Processing Nested Complex Sequence Pattern Queries Over Event Streams

Processing Nested Complex Sequence Pattern Queries over Event Streams

Mo Liu, Medhabi Ray, Elke A. Rundensteiner, Daniel J. Dougherty
Worcester Polytechnic Institute, Worcester, MA 01609, USA
(liumo|medhabi|rundenst|dd)@cs.wpi.edu
Chetan Gupta, Song Wang, Ismail Ari‡, Abhay Mehta
USA Hewlett-Packard Labs, USA
‡Ozyegin University, Turkey
(chetan.gupta|songw|abhay.mehta)@hp.com ‡Ismail.Ari@ozyegin.edu.tr

ABSTRACT

Complex event processing (CEP) has become increasingly important for tracking and monitoring applications ranging from health care, supply chain management to surveillance. These monitoring applications submit complex event queries to track sequences of events that match a given pattern. As these systems mature the need for increasingly complex nested sequence queries arises, while the state-of-the-art CEP systems mostly focus on the execution of flat sequence queries only. In this paper, we now introduce an iterative execution strategy for nested CEP queries composed of sequence, negation, AND and OR operators. Lastly the promise of applying selective caching of intermediate results to optimize the execution. Our experimental study using real-world stock trades evaluates the performance of our proposed iterative execution strategy for different query types.

1. INTRODUCTION

Complex event processing (CEP) has become increasingly important in modern applications, ranging from supply chain management for RFID tracking to real-time intrusion detection [1, 2, 3]. CEP must be able to support sophisticated pattern matching on real time event streams including 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, 5]. 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". A query Q_1 = *SEQ (Recycle r, Washing w, NOT SEQ(Sharpening s, Disinfection d, Checking c), Operating op) with the condition that ([ID] (equality on ID) and op.ins-type = "surgery")* 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 complex negation specifying the non-occurrence of composite event instances, such as negating the composite event of sharpened, disinfected and checked subsequences.

However, the state-of-the-art CEP in the literature including SASE [1] and ZStream [3] do not support such nested queries. Even though the Cayuga system [2] mentions composable queries, they assume the negation filter is only applied to a single primitive event type within the SEQ pattern. Our objective however is to allow the specification of negation within any level of the nested query as in the above example. While CEDR [6] allows applying negation over composite event types within their proposed language, the execution strategy for such nested queries is not discussed. In short, no processing mechanisms for nested complex negation of CEP queries have been discussed in the literature to date. In this work, we address this gap by designing an execution strategy specifically to handle nested CEP queries specified by the nested complex expression query language NEEL[1]. The semantics of this language is presented in [7].

Our contributions in this paper include:

- We introduce an algebraic query plan for nested CEP queries expressed in NEEL.
- We design an iterative topdown execution strategy based on the algebraic plan that applies a window constraint tightening technique designed to correctly process nested sub-queries. Intermediate results are pushed up conservatively for delayed resolution when a child query can't be fully answered locally for nested negation.
- We experimentally evaluate our proposed execution strategy studying nested queries with different properties including sub-query lengths and nesting levels on real data streams.
- Lastly selective caching of intermediate results is introduced as technique for optimizing the execution.

2. NESTED CEP QUERY MODEL

2.1 Event Model

An event instance is an occurrence of interest 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_i, respectively, with e_i.ts $\leq e_i$.te. For a primitive event instance e, e_i.ts = e_i.te. For simplicity, we use the subscript i attached to a primitive instance e to denote the timestamp *i*.

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

A composite event instance is composed of constituent primitive event instances $e = < e_1, e_2, ..., e_n >$. A composite event instance e occurs over an interval. The start and end timestamps of e are equal to $\min\{e_i.\text{ts} \mid e_i \text{ in e }\}$ and $\max\{e_i.\text{te} \mid e_i \text{ in e }\}$, 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 [8]. *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. $e_i \in E_j$ denotes that e_i is an instance of the type E_j. Suppose one of the attributes of E_j is attr and $e_i \in E_j$, then we use e_i.attr to denote e_i's value for that attribute.

2.2 The Nested Complex Pattern Query Language NEEL

We now briefly introduce the *NEEL* query language for specifying complex nested event pattern queries [1, 6, 9] as an extension of basic non-nested languages from the literature. *NEEL* supports the nesting of AND, OR, Negation and SEQ operators at any query nesting level as in Table 1.

<**Query**>::= PATTERN <event-expression> WITHIN <window> [RETURN <output-specification>]
<event-expression> = <ex>
<ex> ::= **SEQ**((<ex> | **!** (<ex>, [<q>]))*,<ex>, (<ex> | **!** (<ex>, [<q>]))*, [<q>]) | **AND**((<ex>, (<ex> | **!** (<ex>, [<q>]))*, [<q>]) | **OR**((<ex>)+, [<q>]) | (<primitive-event type>, [<var>])
<primitive-event type> ::= E_1 | E_2 | ... <var> ::= event variable e_i <q>::= (<elemqual>)* <elemqual> ::= <var>.attr <op> <var>.attr | <var>.attr <op> constant <op> ::= < | > | ≤ | ≥ | = | != <window>::= time duration w | tuple count c

Table 1: NEEL Query Language

A primitive event type E_i itself is an event expression. If E_1, E_2 ,..., E_n are event expressions, an application of SEQ, AND and OR over these event expressions is again an event expression [8]. In other words, nesting of AND, OR and SEQ operators is supported.

SEQ in the PATTERN clause specifies a particular order in which the event instances of interest should occur. If there is a ! (NOT) symbol before an event expression in an operator, we say that the event expression marked by ! is to be negated. Event instances that satisfy the positive components with no events in the stream relative to this match satisfying the negative components are output. If several adjacent event types are marked by ! in a SEQ operator such as SEQ(E_1, ! E_2, ! E_3, E_4), the query requires the non-existence of any E_2 and E_3 events in either order between E_1 and E_4 events within the input stream. In other words $< e_1, e_3, e_4 >$ and $< e_1, e_2, e_4 >$, $< e_1, e_3, e_2, e_4 >$ and $< e_1, e_2, e_3, e_4 >$ all do not result in a valid match for this query.

An event expression exp_i can be used as a component in SEQ, AND and OR operators to construct another expression exp_j. Then we call exp_j the *outer* or parent *expression* of exp_i and exp_i the *inner* (or child) *expression* of exp_j. Qualification in the PATTERN clause contains predicates on single attributes or on attributes across multiple event types in the query [6, 1]. The event variables defined in an outer expression are visible within the scope of its own nested inner expressions. Local predicates are specified directly inside exp_i. Correlated predicates involving events from both an outer and an inner expression are associated with the innermost expression that define an event in the predicate. Correlated predicates involving two adjacent sibling expressions are not allowed since the events in one inner expression are not visible in any sibling.

The WITHIN clause indicates the temporal interval within which the event instances of interest must occur. The RETURN clause transforms the set of matching event instances extracted by the query into a complex event as specified in the output specification.

Q_1 below in Figure 1 is a sample query expressed by *NEEL*.

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

Figure 1: Sample Query Q_1 for Hospital Hygiene

2.3 Nested CEP Query Plan

A query expressed by a *NEEL* specification is translated into a default algebraic query plan composed of the following algebraic operators: Window Sequence (*WinSeq*), Window Or (*WinOr*) and Window And (*WinAnd*). During query transformation, each expression in the event pattern is mapped to one operator node in the query plan. The same window w is assigned to all operator nodes. *WinSeq* first extracts all matches to the positive components specified in the query, and then filters out events based on negative components as specified in the query. *WinOr* returns an event e if e matches any one of the event expressions specified in the WinOr operator. *WinAnd* computes the cross product of its positive components. For queries expressed by *NEEL*, predicates are placed into the respective algebra operators in the nested event expressions (see Section 2.2).

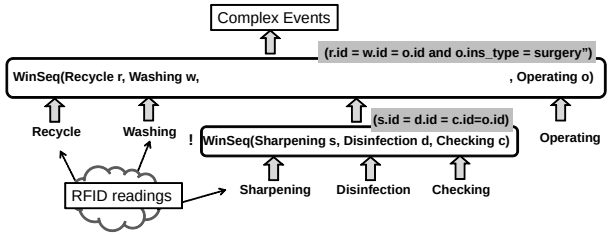


Figure 2: Basic Query Plan

EXAMPLE 1. *Figure 2 depicts the query plan for query Q_1 in Figure 1. The two SEQ expressions in Q_1 are transformed into two WinSeq operator nodes in the plan. The predicate s.id = d.id = c.id = o.id is placed with the inner WinSeq operator node containing the negative component. The other predicates are attached to the topmost WinSeq operator node.*

3. NESTED CEP QUERY PROCESSING

3.1 Execution of Individual Operators

For simplicity, we briefly review the implementation strategy of one of the operators, namely, the SEQ operator, while the others can be implemented in a similar fashion. We adopt the state-of-the-art stack-based strategy for SEQ execution [1, 10, 11]. We associate a stack with each event type in the query. Each received event instance is simply appended to the end of the stack of its type. Event instances are augmented with pointers ptr_i to adjacent events to facilitate quick locating of related events in other stacks during result construction.

The arrival of an event instance e_m of the last event type E_m of a query q_i triggers the compute function of q_i[2]. The result construction is done by a depth first search along instance pointers ptr_i rooted at that last arrived instance e_m. All paths composed of edges "reachable" by that root e_m correspond to one matching event sequence returned for q_i. When negative event types are specified in WinSeq, then during sequence construction any edges "reachable" from the root e_m are skipped if an instance of the negative event type is found in the corresponding stream position. Events that are outdated based on the window constraints are purged.

3.2 Iterative Nested Execution Strategy

Following the principle of nested query execution for SQL queries [12, 13, 14, 15], 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 3) 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 3.3 and 3.4, respectively.

```
Interval IntervalConstraints (Result r_j, Query q_i)
// r_j is one partial result of the outer query
01 Interval ts;
02 if(root operator of q_i is SEQ)
   // gets the position of q_i in outer query
03 { nestedPosition = getNestedPos(q_i);
       // if outer query starts with sub query q_i
04 if(nestedPosition == 0)
       // left bound is time of last event in result r_j - W
05     ts_left = getTime(r_j.LastEve) - W;
       // if outer query ends with sub query q_i
06 if(nestedPosition == r_j.size)
       // right bound is time of first event in result r_j + W
07     ts_right = getTime(r_j.FirstEve) + W;
08 else
09     {ts_left=getTime(r_j.get(nestedPos-1))
10     ts_right=getTime(r_j.get(nestedPos))}
11 if(root operator of q_i is AND)
12 {ts_left = getTime(r_j.lastEve) - W;
13 ts_right = getTime(r_j.lastEve); }
14 if(root operator of q_i is OR)
15 {ts_left = getTime(r_j.lastEve) - W;
16 ts_right = getTime(r_j.lastEve); }
17 return ts;
```

Figure 3: Algorithm to Compute Interval Constraints for an Inner Query Q_i Given an Outer Partial Result r_j

3.3 Processing Nested Queries with Negation

We now describe our approach of supporting negations in nested queries. In SASE [1, 11, 10], flat queries can have negations and they are dealt with using the timestamp information. More precisely, if a query has a negative A between positive B and C event types, they first evaluate the query without the negation, i.e., they compute all B-C pairs. Then for every result generated they check if an A event occurred between the qualified B and C events. If it occurs, such pairs are discarded. 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) result pairs without any B and C events in between them would be returned.

For negative event types at the end of a query, postponed sequence evaluation is applied. That is the execution is continued till the last negation as per our iterative strategy however results are not output. Instead at t he arrival of every new event we note the time stamp of the event and also check whether it is a triggering event for the last negative part of the query. If it is not a triggering event, based on the time stamp of the arriving event, some results from the buffer may be output and removed from the buffer. If it is a triggering event, the negative part of the query is executed and if it produces some partial results, the result buffers of the outer query are completely cleared. However if the negative ending part of the query does not produce any results, some results are output and removed from the result buffers based on the time stamp of the arriving event.

In our nested query model, a sub-query as a whole could also be negated. For example, 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.

We distinguish between the following positions in which the negation clause can occur.

- *Bound by Upper Query.* The existence of a negative event instance could be bounded by positive event instances in the direct upper queries. Examples of this category include SEQ(A, !B, C) and SEQ(A, SEQ(B, !C), D). In the second query, negative C events are bound by B and D events. B events that do not have any C events occurring after them and before D events are passed up to the upper query operator. All B events passed up will be joined with the outer SEQ(A, D) result to construct SEQ(A, SEQ(B, !C), D) results.

- *Bound by Adjacent Query.* The existence of a negative event instance could be bound by positive event instances of an adjacent sibling sub-query. Examples of this type include SEQ(A, SEQ(B, !C), SEQ(D, E), F) or SEQ(A, !B, SEQ(C, D), E). In this case, we apply a contextual delayed constraint technique. Namely, we conservatively pass up additional intermediate results as compared to the case described above. In SEQ(A, SEQ(B, !C), SEQ(D, E), F), outer SEQ(A, F) results $< a_i, f_j >$ are constructed. The constraint window for both children sub-queries SEQ(B, !C) and SEQ(D, E) is [a_i.te, f_j.ts]. When processing the sub-query SEQ(B, !C) within this constraint window, any event of type B should be passed up. We cannot filter out events of type B even though C events exist after it within its constraint window. The reason is that the right bound of the interval constraint of the query SEQ(B, !C) is decided by the results of the query SEQ(D, E). We should not have a C event between a B or D event. However, it is not possible to know time stamps of D events while still processing the query SEQ(B, !C). Hence the decision is postponed until the results of both the inner queries are returned to the outer query and then the filtering of results takes place based on the presence of C events.

3.4 Processing Nested Queries with Predicates

The approach of handling sub-queries with correlated predicates is similar to the basic nested execution described above except that the join is not only based on timestamps but also on other predicates. Below, we list the different cases for predicate handling.

[2]if E_m is a negative event type, postponed sequence evaluation is applied. We omit the details here.

- *Local predicates.* 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 in Figure 2, Operating events where the instrument type is not equal to "surgery" 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 requires events in the inner sub-queries have the same tool id as the outer match. For each outer SEQ(Recycle r, Disinfection d, Operating o) match, the tool id information for the operating instance is thus passed down to the children sub-queries. Inner query results involving events having the same tool id with the outer match are returned to the upper query. As can be seen in Table 1, predicates on negative components are associated directly with the later and not with the operator as a whole. They are thus only evaluated for those subqueries, for which the positive parent context match has already been established.

3.5 Putting It All Together

At compile time, queries with negation bounded by an adjacent sub-query (as discussed in Section 3.3) are marked with label "delayed constraint". More specifically, if a query q_i is labeled as "delayed constraint", it not only needs to pass up potential q_i results, but also negative events are passed up as we can't determine locally if they are in violation or not. The pseudo code of the nested execution algorithm is given in Figure 4. This function is called whenever a new event of the last positive event type in the outer query arrives. Figure 5 shows the algorithm for joining partial outer results with its children query results.

EXAMPLE 2. *Consider the query Q = SEQ(Recycle r, ! SEQ(Washing w, Drying dr, Sharpening s), Disinfection d, SEQ(Checking c, Relabeling rl), Operating op). When event instances of types Recycle, Washing, Drying, Sharpening, Disinfection, Checking, Relabeling and Operating arrive, they are pushed into their respective stacks. The outer query is first evaluated for a given window size followed by the inner sub-query. The outer query construction is triggered by the arrival of Operating events which are of the rightmost positive event type in the root query. For every partial result* $< r_i, d_j, op_k >$ *of the outer query SEQ(Recycle r, Disinfection d, Operating op), we compute the window constraints for its children queries. For details, see Figure 3. If we were to evaluate this query without predicates, all results for SEQ(Washing w, Drying dr, Sharpening s) and SEQ(Checking c, Relabeling rl) would be constructed for events that occur within [*r_i*.te,* d_j*.ts] and [*d_j*.te,* op_k*.ts], respectively. The outer operator joins with all results returned by its positive sub-query SEQ(Checking c, Relabeling rl). The outer result* $< r_i, d_j, op_k >$ *fails if results of the negative child query SEQ(Washing w, Drying dr, Sharpening s) exist. When evaluating Q with correlated predicates [id], the id is passed down from the outer query to the children sub-queries. Results involving events with the same id are constructed in the sub-queries.*

4. PERFORMANCE EVALUATION

The objective of our evaluation is to verify if our strategy gives the correct results so that they can be used as a benchmark to compare alternate future methods against. We verify using various types of queries. We also make note of the execution time to test the effectiveness and practicability of our method.

4.1 Experimental Setup

```
NestedExecution (query q_i, event e_i, Window W))
01 if(e_i triggers q_i result construction)
02 {Interval ts; ts_left=e_i.ts - W; ts_right=e_i.ts
   RecursiveCompute(q_i, e_i, ts)}
// compute q_i results
RecursiveCompute(query q_i, event e_i, ts)
01 finalResult fr[];
   buffers buf_children[];
02 result r[] = selfCompute( q_i , e_i);
03 if (q_i has no children queries)
04    {if(q_i ∈ labeledSubQueries (Sec  3.5))
05    return r[] with negative events in q_i;
06    else return r[]; }
07 else for each result r_j belongs to r[]
08       for each inner query child_j of q_i
09       Interval ts =
         IntervalConstraints(r_j, q_i.childj);
      // compute constraint window for each sub-expression
10       RecursiveCompute(q_i.child_j, e, ts);
11       for each inner query child_j of q_i
12       if (Eval(q_i, q_i.child_j, buf_children))
      // join positive children results

14       continue;
      // stop evaluation if a negative component is not empty.
15
```

Figure 4: Nested Execution Strategy

```
Eval (Query q_i, Query q_j, Buffer buf_children))
01 if (q_j ∈ labeledSubQueries)
02    tighten q_j results with negative events
03 if (q_j is a positive query in q_i)
04    join q_i and q_j results; return true;
05 else if(q_j.results are not empty)
      // q_j is a negative component
06    return false;
```

Figure 5: Result Evaluation

We have implemented our proposed nested query processing framework within the stream management system CHAOS [16] using Java. We ran the experiments on Intel Pentium IV CPU 2.8GHz with 4GB RAM. We evaluated our techniques using the real stock trades data from [17] with 10,000 event instances with a sliding window of size 10 ms.The data contained stock ticker, timestamp and price information.

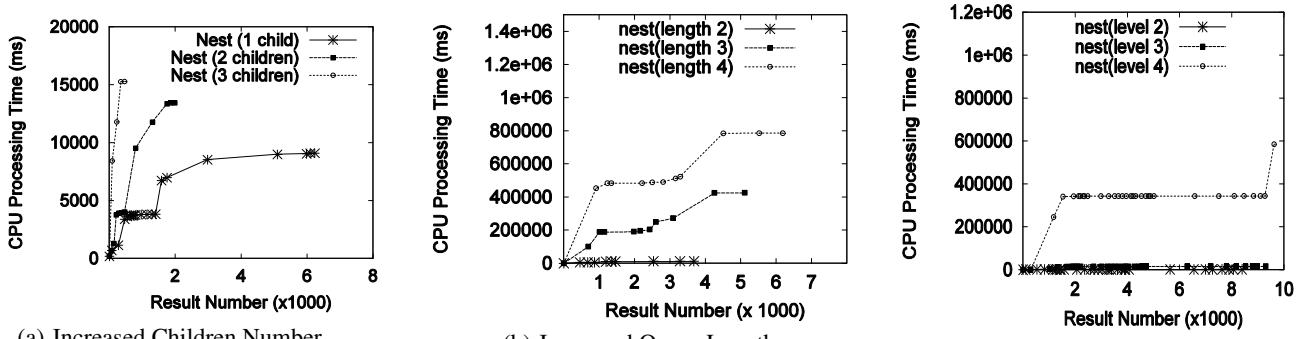
4.2 Varying Children Subquery Number

The first experiment processed queries with increased number of sub-queries from 1 to 3 (Figure 6(a)). q_3 generates minimum results using maximum processing time among the three queries. q_3 has more sub-queries to process which thus consumes more CPU processing time. Also, more outer SEQ(MSFT,ORCL,IPIX,INTC) results are filtered in q_3 as more constraints exist as compared to the other queries. As expected, the computation time increases with the number of sub-queries because the probability of finding patterns decreases with an increasing number of event types.

```
Increased Children Number:
q1=SEQ(MSFT,!SEQ(RIMM,AMAT),ORCL,IPIX,INTC);
q2=SEQ(MSFT,!SEQ(RIMM,AMAT),ORCL,!SEQ(YHOO,DELL),IPIX,INTC);
q3=SEQ(MSFT,!SEQ(RIMM,AMAT),ORCL,!SEQ(YHOO,DELL),IPIX,
        !SEQ(CSCO,QQQ),INTC);
```

4.3 Varying Subquery Lengths

The second experiment processed the queries below with increased sub-query lengths (from 2 to 4) as depicted in Figure 6(b).



(a) Increased Children Number

(b) Increased Query Length

(c) Increased Nesting Levels

Figure 6: Evaluating Nested Patterns

We observed that q_6 generates the most number of results and uses the most CPU processing time among the three queries. This is because q_6 includes the sub-query with the longest length which consumes more computational time. As expected, less outer SEQ(MSFT, ORCL,INTC) results are filtered in q_6 as the existence of a longer pattern is relatively less likely as compared to the other queries with shorter patterns within the same input stream.

```
Increased Query Length:
q4=SEQ(MSFT,!SEQ(RIMM,AMAT),ORCL,INTC);
q5=SEQ(MSFT,!SEQ(RIMM,AMAT,YHOO),ORCL,INTC);
q6=SEQ(MSFT,!SEQ(RIMM,AMAT,YHOO,DELL),ORCL,INTC);
```

4.4 Varying Subquery Nesting Levels

The third experiment processed the queries below with increased sub-query nesting levels as depicted in Figure 6(c). q_9 generates the most number of results and uses the most CPU processing time among the three queries. It is because q_9 includes the sub-query with the largest nesting levels which consumes more time to be computed. Less outer SEQ(MSFT, ORCL, INTC) results are filtered as it is relatively infrequent to have more events in levels occur in a sequence.

```
Increased Nesting Levels:
q7=SEQ(MSFT,!SEQ(IPIX,QQQ),ORCL,INTC);
q8=SEQ(MSFT,!SEQ(IPIX,SEQ(RIMM,AMAT),QQQ),ORCL,INTC);
q9=SEQ(MSFT,!SEQ(IPIX,SEQ(RIMM,SEQ(YHOO,DELL),AMAT),QQQ),
          ORCL,INTC);
```

5. NESTED QUERY OPTIMIZATION

Although the results of nested CEP queries obtained from the iterative execution strategy are correct, it produces results at a very slow rate which is attributed to the re-computation of the results for inner sub-queries every time an outer triggering event arrives which makes the processing expensive. To tackle this deficiency, we propose to cache and incrementally maintain the inner query results. Due to the sliding window, many intermediate results would continue to be valid from one sliding window to the next. Previously calculated results of the previous window should be cached and then be reused in the new window. In this paper we will only propose a direction for such an optimization technique. However this technique is not generic and cannot support negation or predicate correlation.

- **Cache Interval Extraction.** Assume $Q_i = SEQ(E_1, \ldots, E_i, SEQ(E_{i+1}, \ldots, E_{i+j}), E_{i+j+1}, \ldots, E_n)$. For a given triggering event $e_n \in E_n$, the left bound of the interval attached to the subexpression SEQ(E_{i+1}, …, E_{i+j}) is given by e_i.ts such that e_i has the minimum timestamp among all events of type E_i which have arrived so far. Similarly, the right bound of the interval is given by an event e_{i+j+1}.ts such that e_{i+j+1} has the maximum timestamp among all events of type E_{i+j+1} which have arrived so far. The extracted interval is attached to each cache representing the valid time period for the cached results.

- **Interval-driven Cache Expansion.** We update the cache content when a new triggering event e_t arrives. That is, given a new triggering event instance e_i, we calculate the new cache interval. For each subexpression, we compare the interval [i, j] attached to the cache to the new interval [m, n]. By the way our algorithm works, $i = m$, since the left bound is maintained at the event with minimum timestamp. We compute the sub-query SEQ($E_{i+1}, \ldots, E_{i+j}$) for all triggering events e_{i+j} between the interval [j, n] New results are appended to the cache for each subexpression triggered by events occurring between the right bounds of [j, n].

- **Interval-driven Cache Reduction.** When a triggering event e_t arrives, events with timestamp less than e_t - window are purged from their stacks. Similarly, caching results involving events with timestamp less than e_t - window are deleted from the cache as the window constraint will be violated if these results join with the new triggering event e_t in the final result.

EXAMPLE 3. *In Figure 7, when the triggering event o_{26} arrives, it is inserted into the Operating stack and triggers execution. [1, 15] and [8, 26] are extracted time intervals for the subexpressions SEQ(Washing, Drying, Sharpening) and SEQ(Checking, Relabeling), respectively. SEQ(Washing, Drying, Sharpening) results are constructed based on all events that occurred during [1, 15]. Similarly, SEQ(Checking, Relabeling) events occurring during [8, 26] are constructed and cached. When the new triggering event o_{30} arrives, we determine the interval for SEQ(Washing, Drying, Sharpening) is still [1, 15]. Thus the cache is still complete and thus we can reuse results in the cache. For subexpression SEQ(Checking, Relabeling), we find the new interval [8, 30] overlaps with the previous interval [8, 26]. Conceptually, we could reuse the caching results related to [8, 26] and we must compute the new additions to our cache. New SEQ(Checking,Relabeling) results are triggered by Relabeling events occurring between [26, 30] such as rl_{28}. Assume the window size is 30. When o_{34} arrives, all caching results involving primitive events with time-stamp less than 4 expire. So $< w_2, dr_6, s_7 >$, $< w_2, dr_3, s_7 >$ etc are deleted from the cache. The meta-data attached to the cache for SEQ(Washing, Drying, Sharpening) is updated from [1, 15] to [4, 15].*

5.1 Evaluating Optimized Nested Execution: Caching Results

We process query q_{10} comparing the optimized execution by the caching technique to the one without caching as in Figure 8. Caching helps in avoiding repeated computation for the subquery SEQ(QQQ,AMAT,DELL) as our results demonstrate. Clearly, we

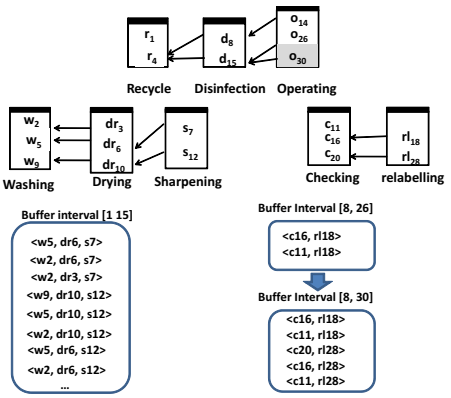


Figure 7: Interval Driven Subexpression caching

will have different gain with different reuse opportunities which may be caused by larger windows, more expensive sub-queries, etc.

```
Increased Nesting Levels:
q10 = SEQ(YHOO,SEQ(QQQ,AMAT,DELL),ORCL,IPIX);
```

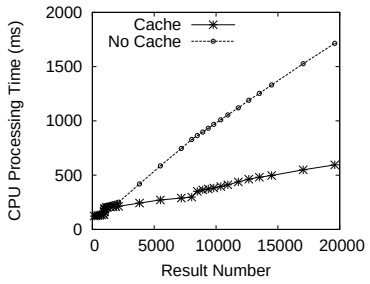


Figure 8: Interval-Driven Caching

6. RELATED WORK

The existing CEP systems [1, 2, 3, 6] do not focus on the execution of nested sequence queries as tackled here. The query language of the CEDR [6] system supports nested sequence queries. However, the execution strategy for nested queries is not given.

Complex queries used in decision support applications often have multiple correlated sub-queries and table expressions, possibly across several levels of nesting. It is usually inefficient to directly execute a correlated query. Consequently, algorithms such as magic decorrelation [18] and complex query decorrelation [19] have been proposed to decorrelate the query. However, existing decorrelation algorithms deal with only relational queries, that is, these algorithms are neither described nor tested in the CEP streaming context.

For SQL queries, [20] discusses whether a query result should be admitted to the cache and which results are to be purged in the static data context. In semantic caching [21], a semantic description of the data in a cache is maintained which allows for a compact specification. Semantic descriptors have also be shown to be of importance for query caching in the XML context [22, 23, 24]. However, sophisticated cache matching algorithms had to be designed to deal with query containment, namely, with extracting related XQuery subexpressions possibly with alternate hierarchical XML structures yet the same content [22].

7. CONCLUSION

In this paper, we introduced a comprehensive iterative execution strategy for processing nested CEP queries. An algebraic query plan for the execution of nested CEP queries was designed. We then developed a window constraint tightening technique to correctly process sub-queries. We also presented execution strategies for handling predicates in nested queries. Optimization using interval driven cache expansion and reduction was introduced. We plan to study additional optimization techniques in the future.

8. ACKNOWLEDGEMENTS

This work is supported by HP Labs Innovation Research Program and National Science Foundation under grants NSF IIS 0917017. Ismail Ari is supported by TUBITAK Grant 109E194. We thank Database System Research Group at WPI for valuable comments.

9. REFERENCES

[1] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams." in *SIGMOD Conference*, 2006, pp. 407–418.

[2] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White, "Cayuga: A general purpose event monitoring system." in *CIDR*, 2007, pp. 412–422.

[3] Y. Mei and S. Madden, "Zstream: a cost-based query processor for adaptively detecting composite events," in *SIGMOD Conference*, 2009, pp. 193–206.

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

[5] "Wireless sensor networks for home health care," pp. 832–837, 2007.

[6] R. S. Barga, J. Goldstein, M. Ali, and M. Hong, "Consistent streaming through time: A vision for event stream processing." in *CIDR*, 2007, pp. 363–374.

[7] M. Liu, E. Rundensteiner, D. Dougherty, C. Gupta, S. Wang, and A. Mehta, "Nested complex event processing for real-time event analytics," in *BIRTE*, 2010.

[8] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, "Composite events for active databases: Semantics, contexts and detection." in *VLDB*, 1994, pp. 606–617.

[9] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White, "Cayuga: A general purpose event monitoring system." in *CIDR*, 2007, pp. 412–422.

[10] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams," in *SIGMOD Conference*, 2008, pp. 147–160.

[11] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman, "On supporting kleene closure over event streams," in *ICDE*, 2008, pp. 1391–1393.

[12] P. Seshadri, H. Pirahesh, and T. Y. C. Leung, "Complex query decorrelation," in *ICDE*, 1996, pp. 450–458.

[13] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan, "Magic is relevant," in *SIGMOD Conference*, 1990, pp. 247–258.

[14] E. Wong and K. Youssefi, "Decomposition - a strategy for query processing," *ACM Trans. Database Syst.*, vol. 1, no. 3, pp. 223–241, 1976.

[15] J. M. Smith and P. Y.-T. Chang, "Optimizing the performance of a relational algebra database interface," *Commun. ACM*, vol. 18, no. 10, pp. 568–579, 1975.

[16] C. Gupta, S. Wang, I. Ari, M. Hao, U. Dayal, A. Mehta, M. Marwah, and R. Sharma, "Chaos: A data stream analysis architecture for enterprise applications," in *CEC'09*, 2009, pp. 33–40.

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

[18] C. Beeri and R. Ramakrishnan, "On the power of magic," *J. Log. Program.*, vol. 10, no. 1/2/3&4, pp. 255–299, 1991.

[19] P. Seshadri, H. Pirahesh, and T. Y. C. Leung, "Complex query decorrelation," in *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*. IEEE Computer Society, 1996, pp. 450–458.

[20] J. Shim, P. Scheuermann, and R. Vingralek, "Dynamic caching of query results for decision support systems," in *SSDBM*, 1999, pp. 254–263.

[21] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan, "Semantic data caching and replacement," in *VLDB*, 1996, pp. 330–341.

[22] L. Chen and E. A. Rundensteiner, "Xquery containment in presence of variable binding dependencies," in *WWW*, 2005, pp. 288–297.

[23] L. Chen, E. Rundensteiner, and S. Wang, "Xcache: A semantic caching system for xml queries," in *ACM SIGMOD*, 2002, pp. 618–618.

[24] L. Chen, S. Wang, and E. A. Rundensteiner, "Replacement strategies for xquery caching systems," in *Data and Knowledge Engineering Journal*, 2004, pp. 145–175.