Optimized Nested Complex Event Processing Using Continuous Caching

by

Medhabi Ray

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 2011

APPROVED:

________________________________________
Professor Elke A. Rundensteiner, Thesis Advisor

________________________________________
Professor Robert E. Kinicki, Thesis Reader
Abstract

Complex Event Processing (CEP) has become increasingly important for tracking and monitoring anomalies and trends in event streams emitted from business processes such as supply chain management to online stores in e-commerce. These monitoring applications submit complex event queries to track sequences of events that match a given pattern. While the state-of-the-art CEP systems mostly focus on the execution of flat sequence queries, we instead support the execution of nested CEP queries specified by the (NEsted Event Language) \textit{NEEL}. However the iterative execution often results in the repeated recomputation of similar or even identical results for nested subexpressions as the window slides over the event stream. This work proposes to optimize \textit{NEEL} execution performance by caching intermediate results. In particular a method of applying selective caching of intermediate results called Continuous Sliding Caching technique has been designed. Then a further optimization of the previous technique which we call the Semantic Caching and the Continuous Semantic Caching have been proposed. Techniques for incrementally loading, purging and exploiting the cache content are described. Our experimental study using real-world stock trades evaluates the performance of our proposed caching strategies for different query types.
Acknowledgements

This work is supported by HP Labs Innovation Research Program and National Science Foundation under grants NSF 1018443 and NSF IIS 0917017, Turkish National Science Foundation TUBITAK under career award 109E194.

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. Her unlimited energy, dedication and enthusiasm was contagious. It was a great privilege to work with her on this thesis.

I have learned a great deal from my fellow graduate students and friends in DSRG and WPI. I am especially grateful to Mo Liu who introduced me to CEP research and helped me extend out on her work which turned into my thesis.

I would also like to thank all of my professors at WPI for inspiring me to question things, think critically, and challenging me. In particular, I would like to thank Professor Robert E. Kinicki, my Thesis Reader for his time and valuable suggestions.

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

1 Introduction 1
  1.1 Motivation ................................................................. 1
  1.2 State of Art ............................................................... 3
  1.3 Problem Definition ...................................................... 3
  1.4 Proposed Approach ..................................................... 4
  1.5 Goals Achieved ............................................................ 5

2 Nested CEP Query Processing 6
  2.1 Preliminaries ............................................................ 6
  2.2 NEEL: The Nested Complex Event Language ....................... 7
    2.2.1 Iterative Nested Execution Strategy .......................... 10
    2.2.2 Processing Nested Queries with Boolean components .......... 10
    2.2.3 Processing Nested Queries with Predicates .................. 12

3 Caching 14
  3.1 General Caching Architecture ...................................... 14
  3.2 Caching Intermediate Results ....................................... 15
  3.3 Continuous Sliding Cache .......................................... 18
    3.3.1 Continuous Cache Design ...................................... 18
    3.3.2 Cache Usage ..................................................... 19
    3.3.3 Cache Maintenance ............................................. 19
3.3.4 Continuous Sliding Caching for Boolean Components .......... 21
3.3.5 Continuous Sliding Caching with Predicate Correlation .......... 22

4 Optimizations on Continuous Caching 24
4.1 Semantic Caching based on Interval ............................... 24
  4.1.1 Semantic Cache Design ........................................... 25
  4.1.2 Handling Overlapping Cache-Intervals .......................... 25
  4.1.3 Semantic Cache Usage ............................................. 26
  4.1.4 Semantic Cache Maintenance ..................................... 27
4.2 Continuous Sliding Cache with Semantic ............................ 30
  4.2.1 Continuous Cache with Semantics Design ......................... 30
  4.2.2 Cache Loading ...................................................... 30
  4.2.3 Cache Usage ....................................................... 31

5 Cost Analysis for Event Expression Evaluation 32

6 Experimental Evaluation 36
  6.1 Experimental Setup ................................................... 36
  6.2 Data Description ..................................................... 37
  6.3 Rewriting Nested Queries .......................................... 37
  6.4 Analysis of Results ................................................ 38
    6.4.1 Effect of Window Sizes on the Caching techniques .......... 38
    6.4.2 Effect of Length of Subqueries on the Caching techniques .. 40
    6.4.3 Effect of Number of Subqueries on the Caching techniques .. 40
    6.4.4 Comparing Caching techniques using Synthetic Data .......... 42
    6.4.5 Comparing Caching techniques against Rewriting technique .. 43

7 Related Work 46
  7.1 Caching ............................................................. 46
7.2 Nested SQL and Decorrelation ................................. 47
7.3 Other CEP Systems ............................................. 47

8 Conclusions and Future Work ................................. 49
8.1 Conclusion ......................................................... 49
8.2 Future Work ...................................................... 49
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Example Query $Q_1$</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Algorithm to Compute Interval Constraints for an Inner Query $Q_i$ Given an Outer Partial Result $r_j$</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Example Query with Predicate Correlation $Q_3$</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Overall System Architecture</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Example Query $Q_4$</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>Recomputation of Intermediate Results</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>Cache Design</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>Continuous Cache Usage</td>
<td>18</td>
</tr>
<tr>
<td>3.6</td>
<td>Object Caching Example</td>
<td>20</td>
</tr>
<tr>
<td>3.7</td>
<td>Object Caching Updating</td>
<td>20</td>
</tr>
<tr>
<td>3.8</td>
<td>Continuous Sliding Caching Example with Predicate Correlation</td>
<td>21</td>
</tr>
<tr>
<td>3.9</td>
<td>Over-Computation in Object Caching</td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>Cache Design</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>Types of overlap between Cache and Query-Intervals</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Cache Usage</td>
<td>28</td>
</tr>
<tr>
<td>4.4</td>
<td>Semantic Caching for a Generating Sub-query</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>Semantic Caching for boolean Sub-Queries</td>
<td>29</td>
</tr>
<tr>
<td>4.6</td>
<td>Continuous Semantic Cache Usage</td>
<td>31</td>
</tr>
</tbody>
</table>
6.1 Sample Data ................................................................. 37
6.2 Varying Window Size .................................................... 38
6.3 Comparing execution time of queries using Semantic Caching, Continuous Sliding
    Caching, Continuous Semantic Caching and iterative technique with Window Size
    100 ............................................................................. 38
6.4 Comparing execution time of queries using Semantic Caching, Continuous Sliding
    Caching, Continuous Semantic Caching and iterative technique with Window Size
    500 ............................................................................. 39
6.5 Comparing execution time of queries using Semantic Caching, Continuous Sliding
    Caching, Continuous Semantic Caching and iterative technique with Window Size
    1000 ............................................................................. 39
6.6 Varying Length of Children Queries .................................... 40
6.7 Comparing execution time with Semantic Caching, Continuous Sliding Caching
    and Continuous Semantic Caching with with varying length of sub-queries .......... 41
6.8 Varying Number of Positive Children Queries ....................... 41
6.9 Comparing execution time for Semantic Caching, Continuous Sliding Caching and
    Continuous Semantic Caching with varying number of sub-queries ..................... 42
6.10 Comparing Caching techniques using Synthetic Data .................. 43
6.11 Varying Length of Boolean Sub Queries ............................... 43
6.12 Comparing execution time for Rewriting technique, Continuous Semantic Caching
    and iterative technique with varying number of sub-queries ................................. 44
List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>NEEL language definition</td>
<td>8</td>
</tr>
<tr>
<td>5.1</td>
<td>Terminology Used in Cost Estimation</td>
<td>32</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Modern applications ranging from scientific and financial data analytic systems to tracking and monitoring in supply chain management systems depend heavily on mining complex patterns over streams of events. Complex Event Processing is the broad umbrella under which such pattern mining techniques over real time event streams are classified. To be fully usable in a variety of applications, 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 [4, 18, 22]. 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 warnings such as “The tool with id = “5” must be disposed”. Query $Q_1$ (Figure 1.1) 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, s.id=d.id=c.id=o.id), 
| WITHIN  | r.id=w.id=o.id and o.ins-type="surgery") |
|         | Operating o, r.id=w.id=o.id and o.ins-type="surgery") | 1 hour |

Figure 1.1: Example Query $Q_1$

One important feature of any query language, as learned from the community’s experience with SQL, is the flexible nesting of query expressions. Nested CEP queries provide users who query the streaming data, with an intuitive way of expressing their requirements. Without this capability, users are severely restricted in forming complex patterns in a convenient and succinct manner [17]. In fact many nested queries cannot be expressed as flat queries or might result in an exponential number of flat queries in order to account for all combinations of events. However, the state-of-art CEP systems including SASE [23], ZStream [16] and Cayuga [2] currently do not support such a flexible combination of nested CEP queries with negation.

Assuming there exists a processing technique for non-nested CEP queries, [14] adopts a top-down iterative process of executing nested CEP queries, where by the outermost query is processed first followed by the processing of its direct children queries and so on. First all component events matching the outer query are identified. In the 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 which in this case are the (Sharpening, Disinfection, Checking) subsequences. As the 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. It misses critical opportunities for optimization as we will illustrate below.
1.2 State of Art

The state-of-art CEP systems (such as SASE [23] and ZStream [16]) do not support nested CEP queries. Cayuga [2] only allows sub-queries in the FROM clause (of standard SQL [9]) and 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 technique and optimization mechanisms for nested CEP queries have not been proposed in the state-of-the-art solutions.

In a recent work by Liu et al [13], a nested CEP language $NEEL$ was proposed, that supports nesting of Sequence, AND, OR and Negation queries, along with an iterative nested execution strategy for processing nested event queries expressed in $NEEL$ [14]. Such iterative nested execution while correct is extremely inefficient. Based on the definition of the query Window in definition 6 in chapter 6, section 3.2, as the query window slides continuously over the event stream the query window overlaps. Results valid for a certain Window often remain valid over the next Window slide. Full results satisfying the nested subexpression, such as instances that match the subsequence $SEQ(Sharpening \ s, \ Disinfection \ d, \ Checking \ c)$ in $Q_1$ will be repeatedly constructed and processed. However, in real time monitoring systems relying on CEP queries, the performance with reference to time and memory is very critical. Optimization strategies such as caching are known to be effective for SQL query processing [7].

The goal of this work is to design an efficient caching mechanism as an optimization strategy for processing nested CEP queries.

1.3 Problem Definition

The problem is challenging as the cache content is under continuous flux, i.e. due to the streaming nature of the data, events are continuously expiring and new events are arriving. There is a need to study what to cache and how to keep the cache up to date as the window slides.
**Problem 1:** In the streaming context one needs to design an effective algorithm for continuously caching results and maintaining the cache as the window keeps sliding.

**Problem 2:** There is also a need for an efficient data structure for accessing, maintaining and retrieving results from the cache.

**Problem 3:** A decision problem arises when there are memory limitations about selective caching. There is a need to figure out a strategy to make this decision.

### 1.4 Proposed Approach

This research work studies the optimization of nested CEP queries using caching of intermediate results. This work proposes a strategy for continuously caching results that have been produced and then further goes on to optimize this strategy. The contributions of this thesis include:

- Designing a general approach for buffering the intermediate results in a continuous sliding cache.

- Proposing the idea of semantic caching based on installing and matching semantic descriptors as an optimization over the basic continuous sliding caching technique.

- Optimizing caching techniques for efficiently handling special types of sub-queries such as boolean sub-queries which return true or false instead of event patterns.

- Describing a cost model analysis of the iterative processing technique with caching intermediate results.

- Implementing this continuous caching solution along with its various optimizations in the E-Cube CEP engine [12]. We conduct experimental studies and compare the nested CEP query execution with and without caching and other state of art Nested CEP processing technique such as rewriting Nested CEP queries. We experimentally evaluate our proposed execution strategy on simulated real time data streams.
1.5 Goals Achieved

These techniques achieve the following goals and has a number of advantages over other existing techniques of processing Nested CEP queries.

- The method is fairly simple to apply with minimum computation overhead. State-of-the-art techniques including Rewriting nested queries [11] are often cumbersome and need complex Rewriting rules and systematic application of such rules.

- Our approach can be extended to handle any kind of nested queries including predicates. Rewriting Nested queries [11] is often not possible under various situation including when a boolean sub-query is nested within another boolean sub-query. In some cases, the presence of predicate correlations limit the applicability of Rewriting rules.

- In spite of being straightforward, our technique performs better in terms of execution time against iteratively processing Nested queries and performs at least as good as the Rewriting technique does.
Chapter 2

Nested CEP Query Processing

This chapter gives a brief background of the concepts on which the work of this thesis is built. To query a stream database for pattern queries we need a query language and a processing technique for processing such queries. This chapter will introduce NEested Event Language NEEL [13] and also describe a default processing strategy for processing queries. The thesis aims to improve upon the default strategy by applying caching.

2.1 Preliminaries

An event instance is an occurrence of interest in a stream which can be either primitive or composite as further introduced below. A primitive event instance denoted by a lower-case letter (e.g., ‘e’) is the smallest, atomic occurrence of interest in a system. $e_i.ts$ and $e_i.te$ denote the start and the end timestamp of an event instance $e$, respectively, with $e_i.ts \leq e_i.te$. For a primitive event instance $e_i$, $e_i.ts = e_i.te$. For simplicity, we use the subscript $i$ attached to a primitive instance $e$ to denote the timestamp $i$. A composite event instance is composed of constituent primitive event instances $e = \langle e_1, e_2, ..., e_n \rangle$. A composite event instance $e$ occurs over an interval. The start and end timestamps of $e$ are equal to $e.ts = \min\{e_i.ts | \forall e_i \in e\}$ and $e.te = \max\{e_i.te | \forall e_i \in e\}$, respectively.

An event type is denoted by a capital letter, say $E_i$. An event type $E_i$ describes a set of attributes that
the event instances of this type share. An event type can be either a primitive or a composite event type [6]. *Primitive event types* are pre-defined in the application domain of interest. *Composite event types* are aggregated event types created by combining other primitive and/or composite event types to form an application specific type. \( e_i \in E_j \) denotes that \( e_i \) is an instance of the event type \( E_j \). We use \( e_i\.type \) to denote the type \( E_j \) of \( e_i \). Suppose one of the attributes of type \( E_j \) is \( \text{attr}_j \) and \( e_i \in E_j \), we use \( e_i\.\text{attr}_j \) to denote \( e_i \)'s value for that attribute \( \text{attr}_j \).

### 2.2 NEEL: The Nested Complex Event Language

We now briefly introduce the **NEstED Event Language** *NEEL* [13] for specifying nested complex event pattern queries. *NEEL* is an extension of non-nested CEP languages from the literature [3, 23, 2]. *NEEL* supports the nesting of AND, OR, Negation and SEQ operators at any level. Figure 2.1 gives the formal definition of *NEEL* language. The semantics of these primitive operators have been described below. \( Q_1 \) in Figure 1.1 is a sample query expressed by *NEEL*. For a more detailed discussion as well as several case studies of the *NEEL* language, the reader is referred to [13].

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. In our language, the time period is expressed as a sliding window, though other window semantics could also be applied. “A set of histories” is returned as a result with each history equal to one “set of instance matches”.

A time-based sliding window is a moving window extending back to the past from time specified by the arrival of the last event instance. Time-based sliding windows enable us to limit the number of events considered by a query. Based on the timestamp of the current incoming event, older events which fall outside the window are purged out and not considered for complex event pattern formation.

**Operators in the PATTERN clause.** SEQ in the PATTERN clause specifies the particular tem-
poral order in which the event instances of interest should occur. The components of the sequence are the stipulated occurrences and non-occurrences of events of certain event types [10].

**Definition 1** [SEQ operator]. \(\text{SEQ}(E_1 e_1, ..., E_i e_i, ..., E_n e_n)\) specifies a temporal order in which the event instances of interest \(e_1, ..., e_i, ..., e_n\) must occur. The output is a composite event \(e\) composed of \(e_1\) to \(e_n\) such that \(e_1.ts < ... < e_i.ts < ... < e_n.ts, and e_n.ts - e_1.ts \leq \text{window}\) with the window specified in the WITHIN clause.

**Example 1** Given \(\text{SEQ(Recycle r, Washing w)}\) and the partial input stream \(r_1, w_2, w_3\) all falling within the window. Then \(\text{SEQ(Recycle r, Washing w)}\) generates 2 results \(\{r_1, w_2\}\) and \(\{r_1, w_3\}\).

**Definition 2** [OR operator]. OR operator specifies disjunction of occurrences of events. \(\text{OR}(E_1 e_1, ..., E_i e_i, ..., E_n e_n)\) means one or more event instances of types \(E_1, ..., E_i, ..., E_n\) occur within a specified time window.
Definition 3 [AND operator]. \( \text{AND}(E_1 e_1, \ldots, E_i e_i, \ldots, E_n e_n) \) means event instances of types \( E_1, \ldots, E_i, \ldots, E_n \) occur within a specified time window, and their order does not matter. \( \text{AND} \) operator computes the cross product of input events of the specified types.

Example 2 Given \( \text{AND}(\text{Recycle} \ r, \text{Washing} \ w) \) and the partial input stream \( w_1, \ r_2, \ w_3 \) within the window. Then the two results \{r_2, \ w_1\} and \{r_2, \ w_3\} are generated.

Definition 4 [Negation]. The symbol “!” before an event expression \( E_i \) expresses the negation of \( E_i \) and indicates that \( E_i \) is not allowed to appear in the specified position [23].

Definition 5 [Exists]. The symbol “\( \exists \)” before an event expression \( E_i \) expresses the existence of \( E_i \) and indicates that at least one \( E_i \) must appear in the specified position [23].

Any component of \( \text{SEQ} \) including at the start or the end can be negated using “!””. \( \text{SEQ}(E_1 e_1, \ ! \ E_2 e_2, \ E_3 e_3) \) indicates that \( e_3 \) follows \( e_1 \) within a specified window without any interleaving instances of \( e_2 \) between \( e_1 \) and \( e_3 \). \( \text{AND}(E_1 \ e_1, \ ! \ E_2 \ e_2, \ E_3 \ e_3) \) indicates that both \( e_1 \) and \( e_3 \) occur while no \( e_2 \) occur anywhere within that same window. If there is a ! (Negation) symbol before an event expression, we now say that the event expression marked by ! is a negative event expression. Otherwise it is a positive event expression. At least one positive event expression must exist in \( \text{SEQ} \) and \( \text{AND} \) operators.

Example 3 Given \( \text{AND}(\text{Recycle} \ r, \text{Washing} \ w, \ ! \text{Checking} \ c) \) and the partial input stream \( c_1, \ w_2 \) and \( r_3 \), no results are generated due to the existence of the Checking event \( c \) within the window.

Nested expression and variable scope. If \( E_1, \ E_2, \ldots, \ E_n \) are event expressions, an application of \( \text{SEQ}, \ \text{AND} \) and \( \text{OR} \) over these event expressions is again an event expression [6]. An event expression \( \text{exp}_i \) can be used as an inner component to construct an outer expression \( \text{exp}_j \). The event instances in an outer expression are visible within the outer expression as well as within the scope of its own nested inner expressions. \( Q_1 \) in Figure 1.1 is an example of a nested expression. The outer expression is \( \text{SEQ}(\text{Recycle} \ r, \text{Washing} \ w, \text{Operating} \ o) \) and the inner expression
is SEQ(Sharpening, Disinfection, Checking). The variables $r$, $w$ and $o$ in the outer expression are visible in the inner expression.

**Predicate specification.** The optional qualification $[<\text{qual}>]$ in the PATTERN clause contains one or more predicates. Predicates only referring to events in expression $exp_i$ are specified directly inside $exp_i$ (simple predicates). Predicates referring to both event instances from the outer and the inner expressions are correlated predicates. They must be placed with the innermost expression where a variable used in the expression is declared.

### 2.2.1 Iterative Nested Execution Strategy

Following the principle of nested query execution for SQL queries [20], the outer query is evaluated first followed by its inner sub-queries. The results of the inner queries are passed up and joined with the results of the outer query. The main idea of our nested execution is about passing down more stringent window constraints from outer queries to inner queries. For every outer partial query result, a constraint window (see Figure 2.1) is passed down for processing each of its children sub-queries. These sub-queries compute results involving events within the substream constrained by the constraint 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 positive sub-queries. The outer sequence result is filtered if the result set of any of its negative sub-queries is not empty. We apply iterative execution until a final result sequence is produced by the root operator. Finally, the process repeats when the outer query consumes the next instance $e$. We will discuss nested queries with negation and predicates in more detail in Sections 2.2.2 and 2.2.3, respectively.

### 2.2.2 Processing Nested Queries with Boolean components

We now describe how to support boolean components which could be positive - “∃” or negative - “!” in nested queries. If a query has a $\exists A$ between positive B and C event types, we first evaluate the query without the negation, i.e., they compute all $<b, c>$ results. Then for every
Figure 2.1: Algorithm to Compute Interval Constraints for an Inner Query \( Q_i \) Given an Outer Partial Result \( r_j \)

result generated we check if an A event occurred between the qualified B and C events. If it occurs, such pairs are valid results. While if the query has a !A between positive B and C event types we reject such <b, c> results which have an A type event between them. When two negative event types are adjacent to each other, their order does not matter. For example, SEQ(A, !B, !C, D) is equivalent to SEQ(A, !C, !B, D). That is, all <a, d> results without any B and C events in between them would be returned.

In the nested query model NEEL [13], a sub-query as a whole could also be negated. For instance, consider the query SEQ(A, !AND(B, C), D). For each outer result of SEQ(A, D), we search for AND(B, C) results occurring between such A and D events. If none exist, then the outer SEQ(A, D) result is returned, otherwise it is filtered out. Similarly there could be a sub-query

\[
\text{Interval IntervalConstraints} \quad \text{(Result } r_j, \text{ Query } q_i) \\
// r_j is one partial result of the outer query \\
01 \text{ Interval } ts; \\
02 \text{ if(root operator of } q_i \text{ is SEQ)} \\
\quad \text{// gets the position of } q_i \text{ in outer query } \\
03 \{ \text{ nestedPosition = getNestedPos(} q_i); \} \\
\quad \text{// if outer query starts with sub query } q_i \\
04 \text{ if(nestedPosition == 0)} \\
\quad \text{// left bound is time of last event in result } r \\
05 \quad \text{ts.left = getTime(} r_j.\text{LastEve) - W;} \\
\quad \text{// if outer query ends with sub query } q_i \\
06 \text{ if(nestedPosition == } r_j.\text{size)} \\
\quad \text{// right bound is time of first event in result } r \\
07 \quad \text{ts.right = getTime(} r_j.\text{FirstEve) + W;} \\
08 \text{ else } \\
09 \quad \{ \text{ts.left = getTime(} r_j.\text{get(nestedPos-1))} \\
10 \quad \text{ts.right = getTime(} r_j.\text{get(nestedPos)}\}) \} \\
11 \text{ if(root operator of } q_i \text{ is AND)} \\
12 \quad \{ \text{ts.left = getTime(} r_j.\text{lastEve) - W;} \\
13 \quad \text{ts.right = getTime(} r_j.\text{lastEve}); \} \\
14 \text{ if(root operator of } q_i \text{ is OR)} \\
15 \quad \{ \text{ts.left = getTime(} r_j.\text{lastEve) - W;} \\
16 \quad \text{ts.right = getTime(} r_j.\text{lastEve}); \} \\
17 \text{return } ts;
which is a positive boolean. For example, SEQ(A, \( \exists \) AND(B, C), D). For each outer result of SEQ(A, D), we search for AND(B, C) results occurring between such A and D events. If none exist, then the outer SEQ(A, D) result are filtered out, otherwise it is returned.

```
SEQ(Recycle r, Washing w, 
   AND(Sharpening s, Disinfection d 
      WHERE s.id = r.id),
   Operating o 
   WHERE r.type = o.type)
WITHIN 10 minutes
```

Figure 2.2: Example Query with Predicate Correlation \( Q_3 \)

### 2.2.3 Processing Nested Queries with Predicates

The approach of handling sub-queries with correlated predicates is similar to the nested execution described above except that the join is not only based on timestamps but also on other predicates. Different cases for predicate handling include:

- **Local predicates within one pattern operator.** Events are filtered based on predicate values before being stored in their stack. Query processing proceeds otherwise as explained above. For example, for the query \( Q_3 \) in Figure 2.2, Operating events where the instrument type is not equal to the instrument type for recycle events will be filtered.

- **Correlated predicates between inner and outer queries.** Nested sub-queries may be correlated with their parent queries by means of predicates. In order to evaluate these queries with predicates, it is necessary to pass down attribute values to the children queries. For example, the query in Figure 2.2 requires events in the inner sub-queries have the same tool id as the outer match. For each outer SEQ(Recycle r, Wash w, Operating o) match, the tool id information for the recycle instance is thus passed down to the children sub-queries. Inner query results involving Sharpening events having the same tool id with the outer match are returned to the upper query.
Discussion of Limitation. The above described evaluation methodology while correct, suffers from several inefficiencies.

- Candidate results of outer expressions \( \text{SEQ(Recycle } r, \text{ Washing } w, \text{ Operating } o) \) initially generated may later need to be discarded.

- Full results for boolean sub-expressions such as negative component \( \text{SEQ(Sharpening } s, \text{ Disinfection } d, \text{ Checking } c) \) are constructed.

- The same inner sub-expressions are repeatedly executed for different outer sub-expression results.

The iterative execution method does not solve these problems [14]. To overcome such inefficiencies, we explore caching intermediate results in the following sections.
Chapter 3

Caching

3.1 General Caching Architecture

Figure 3.1: Overall System Architecture

Figure 3.1 shows the overall system architecture of the caching system for processing *NEEL* queries. The single-lined arrows show the direction of control flow while broad arrows show the flow of data. The Compute engine conducts the actual query processing by controlling the
functionality of query operators, such as AND or SEQ operators which produce pattern query results. The stream is input into this module and Complex Event patterns are output from this module. The Cache Validity Checker module checks if the results required at a given time are present in the cache or not. It may decide to use the cache directly if it contains the required results. Otherwise it would invoke the Cache Maintenance module to update the cache. The Cache Maintenance module maintains what the cache should contain at any given time. It updates the cache by calling the Executor module. It also purges the cached results as the window slides over.

### 3.2 Caching Intermediate Results

**Definition 6** A time-based sliding window is a moving window extending back to the past from time specified by the arrival of the last event instance.

Time-based sliding windows enable us to limit the number of events considered by a query. The query Window is specified by the Query, thus all events considered for query evaluation fall exclusively inside [current time - Window, current time]. The Window in our case moves incrementally and hence a fraction of the events valid in one Sliding Window are also valid for the next Slide. This is a basic assumption we make which makes caching an appropriate optimization technique.

**Definition 7** For a given nested query, the Query-Interval for a sub-expression is defined by the timestamps of the events in the results of its outer expression. If the outer sub-expression is a SEQ, the Query-Interval is given by the timestamps of primitive positive event types adjacent to the sub-query. If the outer sub-expression is an AND, the Query-Interval is given by the timestamps of the first and last primitive positive event types of the query.

**Definition 8** For the nested query \( \text{SEQ}(E_1 e_1,...,E_{i-1} e_{i-1}, \text{SEQ}(E_i e_i,...,E_j e_j), \text{SEQ}(E_{j+1} e_{j+1})...
\)... \(E_k e_k)) \), \( E_{k+1} e_{k+1}, E_{k+2} e_{k+2}...E_n e_n \)) the Query-Interval for the sub-query \( \text{SEQ}(E_i e_i,...,E_j e_j) \), is given by \([e_{i-1}.ts, e_{k+2}.ts]\) where \(E_{i-1}\) and \(E_{j+1}\) are primitive positive event types adjacent to the sub-query.
The re-computation of the results for inner sub-queries every time an outer triggering event arrives can be rather expensive. We propose to exploit the fact that CEP queries work on sliding window over the input stream. It is easy to see that many intermediate results would continue to be valid from one sliding window to the next because they may fall within the same sub-windows within different but likely still overlapping global query windows. Here we will define the concept of Query-Interval for sub-queries.

$$\text{SEQ(Recycle } r, \text{ AND(Sharpening } s, \text{ Disinfection } d), \text{ Washing } w, \text{ Operating } o) \text{ WITHIN } 10 \text{ minutes}$$

Figure 3.2: Example Query $Q_4$

$$\text{SEQ(Recycle } r, \text{ AND(Sharpening } s, \text{ Disinfecting } d), \text{ Washing } w, \text{ Operating } o) \text{ WITHIN } 10 \text{ minutes}$$

Figure 3.3: Recomputation of Intermediate Results
Example 4 Consider the query $Q_4$ shown in Figure 3.2. It is the query plan of a 2-layered nested NEEL query, with an outer SEQ pattern and an inner AND pattern. Semantically this query means that it looking for Recycle, Washing and Operating events in that sequence with any combination of Sharpening and Disinfecting events between the Washing and Operating events. Based on SASE [23] implementation of pattern queries, there are stacks for each event type mentioned in the query. As events stream in, the relevant events are put in their respective stacks. Construction of patterns are triggered by the arrival of triggering events. For pattern SEQ, the triggering event is given by the rightmost positive primitive event type. Hence in the example query, the triggering event is the Operating type events. The events of the stream seen so far have been added into the stack. When the triggering event $o_{18}$ arrives, it forms the outer query result $<r_1, w_{12}, o_{18}>$. The sub-window for the inner query is being computed which is $[1, 12]$ in this case, hence the results $<s_3, d_{10}>$ and $<s_{11}, d_{10}>$ are computed. Then when the event $o_{20}$ arrives and the result $<r_1, w_{12}, o_{20}>$ is formed, the sub-window for the inner query again corresponds to the interval $[1, 12]$. The results during this interval would be unnecessarily recomputed again. Although the Sliding Window has moved over, the sub-window for the inner sub-query has not changed. Thus there is a waste of computational resources considering all the events are arriving in order. Instead we propose that previously calculated results of the previous window could be cached and reused in the new window.

Based on these observations we now propose to apply caching to these intermediate results. Caching is a well established optimization technique designed to speed up query processing keeping track of previously computed function results in databases, or by avoiding expensive accesses to disk and other slow devices. However although the idea of caching in the stream context where the data we are processing are continuously changing does not seem intuitive, we can see from the motivating example above how caching could provide tremendous benefits in reducing repetitive re-computation of intermediate results due to overlapping windows.

We now propose to cache and incrementally maintain the inner query results to serve outer queries from one window to the next.
3.3 Continuous Sliding Cache

![Figure 3.4: Cache Design](image)

3.3.1 Continuous Cache Design

We propose to maintain a cache for each sub-query. In general a cache corresponds to a list of result tuples conforming to the intermediate output schema. We associate an indicator to the cache which indicate the timestamp till which the cache contains all inner sub-query results. The indicator we attach to the cache in this methodology will be called the “rightBound”. For the query shown in Figure 3.4 it is given by $e_{j+1} \cdot ts$ such that $e_{j+1}$ has the maximum timestamp among all events of type $E_{j+1}$ which have arrived so far and for which the cache has been computed as shown in Figure 3.4. In the continuous caching solution the cache will be loaded with all possible results so far in the input stream up to the rightBound.

```java
CacheUsage(Query qi, Interval queryInterval) ()
01 computeNeeded = true;
02 if(queryInterval.rightBound < Cache.rightBound)
03   computeNeeded = false;
04 if(computeNeeded == true)
05   PartialCompute(temp, queryInterval);
06 else Reuse();
```

![Figure 3.5: Continuous Cache Usage](image)
### 3.3.2 Cache Usage

Given an outer query result triggered by an event $e_n$, we calculate the constraint window for each sub-query which we call Query-Interval. The procedure in figure 3.5 summarizes the process of cache usage given a Query-Interval. We will check the “rightBound” of the cache. If the rightBound of the cache is greater than or equal to the $Query$-Interval.rightBound this means that the existing cache contains all the required results. Hence a scan through the cache will give us the required results. Clearly not all results in the cache may be utilized by the current sub-query and they are thus filtered during the scan. If however the $Query$-Interval.rightBound is greater than the timestamp attached to the cache we have to instead first update the cache as explained below before we can extract the desired result.

### 3.3.3 Cache Maintenance

When the $Query$-Interval.rightBound is greater than the rightBound of the cache, we expand the cache content. That is the case when a cache is not sufficient (misses results). For all new “trig-gering” events $e_j$ within the sub-query $SEQ(E_i \ldots E_j)$ in Figure 3.4 that have not been previously loaded into the cache namely $e_j.ts > Cache.rightBound$ and $e_j.ts <= Query$-Interval.rightBound, compute all sub-query results and load them into the cache. Then the rightBound of the cache is updated to reflect the present state of the cache namely $Cache.rightBound := Query$-Interval.rightBound. When an outer query triggering event $e_n$ arrives, events with timestamp less than $e_n.ts - window$ are purged from their stacks. Similarly, caching results involving events with timestamp less than $e_n.ts - window$ are also deleted from the cache. This window is the overall query window specified in the query.

**Example 5** Figure 3.6 shows how on the arrival of event $o_{20}$, Sequence Construction by joining the events of the corresponding stacks is triggered and the interval is extracted for the result $<r_1, w_{12}, o_{18}>$ namely [1, 12]. AND(Sharpening, Disinfecting) results are constructed for interval [1,12] and stored in the cache and the Cache.rightBound is set to 12. When the new triggering
event $o_{20}$ arrives, we obtain two results for the outer query, namely \(< r_1, w_{12}, o_{18}>\) and \(< r_1, w_{19}, o_{20}>\). For the first outer result, the Query-Interval for the sub-query is again \([1,12]\). Hence the existing cache will have all the results. However for the next outer result, the Query-Interval is \([1,19]\) as shown in Figure 3.7. Since the Query-Interval.rightBound > Cache.rightBound, the cache must be updated. Thereafter the Cache.rightBound is updated to 19 to reflect its present
3.3.4 Continuous Sliding Caching for Boolean Components

When the inner sub-query is a boolean expression (e.g. if a negation symbol exists in front of the subexpression), then the only difference would be to search for results during the Query-Interval and return True or False based on whether any results are found or not during the Query-Interval.

Example 6 Consider again the query in the example in Figure 3.6 with a slight change. Let the sub-query now be negated i.e \( \text{SEQ}(\text{Recycle } r, \neg \text{AND}(\text{Sharpening, Disinfecting}), \text{Washing } w, \text{Operating } o) \). In this case we could still reuse the cache. For example for the outer query result \(<r_1, w_{12}, o_{18}>\) the Query-Interval is \([1,12]\). The rightBound at this time is 12. We then search for results during the required interval within the cache. If the cache is empty for the required interval, we will output the outer query result and vice versa.

![Figure 3.8: Continuous Sliding Caching Example with Predicate Correlation](image-url)
3.3.5 Continuous Sliding Caching with Predicate Correlation

Our caching technique is extended to support predicate correlations. Predicate correlations could further be classified into predicates between generating subqueries or boolean sub-queries. The processing is very similar to ones explained above except that we will further partition the cache by the different values of the predicate seen so far and maintaining a rightBound for each partition. Thus for the query $Q_3$ in Figure 2.2, we will have multiple partitions of the cache namely one for each different value of “id” pushed down from the outer Recycle event type.

Example 7 For example for query $Q_3$ in Figure 2.2, for the outer query result $<r_1, w_{12}, o_{18}>$ where $r_1.id = 10$, the Query-Interval $[1,12]$ along with the value of $r_1.id$ is passed down. The corresponding results $<s_3, d_{10}>$ where $s_3.id = 10$ is stored in a cache partition with predicate value 10. For the outer query result $<r_2, w_{12}, o_{18}>$ where $r_2.id = 25$ we search for cache partitions with “id” = 25. It does not exist and is hence computed and stored in a partition annotated with predicate value = 25. Similarly cache updates will be done to respective partitions.

![Figure 3.9: Over-Computation in Object Caching](image)
Discussion of Continuous Sliding Cache

The Continuous Sliding Caching is an improvement over the iterative processing technique.

- It avoids the re-computation of intermediate results.
- Thus it leads to much faster response time of a given nested query.

However our analysis reveals that it still suffers from a few disadvantages namely:

- The continuous sliding cache can grow very large. Then there would be many results to scan to look for the relevant set of results and in fact at the end there may be no match.
- This technique might force us to pre-compute many results that are never used by any outer sub-query. Figure 3.9 shows the numerous results produced between the interval of [20,26] which are never used. For multi-level nested queries these results might further result in producing results for lower sub-queries which will finally never be used by an outer result either.
- For boolean components this technique will end up storing the results in spite that it may not always be necessary to do so because boolean sub-expressions are not output but rather act as filters.
Chapter 4

Optimizations on Continuous Caching

4.1 Semantic Caching based on Interval

To overcome the disadvantages of the continuous caching technique, we now propose to enhance the capacity of the cache with a semantic descriptor that effectively indexes the cache content. Figure 4.1 shows an example of how the caches are related to a given query for Semantic Caching.

Here we now propose to partition the cache into several intervals. Thus given a Query-Interval getting the results for that interval would be much faster using an interval to interval match. It will not have to scan a single repository to find the relevant results, instead it will bound the search space to a smaller region.
4.1.1 Semantic Cache Design

**Definition 9** Cache Interval: Pair of Time Stamps attached to cache during which some or all results of a sub-query is present in the cache valid in the Window

The concept of semantic descriptors in the form of Cache-Interval will be used in the following section. Cache-Interval denotes the time interval for which a cache is guaranteed to contain all possible results for. For a given sub-query we will maintain a list of Cache-Intervals and the results associated with the respective interval.

- The main challenge in this technique is to design a data structure which will support efficient look up.
- However we would avoid duplicate storage of the same results over overlapping intervals

![Diagram of Types of overlap between Cache and Query-Intervals](image)

**Figure 4.2: Types of overlap between Cache and Query-Intervals**

4.1.2 Handling Overlapping Cache-Intervals

In several occasions multiple Cache-Intervals may overlap. This would mean if the full set of results is stored for each overlapping interval, we would end up storing multiple copies of a single result tuple. This could be avoided by storing a result tuple only once, namely, the first time it is created. Thereafter if the same result occurs in another overlapping interval we will not
store it in the overlapping interval. We have two options namely, we could either refer to that interval or a reference to that tuple or being even more space efficient we may simply not store it. While searching for the complete set of results for a given Query-Interval we would then search all Cache-Intervals which overlap with the given Query-Interval. Thus if there are Cache-Intervals [2,7],[2,10],[5,10] and if the Query-Interval is [6,10], we would scan all three Cache-Intervals. There is also added overhead in the cache loading in this case. In case of storing results for overlapping intervals we need to first check if some previously computed results have already been stored in an overlapping cache interval.

**Example 8** Figure 3.7 shows how upon the arrival of event $o_{20}$, the interval extracted for the result $< r_1, w_{12}, o_{18} >$ is [1, 12]. When the new triggering event $o_{20}$ arrives, we obtain the Query-Intervals [1,19]. In this case the interval [1,12] already exists and we would have all results during this interval. Thereafter when [1,19] arrives, we will only store the results which are not in [1,12]. Hence for searching the results, we would scan both caches. This method clearly avoids duplicate storage of the same results for overlapping intervals.

### 4.1.3 Semantic Cache Usage

In this Semantic Caching approach instead of having to scan the raw object cache for matches, we can now match the meta descriptor - the Cache-Intervals, to facilitate efficient access. Once the matching Cache-Intervals have been identified the required results can be efficiently returned because this scheme can avoid scanning through a long list of results which do not belong to the current interval. The extracted Query-Interval is compared to the meta descriptor indexing the cache, namely the list of Cache-Intervals. Several types of overlaps as shown in Figure 4.2 can arise. Based on the type of overlap we will use the cache differently. We will consider them in the following order:

- **Exact Overlap or Contained.** The first type of overlap is shown in Figure 4.2(a). The Query-Interval is completely contained in at least one of the caches in the list of Cache-Intervals. This happens when the leftbound of the Query-Interval is greater or equal to the
Cache-Interval’s leftbound and the rightbound of the Query-Interval is less than or equal to the Cache-Interval’s rightbound. In this case all the results are guaranteed to be present in the list of caches which overlap this Query-Interval. Hence we will scan all the corresponding caches which overlap with this Query-Interval. Some results in the cache might not be used by the present Query-Interval which are filtered out.

- **Partial Overlap.** If the Query-Interval does not exactly match with any of the existing Cache-Intervals, then we would look for a partial overlap. In this case the leftbound of the Query-Interval is greater or equal to the Cache-Interval’s leftbound and the rightbound of the Query-Interval is greater than the Cache-Interval’s rightbound as shown in Figure 4.2 (b). In this case the inner sub-query is computed only for triggering events that have occurred between the Query-Interval.rightbound and Cache-Interval.rightbound similar to the partial compute in the Continuous Sliding Cache. While storing results we would make sure that these newly computed results are not present in any of the overlapping Cache Intervals. If such an overlap does not exist, we would have to recompute from scratch.

### 4.1.4 Semantic Cache Maintenance

When the Query-Interval is not overlapped by any existing Cache-Interval, the cache needs to be updated. We will have to partially or completely recompute the sub-query depending on the portion of the non-overlapping Query-Interval. If a Query-Interval has no overlap with an existing Cache-Interval, the sub-query needs to be computed from scratch. If a part of the Query-Interval overlaps a Cache-Interval as in the case in Figure 4.2(b) we will compute the sub-query for triggering events that occur only in the non-overlapping part.

**Example 9** Figure 4.4 shows the state of the cache at a given time for the given query. When the tuple $o_{30}$ arrives, the query is triggered and the outer result $\langle r_{20}, w_{26}, o_{30} \rangle$ is formed. The Query-Interval extracted is $[20, 26]$. The Interval is matched against the list of Intervals in the Semantic Cache. It is found and the results in that interval are passed up.
CacheUsage(Query q, Interval queryInterval) ()
01 Interval temp = null;
02 computeNeeded = true;
03 For(Interval cacheInterval: cacheIntervalsList)
04 if (queryInterval.isContained(cacheInterval))
05 Reuse(); computeNeeded = false; break;
06 if(queryInterval.overlaps(cacheInterval))
07 if (temp.overlapsLessThan (cacheInterval))
08 temp = cacheInterval;
09 End Loop
10 if(computeNeeded == true)
11 if(temp != null)
12 PartialCompute(temp, queryInterval);
13 else Compute(queryInterval);

Figure 4.3: Cache Usage

SEQ(Recycle r, !AND(Sharpening s, Disinfecting d, Checking c), Washing w, Operating o) partial outer query result
\langle r_{20}, w_{26}, o_{20} \rangle

Figure 4.4: Semantic Caching for a Generating Sub-query

Optimization for Boolean Sub-queries

Semantic Caching is particularly beneficial for negative sub-queries not only in terms of CPU processing costs but also in terms of memory consumption. Negative sub queries need not be joined with the positive outer query results of the query. Rather they act as filters screening some
of the intermediate results of the outer query. Hence we now propose to not store any actual tuples in the caches for the boolean sub-queries. Instead simply storing an “isEmpty” flag for a given interval is sufficient. Thus we check the isEmpty flag for a given Query-Interval and filter out the results if the isEmpty flag is false.

\[
\text{SEQ}(\text{Recycle } r, \\
\text{! AND}(\text{Sharpening } s, \text{Disinfecting } d), \\
\text{Washing } w, \text{Operating } o)
\]

\[
\langle r_{20}, w_{26}, o_{30} \rangle
\]

\[
\text{[20,26] Non Empty}
\]

\[
\text{Check if Cache Contains}
\]

\[
\text{Look Up Semantic descriptors}
\]

\[
\text{Returns True}
\]

\[
\text{Return result}
\]

![Semantic Caching Diagram](image)

Figure 4.5: Semantic Caching for boolean Sub-Queries

**Example 10**  For the query shown in Figure 4.5, when the tuple \(o_{30}\) arrives, the query is triggered and the outer result \(\langle r_{20}, w_{26}, o_{30} \rangle\) is formed. The Query-Interval extracted is \([20,26]\). The Interval is matched against the list of Intervals in the Semantic Cache. For the matched Cache Interval the boolean value of the isEmpty flag is set to True. It therefore returns true and the outer partial result is returned.

**Disadvantages of Interval Driven Semantic Caching**

- The main drawback of this method of caching is that when there are interval overlaps of certain nature, we are unable to make use of the already computed results and hence end up recomputing them. This is not the case with the Continuous Sliding Cache because in this case the cache grew only to the right.
• Even in the case of perfect overlap between a Query-Interval and a Cache-Interval there is a need to scan all other overlapping Cache-Intervals.

• Storing the cached results without duplicating them adds further complexity to this algorithm and hence reduces performance.

4.2 Continuous Sliding Cache with Semantic

Hence a hybrid approach is adopted combining the advantages of the two techniques discussed previously. In this method there will be an initial scan of the set of outer results to make decisions about the Cache-Intervals that would form. This technique guarantees that the cache will move only in one direction that is towards the right.

4.2.1 Continuous Cache with Semantics Design

The data structure for this technique is the same as for the Semantic Caching. For a given sub-query we will maintain a Cache-Interval and the results associated with the respective interval.

4.2.2 Cache Loading

The cache is loaded once for a set of outer query results. Such a pre-processing simplifies the storage of results without storing any duplicates. Thus when an outer query is triggered, a set of outer query results are produced. Given this set of outer sub-query results, we determine the maximum overlapping Query-Intervals and form Cache-Intervals for such overlapping Intervals. We precompute the results of the inner sub-query for these Cache Intervals and store them.

**Corollary:** Cache-Interval always moves to the right.

**Proof:** Proof by Contradiction.

Assume a triggering event \( e_t \) arrives. Let the previous triggering event be \( e_{pt} \). The maximum Query-Interval formed is \([t_x, t_y]\). Let the existing Cache-Interval be \([t_m, t_n]\). If the Cache-Interval
was to move to the left, \( t_x < t_m \). However, this can never be possible because if \( t_x \) is the left bound of the Query-Interval, it is obtained from the timestamp of an event instance of an outer query result say \(<... e_x, e_z, ... e_t >\). If \( e_x \) is joined with \( e_t \), it must have also joined with \( e_{pt} \) forming an outer query result \(<... e_x, e_z, ... e_{pt} >\). Thus the Cache-Interval should have been \([t_x, t_m]\) which is a contradiction.

### 4.2.3 Cache Usage

For a given outer query result, if the Query-Interval is found contained in the list of Cache-Intervals, the results are directly retrieved only from that Cache-Interval. There is no case of Overlapping Cache-Intervals. This is the main reason which saves this technique from the disadvantage of Semantic Caching. On the other hand, since we do not precompute any result, it also does not suffer from the disadvantage of the Continuous Sliding Cache.

```java
CacheUsage(Query q), ResultList ri 01 Interval maxInterval = ri-1.get(0).extractQueryInterval();
02 For(Interval queryInterval:
    ri-1.extractQueryInterval())
03 if(maxInterval < queryInterval)
    maxInterval = queryInterval
04 if(maxInterval.containedIn(CacheInterval))
05 computeNeeded = false;
06 else computeNeeded = true;
07 PartialCompute(temp,
     queryInterval);
08 else Reuse();
```

Figure 4.6: Continuous Semantic Cache Usage
Chapter 5

Cost Analysis for Event Expression Evaluation

This chapter shows the algorithmic costs of the alternate caching solutions. It assumes some average statistics of the data. 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

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{compute(qi)}$</td>
<td>The cost of computing results for a query $q_i$ independently</td>
</tr>
<tr>
<td>$NumE$</td>
<td>Number of total events received so far</td>
</tr>
<tr>
<td>$NumRE$</td>
<td>Number of relevant events received of the types in query set $Q$</td>
</tr>
<tr>
<td>$P_E$</td>
<td>Selectivity of all single-class predicates for event class $E$. This is the product of selectivity of each single-class predicate of $E$.</td>
</tr>
<tr>
<td>$P_{E_1,E_2}$</td>
<td>Selectivity of multi-class predicates between event class $E_1$ and $E_2$. If $E_1$ and $E_2$ do not have predicates, it is set to 1.</td>
</tr>
<tr>
<td>$P^{(E_1,E_2)}$</td>
<td>Selectivity of the implicit time predicate of subsequence $(E_1,E_2)$. The default value is set to 1/2.</td>
</tr>
<tr>
<td>$</td>
<td>S_i</td>
</tr>
<tr>
<td>$</td>
<td>S_{qi}</td>
</tr>
<tr>
<td>$Ov$</td>
<td>The fraction of overlap between the Cache-Interval and the Query-Interval $q_i$</td>
</tr>
</tbody>
</table>

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 ($C_{\text{compute}}$), and the cost to purge ($C_{\text{purge}}$) (Equation 5.1).

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

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

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

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

**Compute Cost for State-of-the-art stack-based execution.** For an event pattern query $q_i = \text{SEQ}(E_1, E_2 \ldots, E_i \ldots, E_n)$, $E_i$ is generating event for $1 < i < n$. Using stack-based pattern evaluation, $C_{\text{compute}(q_i)}$ is formulated in Equation 5.2.

$$C_{\text{compute}(q_i)} = |S_n| \times |S_{n-1}| \times P_{E_n,E_n-1} \times |S_{n-1}| \times P_{E_n,E_{n-1}} \times |S_{n-2}| \times P_{E_{n-1},E_{n-2}} + \ldots$$

$$= \sum_{i=n}^{2} |S_{i-1}| \times \prod_{j=i}^{n} |S_j| \times P_{E_j,E_{j-1}} \times P_{E_{j-1},E_{j-2}} / P_{E_i,E_{i-1}}$$  

(5.2)

**Iterative Nested Execution.** For a nested expression $q_i$, $q_{i, \text{root}}$ represents the outer most event expression and $q_{i, \text{child}_j}$ represents its $j$th child. $C_{\text{compute}_{q_i}}$ mainly consists of computation costs for $q_{i, \text{root}}$, computation costs for $q_{i, \text{child}_j}$ and joining costs as shown in Equation 5.3 with $C_{\text{join}}$.

$$C_{\text{compute}(q_i)} = C_{\text{compute}(q_{i, \text{root}})} + |q_{i, \text{root}}| \times (\sum_{j=1}^{n} C_{\text{compute}(q_{i, \text{child}_j})})$$  

(5.3)

**Iterative Nested Execution with Continuous Sliding Caching.** For a nested expression $q_i$, $q_{i, \text{root}}$
represents the outer event expression and $q_{i-child_j}$ represents its $j$th child. $C_{compute_qi}$ mainly consists of computation costs for $q_{i-root}$, cache maintenance costs for $q_{i-child_j}$ and joining costs as shown in Equation 5.4 and Equation 5.6.

\[
C_{compute(q_i)} = C_{compute_{qi-root}} + |S_{qi-root}| \times 
\left( \sum_{j=1}^{n} (C_{CacheAccess} + C_{Maintenance_{qi-child_j}}) \right) \tag{5.4}
\]

The maintenance cost of a cache is similar to the compute cost except that it is a partial computation, i.e. the results computed are only for the fraction of interval for which the cache does not contain the results already. The average overlap is estimated to be $Ov$.

\[
C_{Maintenance_{Flat(qi)}} = Overlap \times |S_1| \times |S_2| \times \ldots 
\times |S_{n-1}| \times |S_n| 
\times P_{E_1,E_2} \times P_{E_2,E_3} \times \ldots 
\times P_{E_{n-1},E_n} + \ldots 
\]

\[
= Ov \times \sum_{i=1}^{n-1} |S_{i+1}| \times P_{E_{i-1},E_i} \times \ldots 
\times |S_1| \times P_{E_1,E_2} \times |S_2| \times P_{E_2,E_3} \times \ldots 
\times |S_{n-1}| \times P_{E_{n-1},E_n} \times |S_n| 
\]

\[
\prod_{j=1}^{i} |S_j| \times P_{E_j,E_{j+1}} 
\]

where $P_{E_0,E_1} = 1$.

\[
C_{Maintenance_{qi}} = C_{Maintenance_{Flat(qi)}} + |S_{qi}| \times 
\left( \sum_{j=1}^{n} C_{Maintenance_{qi-child_j}} \right) \tag{5.6}
\]

**Iterative Nested Execution with Semantic Caching.**

The computation cost using Semantic Caching is same as the cost for Continuous Sliding Caching. However in this technique the $Ov$ could more often result in 1 due to the Query-Interval
overlapping the Cache Interval on the left side thus resulting in greater number of recomputations.

**Iterative Nested Execution with Continuous Semantic Caching.**

For this technique the Cache is maintained only once per triggering event.

\[ C_{compute}(q_i) = C_{compute_{q_i, root}} + |S_{q_i, root}| + \sum_{j=1}^{n} (C_{CacheAccess} + C_{Maintenance_{q_i, child_j}}) \]  

(5.7)

Similarly the maintenance cost of a continuous semantic cache is once per triggering event instead of once per outer query result.

\[ C_{Maintenance_{q_i}} = C_{MaintenanceFlat_{q_i}} + \sum_{j=1}^{n} C_{Maintenance_{q_i, child_j}} \]  

(5.8)
Chapter 6

Experimental Evaluation

6.1 Experimental Setup

The caching solutions have been implemented inside the stream management system called ECube [12] using Java. Experiments were run on Intel Pentium IV CPU 2.8GHz with 1GB RAM with Microsoft Windows XP operating system. Each query is processed based on a non-deterministic finite automata based approach stacks. In a nested query the processing of each subexpression follows the same strategy. Comparisons are made between the iterative processing technique, the alternative caching techniques and the state-of-the-art optimization technique including Rewriting [11] based on the overall execution time. That is we will run the query over the stock data and plot the execution time against the number of results produced. The queries used in the experiments have been shown above the charts. The diagrams show a tree based representation of the query plan. The Window which is specified as a WITHIN clause according to NEEL specifications have been mentioned in the description of each query. By default it is kept to 500 unless specified otherwise.
6.2 Data Description

The data contains stock ticker and timestamp information [1]. The portion of the trace used has 10,000 unique event instances. It is in the form of a text file which is read by a separate thread. A snapshot of a portion of the data has been shown in figure 6.1 Synthetic data is generated such that many inner query results are computed which are never used by an outer query result.

6.3 Rewriting Nested Queries

In [11] Mo et al developed a set of equivalence rules for rewriting nested NEEL expressions. They then proposed a step-wise procedure that applies these rewriting rules to transform a nested CEP query into an equivalent nonnested query thus opening the opportunity for query optimization. They then use a novel bit-marking process to share execution costs for shared sub-query evaluation by grouping certain sub-queries.
6.4 Analysis of Results

6.4.1 Effect of Window Sizes on the Caching techniques

The query shown in figure 6.2 shows a nested query which is run for three different Window sizes of 100, 500 and 1000 using the caching techniques and iterative processing technique. As complex event patterns are detected over the event stream results are outputted. 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.

![Figure 6.2: Varying Window Size](image)

Figure 6.2: Varying Window Size

The Figures 6.3 a, 6.4 a and 6.5 a demonstrate the effectiveness of our caching techniques over the iterative technique with increasing Window sizes from 100, 500 to 1000. The iterative

![Figure 6.3: Comparing execution time of queries using Semantic Caching, Continuous Sliding Caching, Continuous Semantic Caching and iterative technique with Window Size 100](image)

(a) Comparing Caching and iterative technique  
(b) Comparing alternative Caching techniques

The Figures 6.3 a, 6.4 a and 6.5 a demonstrate the effectiveness of our caching techniques over the iterative technique with increasing Window sizes from 100, 500 to 1000. The iterative
technique clearly take a much higher execution time for processing the same query compared to
the caching techniques for all Window sizes. The Figures 6.3 b, 6.4 b and 6.5 b zoom into the
charts in Figures 6.3 a, 6.4 a and 6.5 a respectively to carefully see the comparison between the
three caching techniques. It is clear that in all three cases, the Continuous Semantic Caching
performs better than the other two techniques. For the given data the number of recomputations are significantly large which make the Semantic Caching expensive compared to the Continuous Caching. However the window sizes do not seem to effect their relative performances significantly.

6.4.2 Effect of Length of Subqueries on the Caching techniques

![Diagram showing varying length of child queries]

Figure 6.6: Varying Length of Children Queries

Figure 6.8 shows three queries with varying lengths of the inner sub-query from two event types to four event types. The Window size is kept constant at 500 and the sub-query has two event types in all of them.

The three figure in Figure 6.7 shows the comparison between the three caching techniques with varying sub-query length. Clearly the length of the sub-query has an effect on the relative performances. The Continuous Semantic Caching and the Continuous Sliding Caching techniques take much less execution time compared to the Semantic Caching as the length of the sub-queries in increased. This is as we would expect because the time for recomputation increases with increase in size of the sub-query.

6.4.3 Effect of Number of Subqueries on the Caching techniques

Figure 6.6 shows 3 queries with varying number of the inner sub-queries from one sub-query to three sub-queries. The Window size is kept constant at 500 and the number of sub-query is kept constant at one.
Figure 6.7: Comparing execution time with Semantic Caching, Continuous Sliding Caching and Continuous Semantic Caching with with varying length of sub-queries

(a) Sub Query Length = 2

(b) Sub Query Length = 3

(c) Sub Query Length = 4

Figure 6.8: Varying Number of Positive Children Queries

Charts shown in Figure 6.9 show a comparison between the three caching technique as we vary the number of sub-queries shown in Figure 6.8. Again as expected we see that the Continuous Semantic Caching takes a much less execution time compared to Continuous Caching or Seman-
Figure 6.9: Comparing execution time for Semantic Caching, Continuous Sliding Caching and Continuous Semantic Caching with varying number of sub-queries.

As the number of subqueries is increased, the difference in performance between Semantic Caching and Continuous Sliding Caching increases as expected because for every subquery the number of recomputations is much larger for Semantic Caching than Continuous Sliding Caching.

### 6.4.4 Comparing Caching techniques using Synthetic Data

The chart in Figure 6.10 is the query shown in Figure 6.2 run over a data where the Semantic Caching performs better than the Continuous Caching. This dataset clearly shows how the over-computation of un-used intermediate results by the Continuous Sliding Caching technique greatly...
deteriorates performance of the technique. However even for this data, the Continuous Semantic Caching outperforms the other two techniques because it not only minimizes recomputations but also minimizes computing unused results for an inner subquery.

### 6.4.5 Comparing Caching techniques against Rewriting technique

We will now compare our Continuous Semantic Caching technique against Rewriting methodology where Rewriting rules are applied to nested queries to normalize them and adopt an optimized shared execution strategy to execute normalized queries. The comparison is made against the Continuous Semantic Caching technique which has consistently performed better over the other two caching mechanisms.

Figure 6.11: Varying Length of Boolean Sub Queries

Figure 6.11 shows three queries with boolean sub-queries with varying lengths of the inner sub-query from two event types to four event types. The Window size is kept constant at 500 and
the sub-query has two event types in all of them.

![Graph](image1.png)

(a) SubQuery Length = 2

![Graph](image2.png)

(b) SubQuery Length = 3

![Graph](image3.png)

(c) SubQuery Length = 4

Figure 6.12: Comparing execution time for Rewriting technique, Continuous Semantic Caching and iterative technique with varying number of sub-queries

The charts in Figure 6.12 show that both the two optimization techniques perform much better than the iterative technique with respect to execution time. However the performance of the techniques vary with the complexity of the boolean queries. As the length of the boolean queries is increased, the Continuous Semantic Caching technique takes a much shorter execution time compared to the Rewriting technique.

Overall, it be can be concluded that both the run-time optimization - caching and compile time optimization - rewriting techniques make the queries run much faster. Continuous Sliding Cache and Semantic Caching bring about a large improvement in the execution time over the
iterative technique. The Semantic Caching results in the making of too many overlapping intervals which greatly increases the computation overhead. However, the Continuous Semantic Caching consistently takes a shorter time of execution than the other two mechanisms. It is a hybrid between the other two mechanisms which results in making fewer number of overlapping intervals and thus emerges as the winner.
Chapter 7

Related Work

7.1 Caching

A cache is a component that transparently stores data so that future requests for that data can be served faster. Caching has been explored in the context of Operating Systems [19], Web systems [15] and database systems [7] alike. A few commonly followed replacement policies for caching are Most Frequently Used Cache, or Least Recently Used Expiry (LRU Cache) for applications which deal with a large amount of data - too large for all of it to be cached. A Time to Live (TTL Cache) is often used in a scenario where we know that the latest entries are going to be most used. Event based caches are used where we cache the data till an external event forces the cache to be expired. In our context a continuous sliding cache most closely correspond to the event based cache where events would expire as the Window slides.

[21] proposed a method for maintaining a semantic cache of materialized XPath views. The cached views include queries that have been previously asked, and additional selected views. Our work borrows the concept of attaching semantics to the cache.
7.2 Nested SQL and Decorrelation

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 [9], 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.

7.3 Other CEP Systems

To the best of our knowledge, existing CEP systems [23, 2, 16, 3, 10] support the execution of only flat sequence queries. SASE [23] and Cayuga [2] proposed an important processing model for CEP which is based on the nondeterministic finite state automata. [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 or sequences. [23] introduced the SASE Event Language which was restricted to sequence and negation operators. Both the above systems do not consider nesting of patterns. They also do not consider operators like AND and OR. However Cayuga [2] does allow sub-queries in the FROM clause. As an extension to the work in SASE [23], SASE+ [8] was proposed which supported Kleene star over event streams. However this too did not tackle nesting of patterns.

Zstream [16] 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 over mul-
tiple expressions nor of nested CEP expressions. 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 [14] Liu et al. describe an iterative strategy to process Nested CEP queries which is too the default method we will compare against in this current work. In [11] Liu et al. developed a set of equivalence rules for rewriting nested \textit{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. They then use a novel bit-marking process to share execution costs for shared sub-query evaluation by grouping certain sub-queries. 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 and Future Work

8.1 Conclusion

This thesis mainly focusses on optimizing iterative processing of Nested Complex Event Processing queries by caching intermediate results. In particular, we designed a Continuous Sliding Caching methodology for storing intermediate results. We described techniques for incrementally loading, purging and exploiting the cache content. We then went on to propose certain optimization techniques over the Continuous Sliding Caching technique by adding semantics to it. We then experimentally compare our caching methodologies against standard iterative processing technique for CPU processing time. Our optimized strategy win by a large margin over iterative processing. The performance of the basic caching technique and the optimized caching techniques vary under different conditions. We also compare the performance of our caching technique against state-of-the-art method of processing Nested CEP queries which is Rewriting [11]. We discuss the performance of each method with their charts under varying conditions.

8.2 Future Work

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

First, we would like to use the cost-model for the caching techniques as well as for the Rewrit-
ing technique and adopt a hybrid model for optimizing NEEL queries using both caching and rewriting techniques based on data statistics.

Secondly, we assume here that memory is never a limited 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.

Thirdly, in case of limited memory, we will also look into cache replacement policies.
Bibliography


