Optimizing Complex Sequence Pattern Extraction Using Caching

Medhabi Ray [#1], Mo Liu [*#2], Elke Rundensteiner [#3], Daniel J. Dougherty [#4]
Chetan Gupta [*5], Song Wang [*6], Abhay Mehta [*7], Ismail Ari [**8]

[#] *Computer Science Department, Worcester Polytechnic Institute, USA*
[1] medhabi@cs.wpi.edu
[2] liumo@cs.wpi.edu
[3] rundenst@cs.wpi.edu
[4] dd@cs.wpi.edu

[*] *Hewlett Packard Innovation Research Laboratories, USA*
[5] chetan.gupta@hp.com
[6] songw@hp.com
[7] abhay.mehta.gupta@hp.com

[**] *Ozyegin University, Turkey*
[8] Ismail.Ari@ozyegin.edu.tr

***Abstract*— Complex Event Processing (CEP) has become increasingly important for tracking and monitoring complex event 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. 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 our NEsted Event Language *NEEL*. However, the iterative execution of nested CEP expressions often results in the repeated recomputation of the same or similar results for nested subexpressions as the window slides over the event stream. In this work we thus propose to optimize *NEEL* execution performance by caching intermediate results. In particular we design two methods of applying selective caching of intermediate results namely Object Caching and the Interval-Driven Semantic Caching. 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.**

I. Introduction

A. Motivation

Complex Event Processing (CEP) has become increasingly important in many modern business applications, ranging from supply chain management for RFID tracking, click stream analysis for e-advertising 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. vspace-2mm vspace-2mm

```
PATTERN SEQ(Amazon a,
            AND(Dell d, Lenovo l, Apple ap, d.drop>0, ap.drop>0, ap.drop>0)
        a.drop>0)
WITHIN 1 hour
```

Fig. 1. Example Query Q_1

Business enterprizes today rely heavily on market analysis strategies suggested by business analysts. Such analysis is often done by mining unstructured or semi-structured data like stocks and news articles. Consider an example where a business analyst forms a hypothesis that the falling stock price of Amazon results in the falling stock price of some laptop brands. He might want to check his hypothesis against recent stock streams. Consider the query Q_1 shown in Figure 1 will give him all the occasions when the falling price of Amazon has been followed by falling prices of laptop brands. By running similar queries and collecting the statistics, important business patterns can be inferred.

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

Fig. 2. Example Query Q_2

Healthcare is another growing industry requiring monitoring services. Tracking the resources within a large hospital from instruments to patient records is of utmost importance. For example, consider reporting contaminated medical equipments in a hospital [4]. Let us assume that the tools for medical operations are RFID-tagged. The system monitors the histories of the equipment (such as, records of surgical usage, 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". Query Q_2 (Figure 2) 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.

However, the state-of-art CEP systems including SASE [1] and ZStream [3] do not support such nested queries. Even though Cayuga system [2] mentions composable queries, they

assume the negation filter is applied to a single primitive event type within the SEQ pattern. In our recent work, we have proposed the nested CEP language *NEEL* to support nesting of Sequence, AND, OR and Negation queries, along with an iterative nested execution strategy for processing of queries expressed in *NEEL* [5]. Such iterative nested execution while correct can be inefficient. As the query window slides continuously over the event stream the query Window overlaps. Full results satisfying the nested subexpression, such as instances that match the subsequence *AND(Sharpening s, Disinfection d, Checking c)* in Q_2 will be repeatedly constructed and processed. However, in real time monitoring systems relying on CEP queries, performance with reference to time and memory is often critical. Hence recomputation of results should be avoided whenever possible to preserve resources. Thus we study the optimization of nested CEP queries using caching of intermediate results. The problem is challenging as the cache content is under continuous flux. We study what to cache and how to keep the cache up to date as the window slides. We propose two ways of caching intermediate results namely object cache and semantic cache, that both are equipped to support negation and predicate correlation. Our contributions include:

- Design a general approach for buffering the intermediate results in an object cache including cache loading and purging algorithms.
- Propose semantic caching based on installing and matching semantic descriptors as an improvement over object caching including the optimization to maintain membership indicators instead of storing the actual results.
- We implement these alternate solutions in the E-Cube CEP engine [6]
- We experimentally evaluate and compare nested CEP query execution with and without caching on real data streams.

We explain the Nested Event Language (NEEL) [7] and the iterative processing technique for *NEEL* in Section II-A. We introduce the object caching technique in Section III followed by semantic caching in Section IV. Section V presents the experimental evaluation and Section VI discusses related work.

II. Nested CEP Query Processing

A. The Nested Complex Pattern Query Language NEEL

B. Iterative NEEL Execution Strategy

Following the principle of nested query execution for SQL queries [9], [10] an iterative execution strategy for nested CEP queries expressed in the NEEL language (Section II-A) has been designed [5]. The main process is to evaluate the outer query first followed by its inner sub-queries when a query triggering event arrives. The results of the inner queries are passed up and joined with the results of the outer query. For every outer partial query result, a constraint window 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. The iterative execution is continued until a final result sequence is produced by the root operator.

TABLE I
NEEL Query Language

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

C. Processing Nested Queries with Negation

We now describe how to support negation in nested queries. If a query has a negative A between positive B and C event types, we first evaluate the query without the negation, i.e., we compute all <b, c> results. Then for every result generated we 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> results without any B and C events in between them would be returned.

In the nested query model NEEL [7], 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.

SEQ(Recycle r, SEQ(Wash w, Dry dr, Sharpen s), Disinfect d, Operate o)

Fig. 3. Example Query Q_2

D. 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.* 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 3, 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 3 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.

III. Object Caching

The re-computation of the results for inner sub-queries every time an outer triggering event arrives can be rather expensive. We observe that CEP queries work on sliding window over the stream. It is easy to see that many intermediate results for inner subexpressions would continue to be valid from one sliding window to the next. Thus we propose to cache and incrementally maintain the inner query results to serve outer queries from one window to the next.

Example 1: Consider the query Q_3 in Figure 3 over the stream S = $\{r_1, a_2, w_3, c_4, d_5, c_6, o_7, o_8 \ldots\}$. When the event o_7 arrives, it forms the outer query result $< r_1, d_5, o_7>$. The sub-window [1,5] for the inner query is being computed. Then when the event o_8 arrives and the outer result $<r_1, d_5, o_8>$ is formed, the sub-window for the inner query is again [1,5]. The results during this interval would be unnecessarily recomputed again. This is a waste considering all events arrive in order. 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.

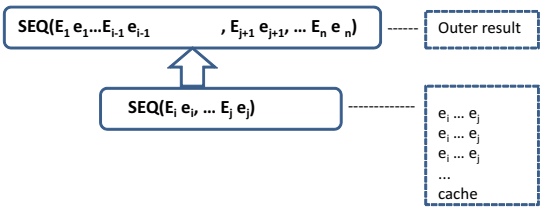


Fig. 4. Cache Design for Object Caching

A. Object Cache Design

We propose to maintain a cache for each sub-query. A cache is a list of result tuples conforming to the intermediate output schema. We also associate a timestamp which we will call the *"rightBound"* with the cache. For the query shown in Figure 4 it is given by e_{j+1}.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. We will henceforth assume that the cache has been loaded with all possible results so far in the input stream up to rightBound.

B. Cache Usage

We calculate the constraint window for each sub-query which we call queryInterval given an outer query result triggered by an event e_n. It is given by the timestamps of the events in the outer query bounding the sub query. For the query in Figure 4, the queryInterval will be given by [e_{i-1}.ts,e_{j+1}.ts] for an outer query result $<e_1 \ldots e_{i-1}, e_{j+1} \ldots e_n>$. We will then check the *"rightBound"* of the cache. If the rightBound of the cache is greater than or equal to the *queryInterval.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 *queryInterval.rightBound* is greater than the rightBound attached to the cache we instead first update the cache as explained below.

C. Cache Maintenance

We extend the cache content when the *queryInterval.rightBound* is greater than the *rightBound* of the cache. That is when a cache is not sufficient (misses results). For all new "triggering" events e_j of the sub-query SEQ($E_i \ldots E_j$) in Figure 4 that have not been previously loaded into the cache namely e_j.ts $>$ *Cache.rightBound* and e_j.ts $<=$ *queryInterval.rightBound*, compute the sub-query results for all e_j between Cache.rightBound and *queryInterval.rightBound*. Then the rightBound of the cache is updated to reflect the present state of the cache namely*Cache.rightBound* = *queryInterval.rightBound*. When a 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 - *window* are also deleted from the cache. The cache content is purged incrementally as the timestamps of the events constituting the results expire out of the sliding window.

Example 2: In Figure 5, we assume that we have stacks for each event type mentioned in the query. As the events arrive we insert them into their respective stacks. The rectangular boxes represent the stacks labeled with the event type it represents. Subscripts denote their timestamp. When the triggering event o_{14} arrives, it is inserted into the Operating stack and triggers execution. [1, 8] is the extracted constraint window for the subexpression SEQ(Washing, Drying, Sharpening). SEQ(Washing, Drying, Sharpening) results are constructed based on all events that occurred during [1,8] and stored in the cache and the *Cache.rightBound* is set to 8. When the new triggering event o_{26} arrives, we obtain 2 results for the outer query namely $< r_1, d_8, o_{26} >$ and $< r_1, d_{15}, o_{26} >$. For the first outer result, sub-query the queryInterval is [1,8]. Hence the existing cache will have all the results while for the next outer result the queryInterval is [1,15]. Since the *queryInterval.rightBound* $>$ *Cache.rightBound*, the cache must be updated. Thereafter the *Cache.rightBound* is updated to 15 to reflect its present state.

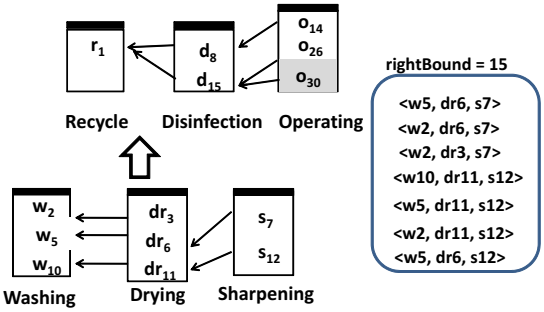


Fig. 5. Object Caching Example

D. Object Caching with Negation

When the inner sub-query is negated, the only difference would be to search for results during the queryInterval and return True or False based on whether any results are found or not during the queryInterval. Consider the query in Figure 5 now with the sub-query SEQ(Washing, Drying, Sharpening) *negated*. We could still reuse the cache. For example for the outer query result $<r_1, d_8, o_{30}>$ the queryInterval is [1,8]. The rightBound at this time is 15. We then search for results during the required interval within the cache. If the cache is empty for the required interval, we output the outer query result and vice versa.

Our object caching technique can be extended to support equality predicate correlations by partitioning the cache by the different values of the predicate seen so far and maintaining a rightBound for each partition.

IV. Semantic Caching based on Interval

There are several disadvantages of the above object caching. One, since the cache stores all results in one list we need to search through the list to find the results within the given queryInterval. Consider the example where the cache rightBound has been extended to 100 while we are looking for results between the queryInterval [5,10]. In this case a full scan of the entire list of results. Hence reducing this search space could result in cost savings. Furthermore space wasted for storing the results of the negative sub-queries even though they are not actually joined to form results. All we need is the information whether a certain queryInterval is empty or not. Hence storing all the results in the object cache is unnecessary. Lastly, we compute and store cache results for the full range of the stream up to *rightBound* even though some sub-ranges may never get utilized.

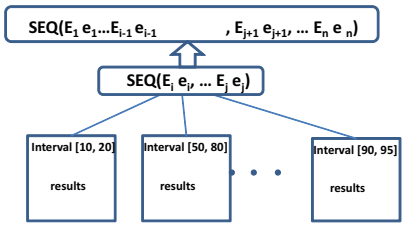


Fig. 6. Cache Design for Semantic Caching

A. Semantic Cache Design

To overcome the above shortcomings we introduce the concept of semantic descriptors in the form of *cacheInterval*. cacheInterval denotes the time interval for which a certain cache is guaranteed to contain all the results for. A given sub-query we maintain a list of cacheIntervals and the results associated with the respective interval. For overlapping intervals the results common to both the intervals could be stored in a common storage. In this case the caches store a reference to the common sub-range.

B. Semantic Cache Usage

Instead of having to scan the raw object cache for matches, we can now match the meta descriptors, the cacheIntervals against the queryInterval, to facilitate efficient access. Once

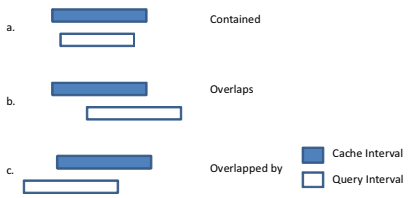


Fig. 7. Types of overlap between cacheInterval and queryInterval

the most appropriate cacheInterval has been identified, the required results can be efficiently returned. Several types of overlaps as depicted in Figure 7 can arise. Based on the type of overlap we utilize the cache differently. We will consider them in the following order:

- **Exact Overlap or Contained.** In the first type of overlap in Figure 7(a). The queryInterval is completely contained in one cacheInterval. Hence all possible results for that queryInterval are guaranteed to be present in the cache. Hence we simple access the results associated with this cache region without any need of cache updating. Once we find such an interval we stop the search.
- **queryInterval overlaps cacheInterval.** The second type of overlap is shown in Figure 7(b). In such a scenario we can reuse the results in this cache but will also have to compute additional results for the non-overlapped region. Hence in this case we will do partial computation. We will form a new cacheInterval once we have collected all the results for this new interval. If we find such an interval we will still search the remainder of the meta-descriptors to check if we can find an overlap of the first type because we prefer an Exact or Contained type of match.
- **queryInterval overlapped by cacheInterval.** The third type of overlap in Figure 7(c). For this type of overlap we will compute the results for the required interval even though some of the results might have already been computed before. For example if we are looking for SEQ(B, C) and the cacheInterval is [5,10] while the queryInterval is [1,7]. This range can contain a result $< b_2, c_8 >$ If we compute the results for triggering events occurring only from [1,5], we will not obtain $< b_2, c_8 >$ which is also a valid result for the given queryInterval. Hence we will compute the sub-query for the entire queryInterval, without any atempt for partial recomputation.
- **Non Overlap.** In all other cases the new sub-query is computed for the given Interval. The cacheInterval is then added to the list.

C. Semantic Cache Maintenance

When the queryInterval does not overlap any cacheInterval, 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 queryInterval. If a queryInterval has no overlap with an existing cacheInterval, the sub-query needs to be computed from scratch. If a part of the queryInterval overlaps a cacheInterval as in the case in Figure 7(b) we compute the sub-query for triggering events that occur only in the non-overlapping part.

Instead of a linear scan of the results, we could in practice sort the results on the leftBound of the cacheIntervals, or maintain an index over the list of intervals to further expedite the search.

Example 3: For the query in Figure 3, when the tuple e_{14} arrives, the query is triggered and the outer result $<r_1,d_8,o_{14}>$ is formed. The results of the sub-query SEQ(Washing w, Drying dr, Sharpening s) are computed and stored in the cache with cacheInterval [1,8]. When the tuple e_{26} arrives, the queryInterval extracted is [1,15]. The cacheInterval [1,8] is reused. The sub-query is also computed for the triggering events in the non-overlapping region of the queryInterval and stored in the cacheInterval [1,15] which also maintains a reference to the cache [1,8]. Now when e_{30} arrives and the queryInterval [1,8] is extracted for the result $<r_1,d_8,o_{30}>$, it can easily look up the results only for the interval [1,8] instead of scanning the results for the large interval of [1,15] as in the previous method.

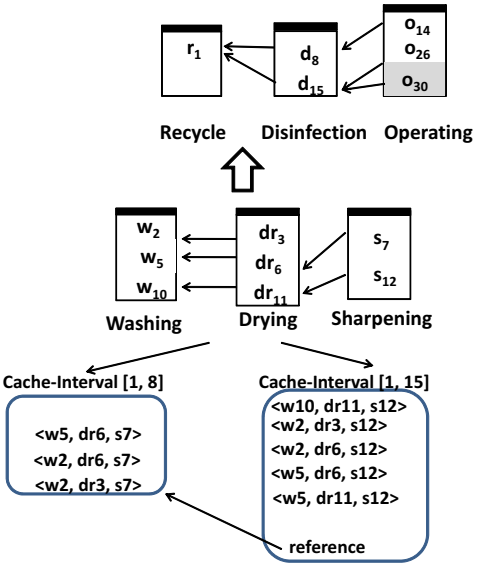


Fig. 8. Semantic Caching showing multiple caches for a sub query for query shown in Figure 3

D. Semantic Caching for Negative 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 the negative sub-queries. Instead simply storing an "isEmpty" flag for a given interval is sufficient. Thus we check the isEmpty flag for a given queryInterval and filter out the results if the isEmpty flag is false.

SEQ(YHOO y, SEQ(QQQ q, AMAT a, DELL d), ORCL o, IPIX i)

Fig. 9. Example Query Q_3

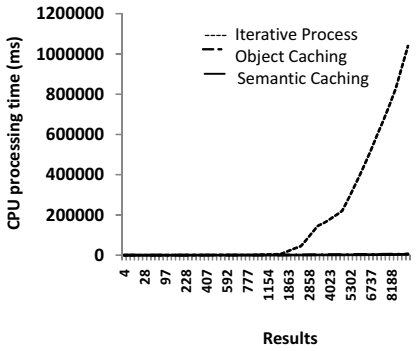


Fig. 10. Comparing CPU costs of Iterative approach and Semantic and Object Caching

V. Performance Evaluation

A. Experimental Setup

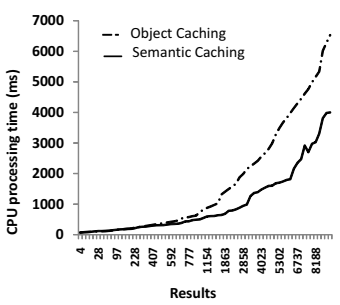
The following experiments use the query shown in Figure 9. The experiments compare the CPU processing time of the various approaches described. CPU processing time means the wall clock time for processing an item e_i in stock trades measured by ($T_{end.ei}$ - $T_{start.ei}$) where $T_{start.ei}$ represents the system time when our processing engine starts processing of the data item e_i and $T_{end.ei}$ represents the system time when the engine finishes processing the data item e_i. It is an atomic process, i.e., our processing engine won't stop processing that tuple until it is fully processed.

We implemented our proposed caching strategies inside the complex event processor called ECube [6] using Java and ran the experiments 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 using stacks. In a nested query the processing of each subexpression follows the same strategy. The data contains stock ticker, timestamp and price information [11]. The portion of the trace we used has 10,000 unique event instances.

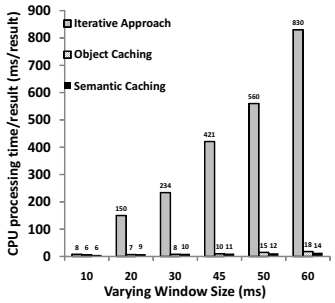
B. Evaluation Results

The chart in Figure 11(b) compares the CPU processing time for the query shown in Figure 9 needed by the Iterative Execution, Object Caching and Semantic Caching. The Object and Semantic Caching perform many orders of magnitude better than the Iterative approach. The chart in Figure 11(b) has a Window Size of 50ms. The chart in Figure 11 (a) zooms into the previous chart to highlight the performance difference of Object vs Semantic Caching. Semantic caching performs better than Object Caching.

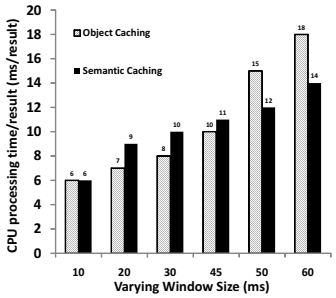
The charts in Figure 11 (b) and (c) show the effect of varying Window Sizes on the performance of the three techniques. As we would expect the average processing time increases as the Window Size in increased. While the two caching methodologies both win over the Iterative approach, the performance benefits of Object vs Semantic Caching varies with the Window Sizes. For small Window Sizes, Object Caching seems to win, while for larger ones the Semantic Caching does better. This is intuitive, because when the Window Size



(a) Comparing CPU cost of Semantic and Object Caching



(b) Comparing average CPU cost per result of three techniques varying Window Size



(c) Comparing average CPU cost per result of Semantic and Object Caching varying Window Size

Fig. 11. Experimental Results

is very small, the cost for searching results in a list is almost similar to the cost of searching through the cacheIntervals. But as the Window Size increases, the number of results increase much more compared to the number of cacheIntervals formed. Hence searching for the right Cache Interval takes a shorter time.

VI. Related Work

The existing CEP systems [1], [2], [3], [8] do not typically focus on the execution of nested sequence queries as tackled in this paper. The query language of the CEDR [8] system supports nested sequence queries. However, the execution strategy for such nested queries is not discussed. [5] has shown a correct but naive strategy to evaluate nested complex event queries.

It is usually inefficient to directly execute a nested query; consequently, algorithms such as magic decorrelation [12] and complex query decorrelation [9], [13] have been proposed to decorrelate the query. However, these existing decorrelation algorithms deal with static relational queries and are neither described nor tested in the streaming context.

For SQL queries, [14] 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 [15], a semantic description of the data in a cache is maintained which allows for a compact specification. We study caching inner queries in the streaming context and apply interval driven caching by using validity intervals as semantic descriptors. Semantic descriptors have also be shown to be of importance for query caching in the XML context [16], [17]. However, sophisticated cache matching algorithms had to be designed to deal with query containment, namely, with extracting related yet not identical subexpressions possibly with alternate hierarchical XML structures yet the same content [16].

VII. Conclusion

In this paper, we propose to optimize *NEEL* execution by caching intermediate results. In particular, we design two methods of applying selective caching, namely Object Caching and Interval-Driven Semantic Caching. We described techniques for incrementally loading, purging and exploiting the cache content. Lastly, an optimization technique for negated sub-queries is introduced. Our preliminary experimental results clearly demonstrate the performance benefits of the caching techniques.

VIII. Acknowledgements

This work is supported by HP Labs Innovation Research Program and National Science Foundation under grants NSF 1018443 and NSF IIS 0917017, Turkish foundation under TUBITAK career award 109E194. We also thank Database System Research Group members at WPI for their valuable comments.

References

[1] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams." in *SIGMOD Conference*, 2006, pp. 407–418.
[2] Alan J. Demers et al., "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] Mo Liu and Medhabi Ray and Elke Rundensteiner and Chetan Gupta and Song Wang and Ismail Ari and Daniel J. Dougherty, "Processing nested complex sequence pattern queries over event streams," in *DMSN, VLDB Workshop*, September, 2010.
[6] Mo Liu et al., "E-Cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing," Worcester Polytechnic Institute, Technical Report WPI-CS-TR-09-08, 2009.
[7] M. Liu, E. Rundensteiner, and D. J. Dougherty, "Nested complex event processing for real-time event analytics," in *BIRTE*, 2010, pp. 358–369.
[8] 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.
[9] Praveen Seshadri et al., "Complex query decorrelation," in *ICDE*, 1996, pp. 450–458.
[10] E. Wong and K. Youssefi, "Decomposition - a strategy for query processing," *ACM Trans. Database Syst.*, vol. 1, no. 3, pp. 223–241, 1976.
[11] "I. inetats. stock trade traces. http://www.inetats.com/."
[12] C. Beeri and R. Ramakrishnan, "On the power of magic," *J. Log. Program.*, vol. 10, no. 1/2/3&4, pp. 255–299, 1991.
[13] W. Kim, "On optimizing an sql-like nested query," *TODS*, vol. 7, no. 3, pp. 443–469, 1982.
[14] Junho Shim et al., "Dynamic caching of query results for decision support systems," in *SSDBM*, 1999, pp. 254–263.
[15] Shaul Dar et al., "Semantic data caching and replacement," in *VLDB*, 1996, pp. 330–341.
[16] Li Chen et al., "Xquery containment in presence of variable binding dependencies," in *WWW*, 2005, pp. 288–297.
[17] Li Chen Elke Rundensteiner and Song Wang, "Xcache: a semantic caching system for xml queries," in *SIGMOD Conference*, 2002, p. 618.