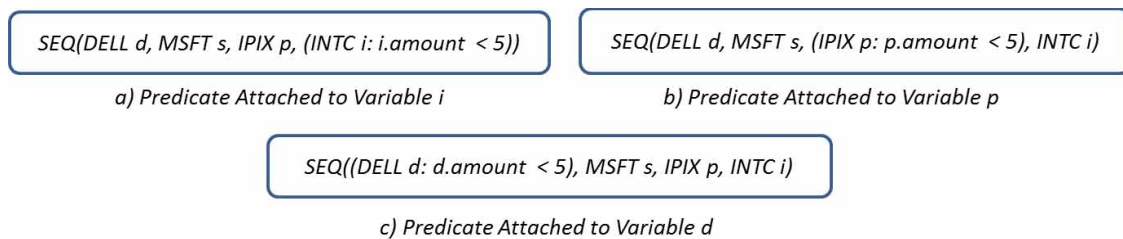


Figure 6.2: Varying Positions Predicates

Figure 6.2 shows three query plans with the same flat query but the predicates are associated with different variables. As the computation starts from the rightmost variable of a query, we

expect that the closer the predicate is attached to the rightmost event, the more execution time our technique will save compared to the basic system. The Window size is kept constant at 100 seconds. We compare the predicate pushdown technique against the basic iterative processing technique. As complex event patterns are detected over the event stream, results are outputted for every triggering event which in this case are all “INTC” events. The cumulative execution time at that instant is recorded on the Y-axis against the cumulative number of results on the X-axis.

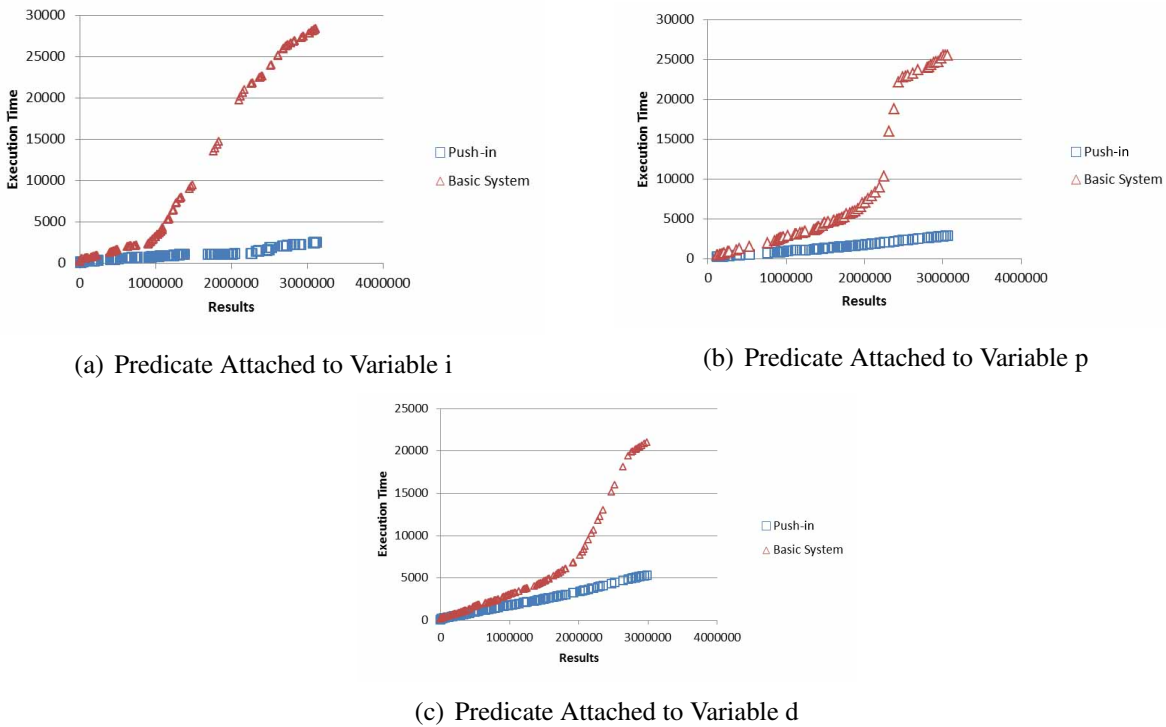


Figure 6.3: Comparing Execution Times of Queries with Different Predicate Positions between the Basic System and the Basic System with Predicate Push-in

Figures 6.3 a, b and c demonstrate our predicate pushdown technique reduces the execution time in all three cases. As expected, our technique reduces the execution time most efficiently for Query 6.2 a. It outperforms the basic system by 700%. The execution time on Query 6.2 b is 300% less compared to the basic system, while on Query 6.2 c, our technique saves the least, which is 80% more efficient than the basic system. As stated in the beginning of Chapter 4, many results will be generated in a sliding window. The main factor that affects the throughput of the CEP queries fall on the predicate selectivity. Therefore, we can conclude that no matter where the

predicate is located, as long as it is selective, the pushdown technique will always help improve the performance due to the early termination it brings to the query process.

Effect of Different Window Sizes

```
SEQ(DELL d, MSFT s, (IPIX p: p.amount > i.amount), (INTC i: i.amount > 100))
```

Windows: 10s, 100s and 500s

Figure 6.4: Varying Window Sizes

The flat query in Figure 6.4 is configured by substituting three different Window sizes of 50, 100 and 500 seconds. We compare the execution time of the basic system with predicate push-in technique and basic system itself. As the window sizes increase, the matching candidates generated in each window will increase. Thus more candidates will be filtered by the predicates, which means termination will happen more often for more tuples. Therefore, we expect the execution time savings ratio will also increase as the window sizes increase. In each experiment, we compute and output the result patterns for every triggering event INTC. The cumulative execution time at that instant is recorded on the Y-axis against the cumulative number of results on the X-axis.

Figures 6.5 a, b and c demonstrate the improvement of our predicate push-in techniques on the basic system. With the window sizes increasing, the basic system with predicate push-in outperforms the basic system by 4 folds, 21 folds and 60 folds on average respectively. As expected, the larger the window size is, the more the predicate push-in technique wins. As a larger window will potentially contain more matching candidates, this leads to the predicate push-in technique having more often chances to succeed in the early termination of the computation process.

Effect of Different Query Types

Figure 6.6 shows three nested query plans, where Figure 6.6 a is a query with a positive nested query, Figure 6.6 b is the same outer query with a negative nested query and Figure 6.6 c cor-

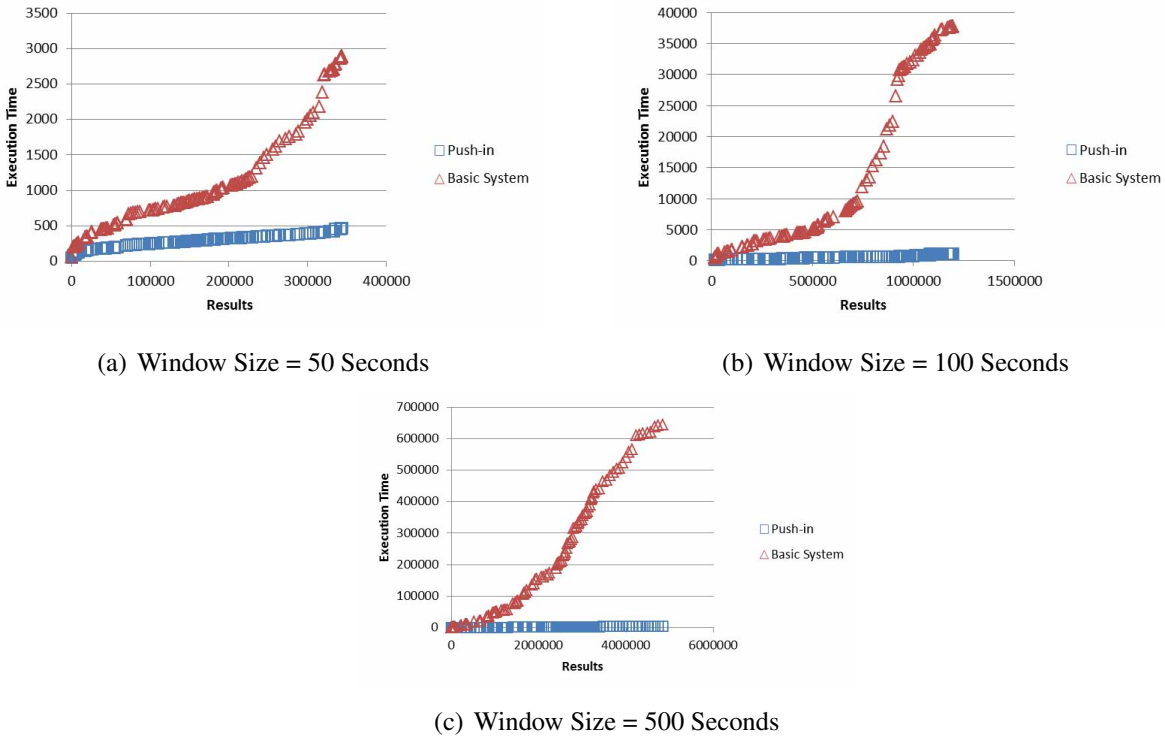


Figure 6.5: Comparing Execution Times of Queries Using Different Window Sizes between the Basic System and the Basic System with Predicate Push-in

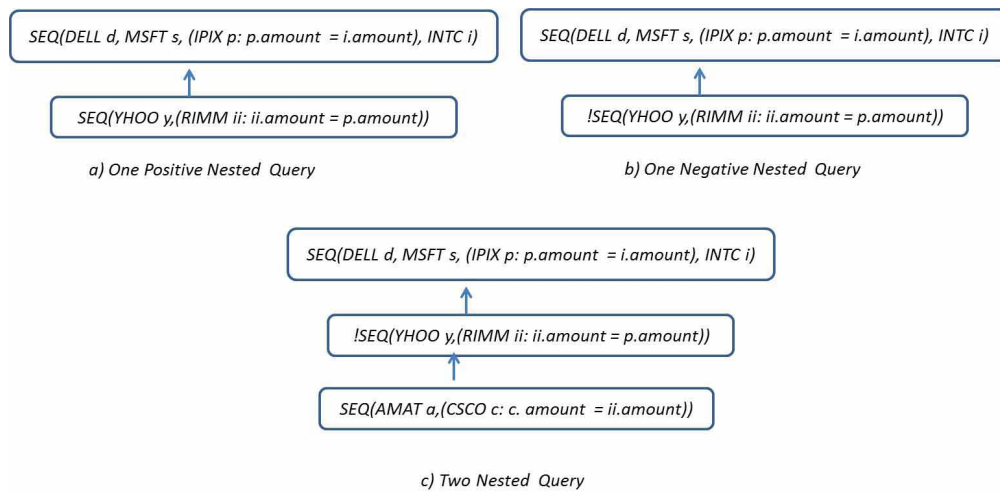
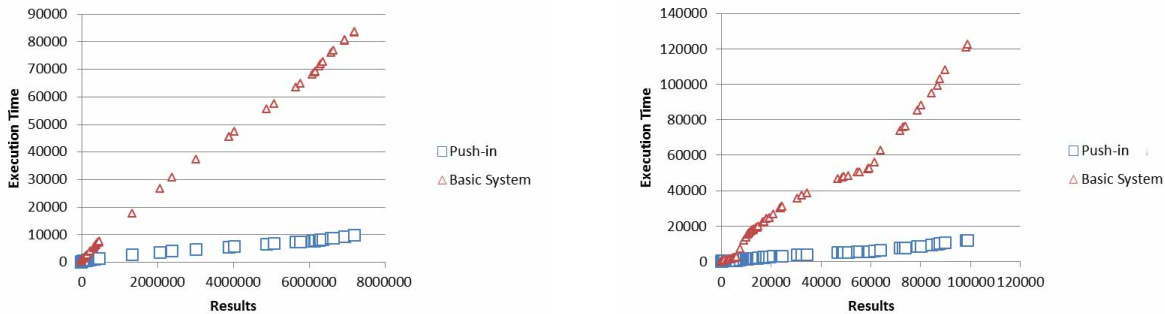


Figure 6.6: Varying Query Types

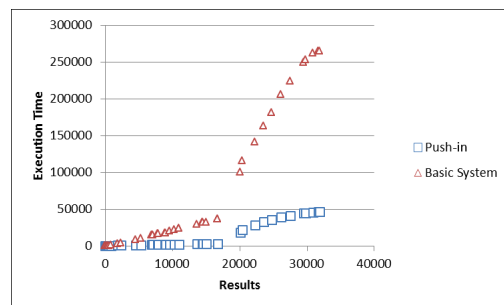
responds to a three layer query with a negative query as the first sub-query and a positive query as the second sub-query. As the push-in technique optimizes each individual query component in a nested query, we expect that it would reduce the execution time for different query component

combinations compared to the basic system. The window size is kept at 100 seconds in all cases. We compute and record the cumulative execution time on the Y-axis against the cumulative number of results on the X-axis.



(a) A Query with a Positive Nested Sub-query

(b) A Query with a Negative Nested Sub-query



(c) A Query with a Positive Nested Sub-query and a Negative Sub-query

Figure 6.7: Effect on Execution Times of the Basic System and the Basic System with Predicate Push-in Using Different Query Types

Figure 6.7 shows the cumulative execution time versus the cumulative number of results of different types of queries. As expected, the predicate push-in technique reduces the execution time in all three cases. The reduction ratios are 5 fold, 7 fold and 7 fold on average respectively. The experiment demonstrates that the predicate push-in reduces the execution time for a rich variety of types of queries, varying from different types of nested sub-queries to different number of nested levels.

6.3.2 Comparison of NEEL+ System with and without Shortcutting Interquery Predicate Operation

Effect of Different Attribute Domain Sizes

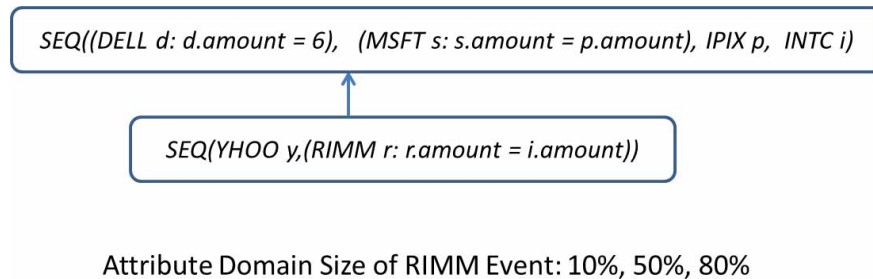
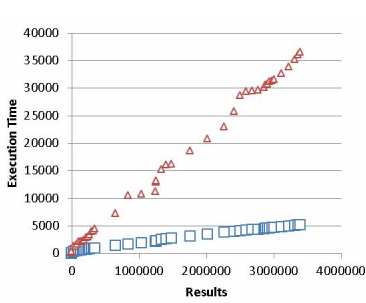


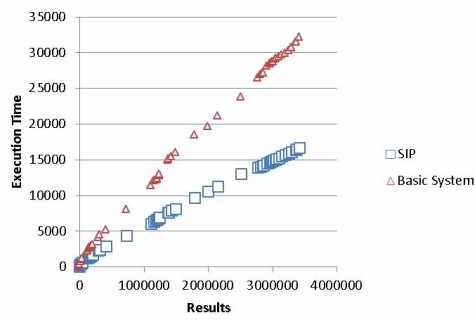
Figure 6.8: Varying the Attribute Domain Size of the RIMM Event

Figure 6.8 shows a flat query which is run by varying the attribute domain size of the event RIMM from the inner query. The attribute domain size of RIMM in each case is 10%, 50% and 80% of the attribute domain size of INTC respectively. As stated in Chapter 4, the smaller the inner variable domain size is, the more often the system would have an opportunity to shortcut the predicates to early terminate the computation. Therefore, we expect to see that by using SIP technique in the basic system, the execution time will reduce when the attribute domain size of the inner variable is small compared to that of the outer variable. We also expect the SIP technique reduces most execution time for the 10% domain size query compared to the basic system, followed by the 50% domain size query, then the 80% domain size query. The cumulative execution time is recorded and plotted on Y-axis while the cumulative number of results is recorded and plotted on X-axis. The window size is kept 100 seconds in each query.

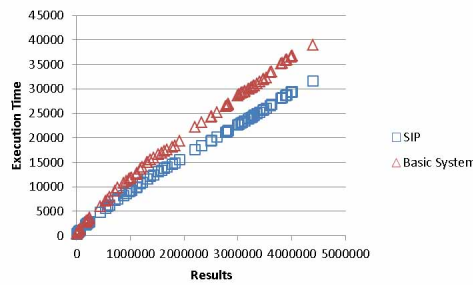
Figures 6.9 a, b and c show that our SIP technique reduces the execution time compared to execution time obtained from the basic system in all three cases. As expected, in Figure 6.9 the system with SIP wins the basic system on execution time by 400%, while in Figure 6.9 it outperforms the basic system on execution time by 100% and in Figure 6.9 c it wins by 25%. As stated above, the smaller the inner variable domain size is, the more often the system would have an opportunity to shortcut the predicates to early terminate the computation. Therefore, when the



(a) Inner Variable Attribute Domain Size/Outer Variable Attribute Domain Size: 10%



(b) Inner Variable Attribute Domain Size/Outer Variable Attribute Domain Size: 50%



(c) Inner Variable Attribute Domain Size/Outer Variable Attribute Domain Size: 80%

Figure 6.9: Comparing Execution Times of the Basic System and the System with SIP Technique by Varying the Attribute Domain Size

inner variable’s domain size is only 10% of the outer variable’s domain size, our technique saves most while in the 80% case, our technique saves the least compared to the 10% and 50% cases.

Varying the "Shortcut" Predicate Position

Figure 6.10 shows three queries with the same nested sub-queries but the "shortcut" predicate is located in different places. The "shortcut" predicates in Figure 6.10 a, b and c all have the outer variable referencing to the same variable in the root query, while the inner variable references to the variable of the RIMM event in a different query level. As the predicate’s position gets deeper, the basic system would have to compute more till it meets the predicate, so that it can filter out unsatisfied results. Therefore, we expect that the deeper the "shortcut" predicate it is, the more the optimization technique will outperform the basic system in terms of execution time saving ratio. The domain size of the RIMM’s attribute is kept at 30% of the domain size of the INTC’s attribute.

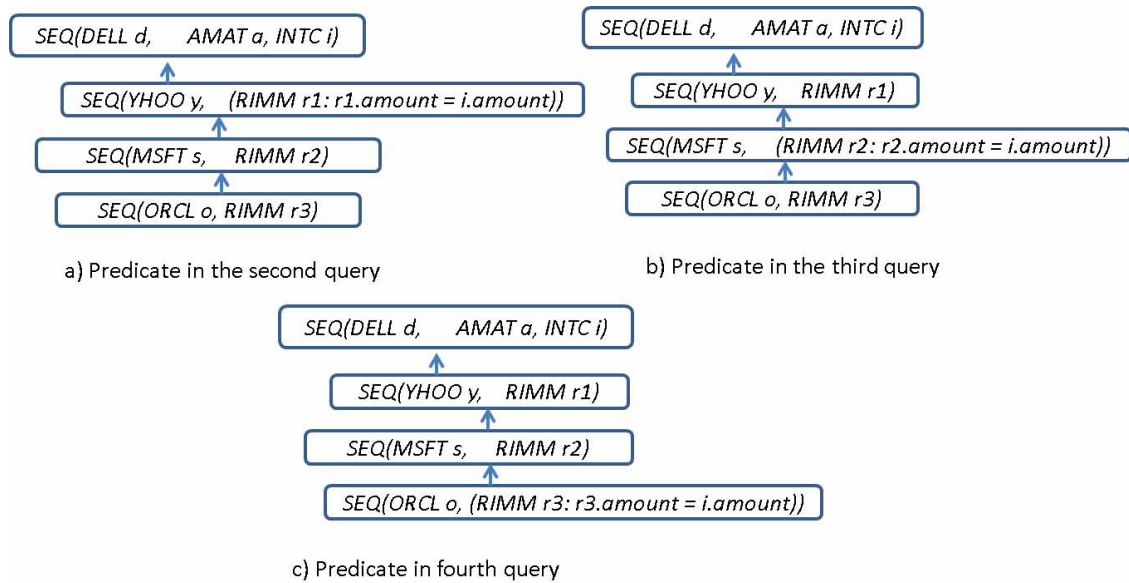
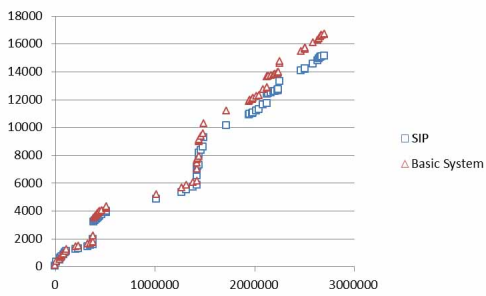


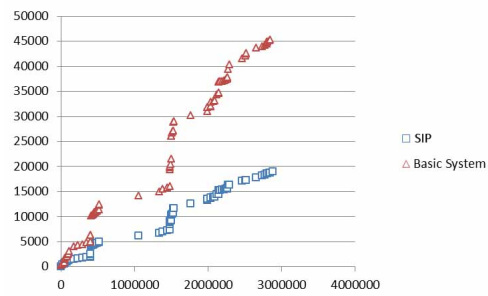
Figure 6.10: Varying the Inter-query Correlated Predicate Position

The window size is kept constant at 100 seconds. We compare the execution time of our technique against the basic system. The cumulative execution times are recorded on the Y-axis against the cumulative number of results on the X-axis.

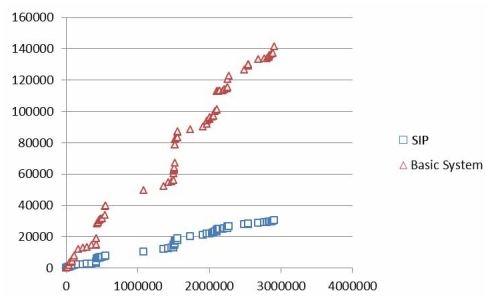
The Figures 6.11 a, b and c show the comparison between the SIP technique and the basic system by varying the "shortcut" predicate position. We observe that the system with SIP wins over the basic system on execution time by 8%, 130% and 340% on average respectively. As expected, the winning ratio of the SIP technique over the basic system on execution time shows a rising trend when the inner variable references to the variable in deeper sub-query. As stated above, when the predicate's position gets deeper, the basic system would have to compute more till it meets the predicate in order to filter out unsatisfied results. Therefore, we expect that the deeper the "shortcut" predicate it is, the more computation cost would be saved as we skip more query components. As a result, the optimization technique will save more when the "shortcut" predicate is located in deeper query components compared to shallower query components.



(a) Sub Query Length = 2



(b) Sub Query Length = 3



(c) Sub Query Length = 4

Figure 6.11: Comparing Execution Times of the Basic System and the System with SIP Technique by Varying Positions of "shortcut" predicates

Chapter 7

Related Work

The existing CEP systems [20, 3, 12] support the execution of only flat sequence queries.

SASE [20] and Cayuga [2] proposed an important processing model for CEP which is based on the nondeterministic finite state automata. SASE supports novel language features such as negation, and demonstrates performance gain in processing complex event queries compared to traditional data stream processing system TelegraphCQ. [2] proposed an SQL like language called the Cayuga Event Language which comprised of traditional select, project and join over multiple streams and also temporal joins of sequences. However, SASE only supports negation over primitive event types. Cayuga [2] does allow sub-queries in the FROM clause, but it does not discuss applying negation over composite event types.

As an extension to the work in SASE [20], SASE+ [9] was proposed which supported Kleene star over event streams. However this too did not tackle nesting of composite event types and they considered only flat queries with predicates.

Zstream [14] also had a similar language which expressed sequences and negation over primitive events. They consider the ordering of CEP queries using a tree-based query plan – similar to join ordering in traditional relational databases. ZStream doesn't consider optimization for queries with predicates.

CEDR [3] which is Microsoft's CEP language which allows the specification of negation of

composite patterns. However, it is not completely flexible nesting and also the execution strategy for such nested queries is not discussed. In [13] Liu et al describes an iterative strategy to process Nested CEP queries, but it does not consider much on queries with predicates.

SQL is the standard language for data retrieval and manipulation in relational database systems. One of the most powerful features of SQL is nested queries. Theoretically, a query can have an arbitrary number of subqueries nested within it. Since it is usually inefficient to directly execute nested queries in their original form, optimization of nested queries has received considerable attention. One approach concentrates on unnesting nested queries [10], rewriting a nested query into a flat form.

[5] proposes an approach to unnest nested SQL queries using hash join each sub-query in a uniform manner, regardless of predicate or level. They process nested queries independently and then joining the results from different levels by the correlated predicates. Consequently, algorithms such as complex query decorrelation [17] have been proposed to decorrelate the query. However, existing decorrelation algorithms deal with only a limited class of queries.

In [11] Liu et al developed a set of equivalence rules for rewriting nested *NEEL* expressions. They then proposed a step-wise procedure that apply these rewriting rules to transform a nested CEP query into an equivalent nonnested query thus opening the opportunity for query optimization. However they are not able to rewrite all nested queries into unnested normal forms and their rewriting rules are often limited by the presence of predicate correlations.

Chapter 8

Conclusions

8.1 Summary And Contribution

This thesis focuses on designing, implementing and optimizing the processing of NEEL+ queries. In particular, we designed and implemented a processing paradigm that correctly processes simple queries, nested queries and queries with predicates. The iterative processing integrated with predicate evaluation can handle queries with unary predicates, binary predicates within the same query and binary predicates correlating different query components in a nested query. We also design and implement certain optimization techniques for handling queries with predicates. We then experimentally compare the optimized methodologies against standard iterative processing technique for CPU processing time. Our optimized strategy wins by a large margin over the basic processing. The performance of the optimization techniques varies under different conditions. We discussed the performance of each method under varying conditions.

8.2 Future Work

For future work, we plan to extend our study in the following directions.

8.2.1 Improve the Current System

Currently, the system doesn't handle the situation that negative expressions appear at the beginning or the end. We will add support to this special case in the future. Also, we have the syntax and semantics for AND and OR expression, but we don't have the implementation for that kind of queries. We will also address that in the future to support various types of queries.

8.2.2 Join Ordering

In our work, we do not focus on the problem of optimizing via join ordering [14] and use the a fixed order processing from the rightmost event to the leftmost event of a query. Therefore we mainly focus on predicate optimization. In the future, we will look into the combination of both optimization to obtain more optimal execution plans for NEEL+ queries.

8.2.3 Rewriting

We had implemented the rewriting rules in NEEL [11] before building up our NEEL+ system. However, [11] is not able to rewrite all nested queries into unnested normal forms and their rewriting rules are often limited by the presence of negations and predicate correlations. We will develop new rewrite rules that can correctly handle negations and correlated predicates. Then we will adapt the current rewriter to our NEEL+ system to help improve the processing performance.

8.2.4 Query Decorrelation

Complex SQL queries used in decision support applications often include correlated subqueries. Significant research efforts [17, 10, 5] have been devoted to the optimization of nested queries by decorrelating them. In principle, such problem could also be applied to advanced CEP queries. We will consider views/caches and joins between separate views and a query, in order to avoid repeatedly computing sub-queries.

8.2.5 Memory Awareness Processing

In our work, we assume here that memory is an unlimited resource. However that is not practical and in that case we would have to look into the issue of selectively caching some results based on statistics.

8.2.6 Sharing Common Expression

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 typically focuses on static relational databases and identifies common subexpressions among queries such as common joins or filters. In principle, such problem could also be applied to advanced CEP queries. We will study the sharing of common CEP expressions and will also take into account the predicates in such common expressions, in order to reduce re-computation.

Bibliography

- [1] I. inetats. stock trade traces. <http://www.inetats.com/>.
- [2] A. J. Demers et al. Cayuga: A general purpose event monitoring system. In *Conference on Innovative Data Systems Research (CIDR)*, pages 412–422, 2007.
- [3] R. S. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *Conference on Innovative Data Systems Research (CIDR)*, pages 363–374, 2007.
- [4] J. M. Boyce and D. Pittet. Guideline for hand hygiene in healthcare settings. *Morbidity and Mortality Weekly Report (MMWR)*, 51:1–45, 2002.
- [5] B. Cao and A. Badia. A nested relational approach to processing sql subqueries. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 191–202, New York, NY, USA, 2005. ACM.
- [6] J. et al. Wireless sensor networks for in-home healthcare: Potential and challenges. In *Proceedings of HCMDSS Workshop*, 2005.
- [7] V. S. et. al. Sensor networks for medical care. In *Proceedings of SenSys*, 2005.
- [8] C. Gupta, S. Wang, I. Ari, M. C. Hao, U. Dayal, A. Mehta, M. Marwah, and R. K. Sharma. Chaos: A data stream analysis architecture for enterprise applications. In *CEC'09*, 2009.

- [9] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman. On supporting Kleene closure over event streams. In *International Conference on Data Engineering (ICDE)*, pages 1391–1393, 2008.
- [10] W. Kim. On optimizing an sql-like nested query. *ACM Trans. Database Syst.*, 7:443–469, September 1982.
- [11] M. Liu, E. A. Rundensteiner, D. Dougherty, C. Gupta, S. Wang, I. Ari, and A. Mehta. High-performance nested CEP query processing over event streams. In *International Conference on Data Engineering (ICDE)*, April, 2011.
- [12] M. Liu, E. A. Rundensteiner, D. J. Dougherty, C. Gupta, S. Wang, I. Ari, and A. Mehta. NEEL: The nested complex event language for real-time event analytics. In *In Business Intelligence for the Real Time Enterprise (BIRTE)*, pages 116–132, 2010.
- [13] M. Liu, M. R. E. A. Rundensteiner, D. J. Dougherty, C. Gupta, S. Wang, I. Ari, and A. Mehta. Processing nested complex sequence pattern queries over event streams. In *Data Management for Sensor Networks (DMSN)*, pages 14–19, 2010.
- [14] Y. Mei and S. Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *Special Interest Group on Management of Data (SIGMOD) Conference*, pages 193–206, 2009.
- [15] P. C. Michael V. Mannino and C. T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys (CSUR)*, 20, 1988.
- [16] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *Special Interest Group on Management of Data (SIGMOD) Conference*, pages 247–258, 1990.
- [17] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *International Conference on Data Engineering (ICDE)*, pages 450–458, 1996.

- [18] J. M. Smith and P. Y.-T. Chang. Optimizing the performance of a relational algebra database interface. *Commun. ACM*, 18(10):568–579, 1975.
- [19] E. Wong and K. Youssefi. Decomposition - a strategy for query processing. *ACM Trans. Database Syst.*, 1(3):223–241, 1976.
- [20] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Special Interest Group on Management of Data (SIGMOD) Conference*, pages 407–418, 2006.