High-Performance Complex Event Processing using Continuous Sliding Views

Medhabi Ray, Elke A. Rundensteiner, Mo Liu, Chetan Gupta¶, Song Wang¶, Ismail Ari‡
Worcester Polytechnic Institute, USA, ¶HP Labs, USA, ‡Ozyegin University, Turkey
(medhabi|rundenst|liumo)@cs.wpi.edu
¶(chetan.gupta|songw)@hp.com
‡ismail.ari@ozyegin.edu.tr

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) *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. 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. The first is the Continuous Sliding Caching technique. The second is a further optimization of the previous technique which we call 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.

1. INTRODUCTION

1.1 Motivation

∽ Complex event processing (CEP) has recently gained importance due to its wide range of applicability in modern applications, ranging from RFID based inventory management to real-time intrusion detection. To fully harness the power of event processing, languages for specifying powerful pattern matching queries over event streams with constructs for expressing sequencing, negation, and complex predicates have been proposed [1, 2, 3]. In particular, nested CEP query languages [4] provide users with flexible nesting of operators. Without this capability, users are severely restricted in forming complex patterns in a convenient and succinct manner. Often, some complex requirements cannot even be expressed as flat queries. For instance consider the nested CEP query written using *NEEL* syntax [4] shown in Figure 1. The query consists of three

```
SEQ(Recycle r, Wash w,
        !AND(Sharpen s, Disinfect d ,s.roomID=d.roomID,
        s.id = r.id, d.id =r.id),
 Operate o, r.id = o.id, r.id =w.id)
 WITHIN 10 minutes
```

Figure 1: Example Query Q_1

levels of nested operators. The outermost "SEQ" specifies an ordered sequence of "Recycle, Wash and Operate" events. The inner "AND" specifies an unordered collection of "Sharpen" and "Disinfect" events occurring in the event stream between the "Wash" and "Operate" events. The negation operator "!" signifies that "Sharpen and Disinfect" events both must not occur in either order in the stream between the matched "Wash" and "Operate" events. This particular query detects the faulty condition that after being recycled and washed, a surgery tool is being put back into use for operation without first being sharpened and disinfected in a same quality assurance room. Several such queries may be registered together in the Quality Assurance system to detect violations. In the

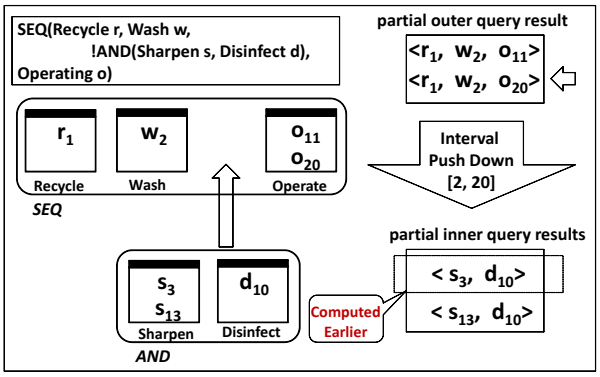


Figure 2: Iterative Processing of nested CEP Query

absence of a customized approach of processing such queries, the principle of nested SQL query execution [5] is adopted [6]. That is the outer query is evaluated first followed by its inner sub-queries. Consider figure 2 where the hierarchical stack based CEP operators implementing each of the sub-queries of Q_1 has been depicted. The subscripted letters denote event instances and the subscripts are the timestamps. The outermost SEQ(Recycle, Washing, Operating) query is triggered by the arrival of an "Operate" event and the partial result <r_1,w_2, o_{11}> is produced. A tighter bound of [2,11] (given by the timestamps of w_2 and o_{11}) is pushed down

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy.

to the inner !AND(Sharpening, Disinfection) query which in turn produces the result set <s_3,d_{10}> within the specified interval. The predicate testing is done at this stage to eliminate potential match candidates. Based on the results returned, the"!" operator filters out the result <r_1,w_2,o_{11}>. For the next outer result <r_1,w_2,o_{20}>, the constrained window is [2,20]. The default method would now compute the inner query **once more** and again the result set <s_3,d_{10}>, <$s_1 3$,d_{10}> filters out the outer result. However, a part of the computation for the inner sub-query is repeated. This is wasteful of critical computing resources and not efficient in handling high volume data rates. Figure 3 shows the application of rewriting rules on the example query Q_1. It shows how the number of flat queries grows exponentially with the number of events inside the AND operator. Moreover, this rewriting is also incomplete and does not capture the true semantics of the original query. The last two flat queries in figure 3, do not have the predicates that specify that sharpening and disinfection of the tools must occur in the same room. The reason is that specifying predicate correlation between non-existent events is syntactically and semantically wrong.

SEQ(Recycle r, Wash w, !(Sharpen s, s.roomID=d.roomID, s.id = r.id), Disinfect d, Operate o, r.id = o.id, r.id =w.id , d.id =r.id)

SEQ(Recycle r, Wash w, Sharpen s, !(Disinfect d, s.roomID=d.roomID, d.id = r.id), Operate o, r.id = o.id, r.id =w.id , s.id =r.id)

SEQ(Recycle r, Wash w, !(Disinfect d, s.roomID=d.roomID, d.id = r.id), Sharpen s, Operate o, r.id = o.id, r.id =w.id , s.id =r.id)

SEQ(Recycle r, Wash w, !(Disinfect d, d.id = r.id), !(Sharpen s, s.id = r.id) Operate o, r.id = o.id, r.id =w.id)

SEQ(Recycle r, Wash w, !(Sharpen s, s.id =r.id), !(Disinfect d, d.id = r.id), Operate o, r.id = o.id, r.id =w.id)

Figure 3: Incompletely rewritten Q_1

1.2 State of the Art

While most state-of-the-art CEP systems including [1, 3, 7, 2] support the specification of CEP queries with sequencing, negation and predicates some [4] also allow the flexible nesting of negation with other CEP operators into nested CEP expressions as shown in the example query Q_1. However there is a lack of effective techniques in the CEP literature to support efficient execution of powerful queries such as Q_1. While an optimization technique involving rewriting of nested queries was proposed in [8] it is not generally applicable as shown by our example above-. In short, existing state-of-the-art CEP solutions do not have an efficient way of handling nested queries.

On the other hand, nested processing is a well studied area in the SQL context. Mainly two directions have been explored. Approaches for decorrelating have been proposed in [9] to merge nested queries by rewriting them. Orthogonally, Semijoin like techniques involving materializing intermediate results in views [10, 11] have been studied for optimizing multi-block nested queries. In [12] where incremental views were created over nested data models to efficiently process queries on such data models. In each case the goal was to avoid redundant computation in the nested sub-queries. Although, these techniques worked well in the static context, the problem is different in the streaming context, where indexes are generally found to be expensive and views become outdated very frequently. Hence light-weight semantics are needed to ensure view completeness. We will now borrow the idea of attempting to avoid redundant sub-expression computation from the literature of static database by adopting and adapt them to the streaming environment.

1.3 Proposed Approach

Since nested CEP queries are widely exploited in real-time applications which need prompt responses efficient processing strategies are extremely important. Repeated re-computations like the ones shown in Figure 2 waste critical CPU cycles and must be avoided at all costs. Thus we now propose to tackle this challenge by introducing the notion of "Continuous Sliding Views" of the inner queries over stream. We call these Continuous Sliding Views because they are views created for continuous sub-queries that need to slide over the event stream in lock-step with the outer query expressions of the nested query.

We store results of sub-queries and incrementally update and re-use them for answering continuous queries over sliding windows. Given streams are potentially unbounded, CEP queries are equipped with temporal windows that semantically extract a sub stream out of the infinite stream. The query is then run on this subset of the stream. As example the CEP query Q_1 works on a sliding window of "10 min". The window advances as new events arrive - "Slide by tuple" or as time passes "Slide by time". In either case there is generally an overlap of events between consecutive window slides. Thus if results for a CEP query are cached in continuous sliding views, a large part of the required result set may be already present in these views for subsequent query invocation on the next window slide. This raises the challenge that these continuous views must be maintained because the data is under continuous flux yet memory is limited. We need to continuously update the content of the views by adding new intermediate results and removing expired ones. In addition, we need to carefully design a light-weight indexing structure to update and access the stored view content to assure the needed efficiency. In this work we thus propose to tackle these challenges by developing effective continuous sliding view optimization strategies.

1.4 Contributions

Our contributions can be summarized in the following points.

- Designed an aggressive pre-fetch based approach called "Aggressive Continuous Sliding Views" (A-CSV) for maintaining and reusing intermediate results of sub-queries to speed up nested CEP execution over streams.
- To avoid overly aggressive pre-fetching of sub-query results which may potentially lead to over-computations in some cases, we design the "Passive Continuous Sliding Views" (P-CSV) by attaching temporal intervals.
- Proposed a hybrid approach called "Balanced Continuous Sliding Views" (B-CSV) that takes a clever middle-ground approach by borrowing principles from the above two mechanisms.
- Developed and analyzed a cost model for a comparative study of the alternative view maintenance strategies.
- Implemented the above CSV strategies in the *NEEL* query execution engine [8].
- Conducted experimental studies comparing the nested CEP query execution with and without the optimizations and the state-of-the-art nested CEP processing technique using rewriting of nested CEP queries [13] and achieved upto 89 percent speedup in execution time.

While we use the Nested Event Language proposed by Liu. et al in [4] to specify our queries, our approach is generally applicable in any nested CEP queries. Our proposed approach is simple and yet effective. Most importantly they are applicable to any nested

CEP queries and achieve significant gain over existing techniques for processing nested CEP queries.

2. RELATED WORK

CEP Systems: Most state-of-the-art CEP systems (such as SASE [1] and ZStream [3]) do not support nested pattern queries over streams. In Cayuga [2] the Cayuga Event Language allows the specification of sub-queries in the form of an SQL-like algebra [9]. However it doesn't support applying negation over composite event types. While CEDR [14] allows applying negation over composite event types within their proposed language, the execution strategy for such nested queries is not discussed. [4] proposed the language *NEEL* to specify nested CEP queries. Similarly in [15], an XML based language was proposed which had the capability of supporting nested Sequence queries, however it did not have negation and it did not focus on the processing paradigm of these nested queries. In short, processing and optimization of nested CEP queries need greater attention to achieve much more optimized realtime performance. Rewriting based approach proposed in [8] optimized performance in some cases but they were not applicable for a wide range of queries.

View Materialization: Views have been used in database systems for primarily two purposes. Firstly views provided logical and physical data independence. Secondly computing a query using previously materialized views can speed up query processing because part of the computation necessary for the query may have already been done while computing the views. The idea of answering queries using views has existed for a long time.

[16] proposed the idea of precomputing views over static data which can be used by other queries. Algorithm design for deciding the minimum set of views to materialize for answering a query has drawn a lot of attention [17]. In [18] the idea of intermediate result caching was introduced for sequence queries in static database systems. There have also been work on query optimizations by rewriting queries in order to reuse views. Answering queries using views plays a key role in developing methods for semantic data caching in client-server systems [19], [20]. In these works, the data cached at the client is modeled semantically as a set of queries, rather than at the physical level as a set of data pages or tuples. Hence, deciding which data needs to be shipped from the server in order to answer a given query requires an analysis of which parts of the query can be answered by the cached views.

In [12] incremental views were proposed over nested data model [12] to decrease the number of disk I/Os while answering queries. However, in these systems the data is static with infrequent insertions and deletions. On the other hand, nested CEP queries run on continuously arriving and expiring events. While they focussed only on incremental update of views, we propose a more general integrated approach for maintaining and exploiting views for continuous CEP queries.

Decorrelation: Kim [9] and subsequently others [21, 22] have identified techniques to unnest a correlated nested SQL query and ŞflattenŤ it to a single query. Cao and Badia [21] processed nested queries independently and then joined the results from different levels by the "correlated" predicates. Consequently, algorithms such as "complex query decorrelation" [10] have been proposed to decorrelate the query. However, existing decorrelation algorithms for relational databases are inherently different from CEP queries. Along the lines of nested query decorrelation, Liu et al in [8] developed a set of equivalence rules for rewriting nested *NEEL* CEP expressions into unnested form for faster execution. However not all nested queries can be rewritten into unnested forms. Also their rewriting rules were often limited by the presence of predicate correlations. Continuous SLiding Views on the other hand is a general principle that is scalable and can be applied to any nested CEP query. Our discussion of our core techniques and experiments will further prove this claim.

3. PRINCIPLES OF CONTINUOUS SLIDING VIEWS

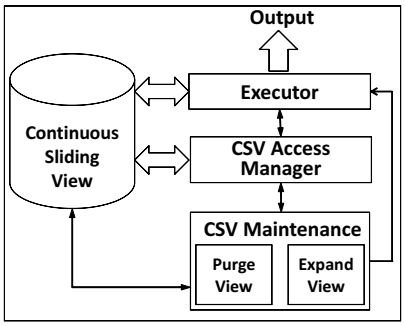


Figure 4: CSV System Architecture

Our proposed "Continuous Sliding Views" are cached results for inner CEP sub-queries. As continuous queries run over sliding windows, result sets are produced for each sub-query. These results are cached in memory to be re-used in subsequent window slides. Thus a Continuous View is formed for each sub-query for holding its respective result tuples. As the window advances, events in the data stream expire out and so do result tuples from the views. The view is continuously updated as new results are produced from the incoming event stream. However, to guarantee correctness and completeness of results, we will need to attach descriptors for these Continuous Sliding Views to denote their validity scope.

Figure 4 shows the overall system architecture of the nested query processing system with intermediate result storage. The single-lined arrows show the direction of control flow while thick arrows show the flow of data. The Executor conducts the actual query processing by controlling the functionality of query operators, such as AND, SEQ and "!" operators which produce pattern query results. The stream is input into this module and the complex event patterns it produces are stored in the views. The View Access module checks if the results required at a given time are present in the Sliding View or not. It may use the view directly if it contains the required results. Otherwise it would invoke the CSV Maintenance module to update the view. This maintenance could be in the form of inserts as well as deletes from the continuous sliding view. It inserts more tuples by calling the Executor module and purges the cached results in the view as the window slides over. In the following sections we will introduce three alternate but closely related nested CEP optimization techniques. Each of these strategies share three common phases - Loading, Usage and Maintenance (Purging and Expansion).

In order to describe our technique in greater detail, we will introduce some general terms that will be used in the following discussions repeatedly.

DEFINITION 1. ***Query-Interval:*** *It is an ordered pair of timestamps (leftbound and rightbound) for which a given query is computed.*

- *For a root query, the Query-Interval is given by [$e.time - stamp$, $e.timestamp$ - query.Window] where e is the last event that has arrived.*

- *For SEQ(E_1,..E_i,(subquery q_i),E_j,..E_n) Query-Interval of q_i is given by [e_i.timestamp, e_j.timestamp] where <e_1, .., e_i, e_j, .., e_n> is a valid result of the query SEQ(E_1,. .E_i,E_j,.. E_n) and E_i and E_j are primitive positive event types.*
- *For AND(E_1,..E_i,(subquery q_i),E_j,..E_n) Query-Interval of q_i is given by [e_1.timestamp, e_n.timestamp]where <e_1, .., e_i, e_j, .., e_n> is a valid result of the query SEQ(E_1,..E_i,E_j,.. E_n) and E_1 and E_n are primitive positive event types.*
- *If there are no primitive positive event types adjacent to a nested sub-expression, the Query-Interval is inherited from its parent sub-expression without further restriction.*

4. AGGRESSIVE CONTINUOUS SLIDING VIEW

Our first CSV strategy is that of aggressively pre-computing results for all sub-queries upto a certain timestamp, so that at any time, when the outer query looks up the views before that time, all results are guaranteed to be present in the view. We call this approach Aggressive Continuous Sliding View or A-CSV.

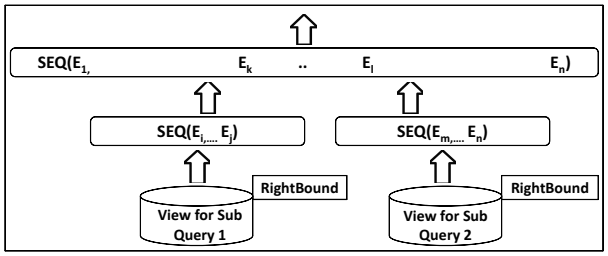


Figure 5: Aggressive Continuous Sliding View Design

We maintain an A-CSV structure for each sub-query. An A-CSV corresponds to a list of result tuples conforming to the intermediate output schema. We associate an indicator with each view called the *"rightBound"* which indicates the timestamp until where the view has progressed and contains all valid results for that sub-query. For the query shown in Figure 5 it is given by e_{j+1}.ts with e_{j+1} the event with the maximum timestamp among all events of type E_{j+1} which have arrived so far and for which the view has been computed. In the continuous view maintaining solution the view will be loaded with all potential candidate results so far in the input stream up to the rightBound.

4.1 A-CSV Loading

When a new nested CEP query starts running, an A-CSV structure is created for every sub-query. This structure is a lightweight list of result tuples whose size grows and shrinks dynamically as the query runs over incoming event streams. As new results are computed for a sub-query, the"View.rightBound" is given by the largest "Query-Interval.rightbound" that has been processed so far by that query. A-CSV maintenance in section 4.3 describes in detail how the *"rightBound"* is computed.

4.2 A-CSV Usage

As events stream in and result construction is triggered, new results are computed for the outer query. Here we emphasize that for the outer-most query, which is triggered by the arrival of new triggering events, there is never a chance of recomputing old results. Given an outer query result triggered by an event e_n, we calculate the constraint window, Query-Interval for each sub-query. If the rightBound of the view is greater than or equal to the *Query-Interval.rightBound* it means that the existing view contains all the required results. Hence a scan through the view will give us the required results. Clearly not all results in the view 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 view we have to instead first update the view as explained below before we can extract the desired result.

4.3 A-CSV Maintenance

View Update. When the *Query-Interval.rightBound* is greater than the *rightBound* of the view, it means that the view might not contain all possible results that the current Query-Interval needs. That is the view must be expanded. For all new "triggering" events e_j for the sub-query SEQ($E_i \ldots E_j$) in Figure 5 the results of the sub-query do not exist in the current view. We thus compute this sub-query for all triggering events e_j such that e_j.ts > *View.rightBound* and e_j.ts <= *Query-Interval.*
rightBound and load them into the view. Then the rightBound of the view is updated to reflect the present state of the view namely *View.rightBound* := *Query-Interval.rightBound*.

Purge. When an outer query triggering event e_n arrives, events with timestamp less than (e_n.ts - *window*) are purged from their stacks. Similarly, results in the views involving events with timestamp less than (e_n.ts - *window*) are also deleted from the view. This window is the overall query window specified in the query.

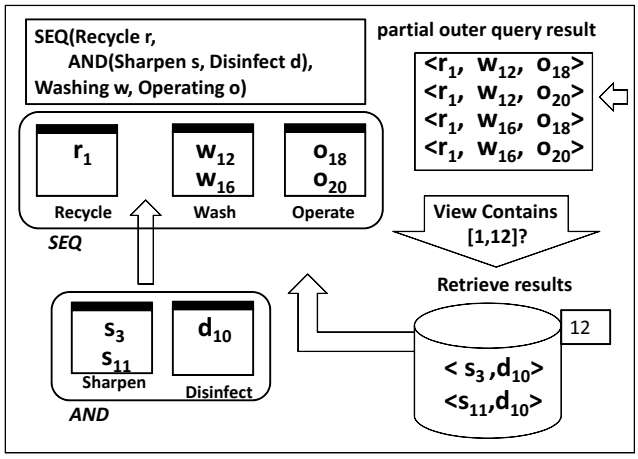


Figure 6: A-CSV Usage Example

EXAMPLE 1. *Figure 6 shows how on the arrival of event o_{20}, Sequence Construction by joining the events of the corresponding stacks is triggered. The interval is extracted for the result < r_1, w_{12}, o_{18}>, namely [1, 12]. AND(Sharpen, Disinfect) results are constructed for interval [1,12] and stored in the view. Lastly, the View.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_{20} > and < r_1, w_{16}, o_{20} >. For the first outer result, the Query-Interval for the sub-query is again [1,12]. Hence the existing view will have all the results. However for the next outer result, the Query-Interval is [1,16] as shown in Figure 7. Since the Query-Interval.rightBound > View.rightBound, the view must be first updated. Thereafter the View.rightBound is updated to 16 to reflect its present state.*

4.4 A-CSV for Negated Sub-queries

Continuous Sliding View for Negative Sub-Queries is essentially the same as described above. When the inner sub-query is a boolean

expression (i.e., if a negation qualifier exists in front of the subexpression), then the only difference would be to search for results during the Query-Interval. We return True or False based on whether any results are found or not during the given Query-Interval.

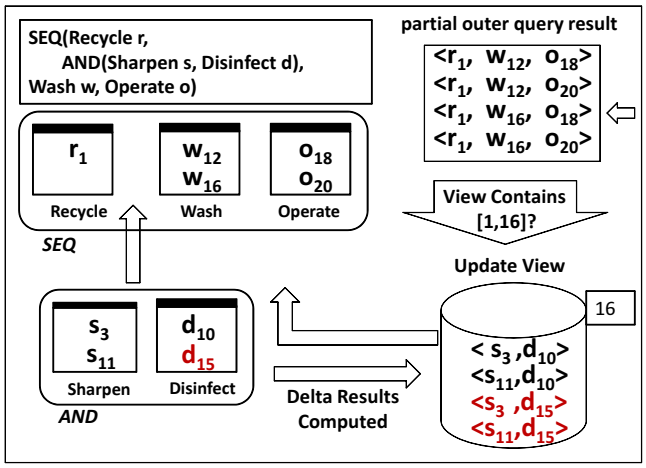


Figure 7: A-CSV Update Example

EXAMPLE 2. *Consider again the query in the example in Figure 6 with a slight change. Let the sub query now be negated i.e SEQ(Recycle r, !AND(Sharpen, Disinfect), Washing w, Operating o). In this case we could still reuse the view. 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 view. If the view is empty for the required interval, we will output the outer query result and vice versa.*

4.5 A-CSV with Predicate Correlation

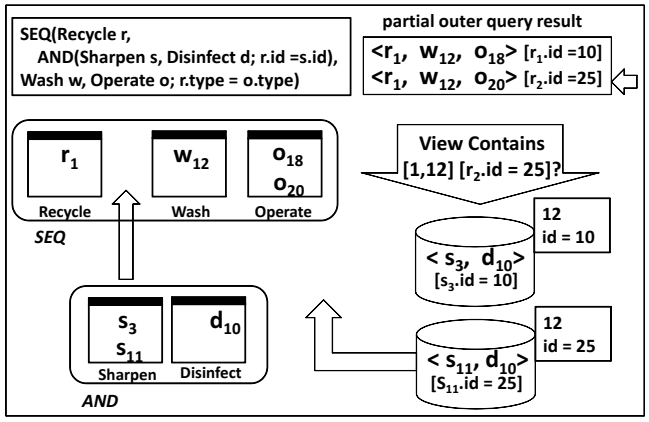


Figure 8: A-CSV with Predicate Correlation

Our view structure is extended to support efficient predicate correlations. We borrow from the literature, techniques for value based partitioning the views for efficient retrieval of results. The processing is very similar to ones explained above. However, when there is equality predicate correlations between outer and inner sub-queries, we will further partition the view by the different values of the predicate seen so far and maintaining a rightBound for each partition. Thus for the query in Figure 8, we will have multiple partitions of the view namely one for each different value of "id" pushed down from the outer Recycle event type.

EXAMPLE 3. *Consider the example shown in Figure 8. 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 view partition with predicate value 10. For the outer query result* $<r_2, w_{12}, o_{18}>$ *where* $r_2.id = 25$ *we search for view partitions with "id" = 25. It does not exist and is hence computed and stored in a partition annotated with predicate value = 25. Similarly view updates will be done to respective partitions.*

4.6 Preliminary Performance Evaluation

Experimental Setup: The Continuous Sliding Views have been implemented inside the stream management system called ECube [13] using Java. Experiments were run on Intel Pentium IV CPU 2.8 GHz with 4GB RAM running Microsoft Windows 7 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. The data contains stock ticker and timestamp information [23]. The portion of the trace used has 10,000 unique event instances. In most of our experiments the cumulative execution time has been recorded on the Y-axis against the cumulative number of results.

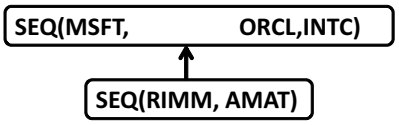


Figure 9: Nested Query Used For Experiments

Evaluation of results: The query depicted in Figure 9 shows a nested query which is run for three different Window sizes of 100, 500 and 1000 seconds using the continuous view maintenance technique and the un-optimized iterative processing technique. Figures 10 a, b and c demonstrate the effectiveness of our view maintenance techniques over the un-optimized iterative technique with increasing Window sizes from 100, 500 to 1000. Our discussions about cost analysis in Section 7 will give an insight into how the view maintenance solution reduces the execution time due to the overlap of results between consecutive window slides. By increasing the size of the Window, the overlap increases and hence the potential gain of the view maintenance based solution. Thus the our solution produces results 6, 9 and 16 times faster than the default processing technique for the increasing Window sizes.

4.7 Discussion of Aggressive Continuous Sliding View

The Continuous Sliding View provides about 89 percent improvement over the iterative processing technique. It avoids the re- computation of intermediate results thus leading to a much faster response time of a given nested query. However, to ensure correctness, this strategy of keeping the view complete upto a certain timestamp, is wasteful of memory as it often leads to computing and storing many result tuples that are never used. The following example proves the above claim.

EXAMPLE 4. *Consider the scenario shown in Figure 11. The outer query produces a result* $<r_{20}, w_{26}, o_{30}>$*. The Query-Interval extracted is thus [20,26]. For the sake of correctness of this technique the view needs to be updated up to timestamp 26. Hence all results between [16,26] are computed for the sub-query AND(Sharpen s, Disinfect d). This may lead to computing many results which*

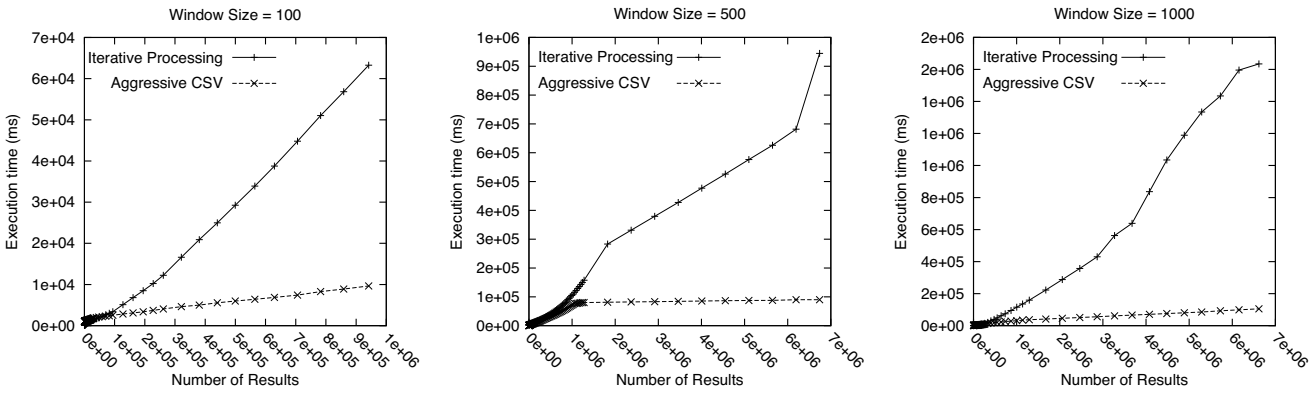


Figure 10: Comparing Execution Time of Queries using Iterative Technique and Aggressive Continuous Sliding Views with Varying Window Sizes

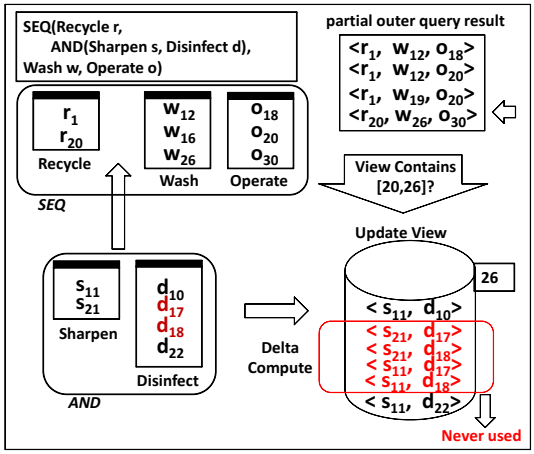


Figure 11: Over-Computation in A-CSV

never get joined to any outer query result. For example the result instances shown in red never get used.

The analysis in Example 4 reveals that our Aggressive CSV suffers from several critical shortcomings, namely:

- This technique might force us to pre-compute many results that are never used by any outer sub-query. Example 4 shows the several 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 negative sub-queries 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.

5. PASSIVE CONTINUOUS SLIDING VIEW

The main disadvantage of maintaining a continuous view is the need to ensure completeness. However, this may result in computing possibly huge set of results which get computed only for the sake of completeness of the view yet may never get used. A parallel can be drawn to "blind pre-fetching of results" instead of a re-use based methodology where results that have been used previously are stored due to the likelihood of them being re-used in the future. To overcome this disadvantage, we now propose to enhance the view design by adding a semantic descriptor that effectively indices the view content. Effectively we will maintain discontinuous pieces of the view instead of a continuous view. Figure 12 shows an example of how the views are related to a given query for Passive Continuous Sliding Viewing (I-CSV).

Here we now propose to partition the view 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 at the smaller meta-data level.

5.1 Passive Continuous Sliding View Design

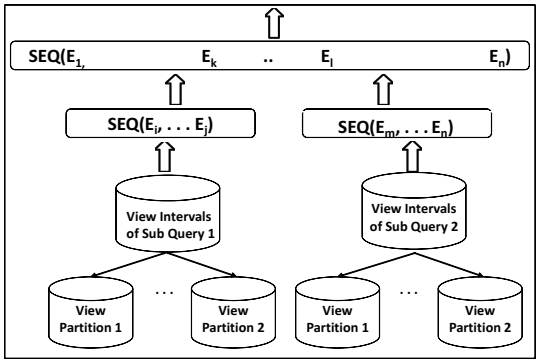


Figure 12: Passive Continuous Sliding View Design

The semantic descriptors attached to the view partitions will be an extension of the viewBound concept used in previous sections. We will refer to these descriptors as **View Interval**.

DEFINITION 2. ***View Interval*** *is an ordered pair of time stamps attached to view during which some or all results of a query is present.*

It denotes the time interval for which a view is guaranteed to contain all possible results. For a given sub-query we will maintain a list of View-Intervals and the results associated with the respective interval.

5.2 P-CSV Loading

When a query starts running, a view is created for every sub-query. As results for a sub-query are computed, they are stored in their respective views. The View-Interval defined above is also attached to the view at this time.

5.3 P-CSV Usage

In contrast to the Continuous Sliding View, the Passive Continuous Sliding View allows an efficient access to the view content by matching the meta descriptor - View-Intervals. Once the matching View-Intervals have been identified the required results can be efficiently returned. The extracted Query-Interval is looked up against the list of View-Intervals. If there exists a match a direct reuse is made without having to update it. This method is a significant improvement over the previous view maintaining technique where there was a need for a scan over the entire result set with a very small fraction of it catering to the real reuse. Thus this acts as an efficient index over the view while avoiding storage of many unused results.

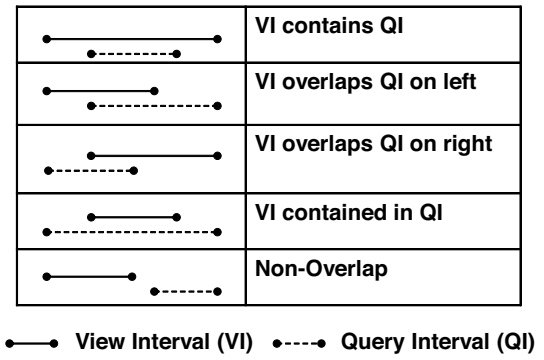


Figure 13: Types Of Overlap Between View-Intervals And Query-Intervals

5.3.1 Handling Overlapping View-Intervals

In several occasions multiple View-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 View-Intervals which overlap with the given Query-Interval. Thus if there are View-Intervals [2,7],[2,10],[5,10] and if the Query-Interval is [6,10], we would scan all three View-Intervals. There is also an added overhead in the view loading in this case. To store results for overlapping intervals we need to first check if some previously computed results have already been stored in an overlapping view interval.

- **"Exact Overlap" or "Contained".** This referes to the first type of overlap in Figure 13. The Query-Interval is completely contained in at least one of the views in the list of View-Intervals. This happens when the leftbound of the Query-Interval is greater or equal to the View-Interval's leftbound and the rightbound of the Query-Interval is less than or equal to the View-Interval's rightbound. In this case all the results are guaranteed to be present in the list of views which overlap this Query-Interval. Hence we will scan all the corresponding views which overlap with this Query-Interval. Some results in the view might not be used by the present Query-Interval which are filtered out.
- **"Partial Overlap on the right/left" or "Contains".** If the Query-Interval does not exactly match with any of the existing View-Intervals, then we would look for a partial overlap. In the case when the leftbound of the Query-Interval is greater or equal to the View-Interval's leftbound and the rightbound of the Query-Interval is greater than the View-Interval's rightbound as shown in the second type of overlap in Figure 13 we call it a Partial Overlap on the Right. In this case the inner sub-query is computed only for triggering events that have occurred between the Query-Interval.rightbound and View-Interval.rightbound similar to the partial compute in the Continuous Sliding View. While storing results we would make sure that these newly computed results are not present in any of the overlapping View-Intervals. However when there is a Partial Overlap on the Left or the Query-Interval contains the View-Interval, a partial incremental compute is not possible due to the SASE-based right to left stack-based joins. It is thus always desirable for the Query-Intervals to move to the right, so that Overlaps on the Right occur more often.
- **"Non-overlap".** If the Query-Interval does not have exactly overlap or partial overlap on the right with any existing View-Intervals, we need to compute results for the given Query-Interval from the scratch.

5.4 P-CSV Maintenance

When the Query-Interval is not overlapped by any existing View-Interval, new results need to be computed. The sub-query will be computed for the given Query-Interval. It is then added to the list of View-Intervals so that it can be further re-used by future runs. While if a Query-Interval is overlapped by a View-Interval we will have to compute the sub-query completely or partially depending on the type of overlap and will merge the results with the existing View-Interval by appending the results. Similarly for Query-Intervals Containing a View-Interval, the Query will be recomputed and the existing View-Interval will be extended. The steps for view maintenance are summarized in the Algorithm 1

Algorithm 1 Passive Continuous Sliding View Usage

Require: Query q_i, View-Interval-List $viewInterval$
91: **for** all viewIntervals v_i **do**
92: **if** $queryInterval$ containedIn v_i **then**
93: Reuse(v_i)
94: **end if**
95: **if** $queryInterval$ partialOverlapsOnLeft v_i **then**
96: Reuse(v_i)
97: **end if**
98: **if** $queryInterval$ partialOverlapsOnRightt v_i **then**
99: Reuse()
910: ViewUpdate(q_i)
911: **end if**
912: **if** $queryInterval$ contains v_i **then**
913: Reuse()
914: ViewUpdate(q_i)
915: **end if**
916: **if** $queryInterval$ NonOverlap v_i **then**
917: ViewUpdate(q_i)
918: **end if**
919: **end for**

There is a search needed to find the overlapping View Intervals. Any interval tree index can be used to maintain the list of view intervals. The indexing scheme described in [24] works well for time intervals and is a simple light weight augmentation of the B+ tree. This indexing speeds up the view retrieval for this Passive CSV design.

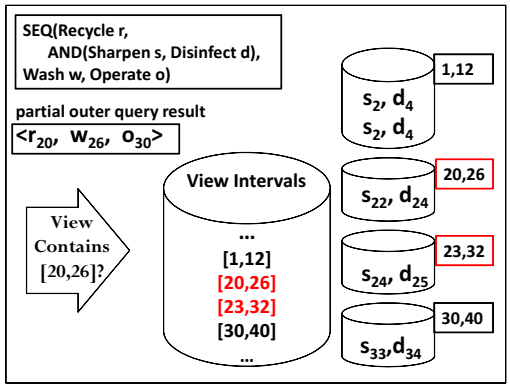


Figure 14: Example demonstrating Passive Continuous Sliding Views

EXAMPLE 5. *Figure 14 shows the state of the view at a given time for the given query. It contains a set of View-Intervals for which results have already been computed. When the tuple o_{30} arrives, the query is triggered and the outer result $<r_{22}, w_{26}, o_{30}>$ is formed. The Query-Interval extracted is [22,26]. The Interval is matched against the list of Intervals in the Passive Continuous Sliding View. Two View-Intervals are found to overlap with the given Query-Interval. They are both probed to get the final results for that sub-query.*

5.4.1 Optimization for Negative Sub-queries

Passive CSV 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 view 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.

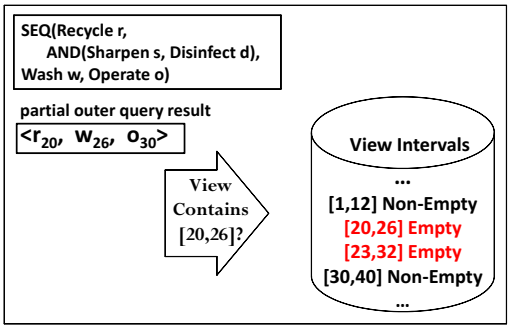


Figure 15: Passive CSV for Negative Sub-Queries

EXAMPLE 6. *For the query shown in Figure 15, when the tuple o_{30} arrives, the query is triggered and the outer result $<r_{20}, w_{26}, o_{30}>$ is formed. The Query-Interval extracted is [20,26]. The Interval is matched against the list of Intervals in the Passive View. For the matched View Interval the boolean value of the isEmpty flag is set to True. It therefore returns true and the outer partial result is returned.*

Figure 25 compares the memory usage of Passive Continuous Sliding Views and Continuous Sliding Views for the query specified

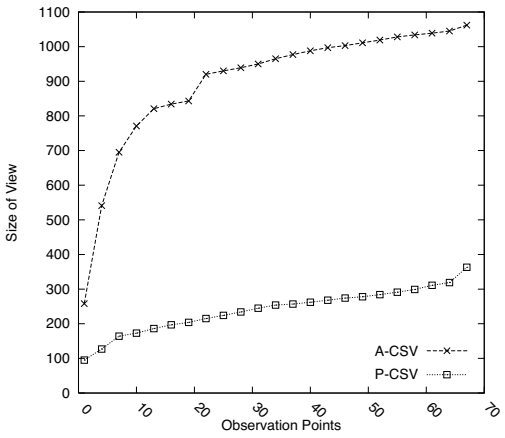


Figure 16: Comparing Memory Usage by Passive Continuous Sliding Views v/s Continuous Sliding Views

in Figure 14. By partitioning the view into smaller fragments and indexing them by the intervals attached to them, over-computation of results is avoided. The Aggressive CSV computes 3 times more results than the Passive CSV. In a scenario with memory constrains, the passive model is more practical than the aggressive one. In the worst case, the Sliding View will end up computing and storing results which are never used at all. This problem can be completely avoided using the Passive Continuous Sliding Viewing technique.

5.5 Disadvantages of Passive Continuous Sliding Views

Despite its advantages over the pre-fetch based view maintaining technique, Passive Continuous Sliding View Maintenance suffers from a few drawbacks.

- The main drawback of this method of view maintaining is the complexity of storing the results without duplication. A given result could belong to multiple View Intervals. However to make sure it is stored once, it is necessary to reverse index these results. However this adds significant overhead on the execution time.
- On the same lines, finding all correct results spanning over multiple View Intervals requires the maintenance of a smart light-weighted index. However this too adds significant performance overhead.

6. BALANCED CONTINUOUS SLIDING VIEWS

Choice between Continuous Sliding Views and Passive Continuous Sliding Views is that of memory-CPU trade-off. This leads us to propose a hybrid approach of 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 View-Intervals that would be formed. This Balanced Continuous Sliding View (B-CSV) technique guarantees that the view will move only in one direction that is towards the right. This is beneficial as we can do partial compute to update the view. In this method, we do some pre-fetching of results similar to Continuous Sliding View, however making sure that such results do get used at some point of time thus overcoming the major problem of the aggressive view maintenance technique. At the same time we maintain a finer level of granularity of the view like the passive approach.

This technique can be rightly called a hybrid of the above two approaches because of two reasons: Firstly, this approach selectively pre-fetches results. It does not blindly pre-fetch results like the first technique nor does it load the view based on complete re-use. Secondly, it maintains the stored results are a level of granularity that is finer than the aggressive continuous sliding view and courser than the passive approach.

6.1 B-CSV Design

The data structure for this technique is the same as for the Passive Continuous Sliding View Maintenance. For a given sub-query we will maintain a View-Interval and the results associated with the respective interval.

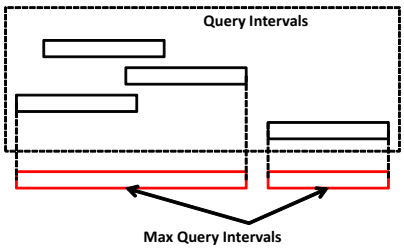


Figure 17: Maximum Query-Interval

6.2 View Loading

The view 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 View-Intervals for such overlapping Intervals. We precompute the results of the inner sub-query for these View-Intervals and store them. Figure 17 visually explains the meaning of maximum overlapping Query-Interval.

6.3 View Usage

For a given outer query result, if the Query-Interval is found contained in the list of View-Intervals, the results are directly retrieved only from that View-Interval. There is no case of Overlapping View-Intervals. This is the main reason which saves this technique from the disadvantage of Passive Continuous Sliding Views. On the other hand, since we do not precompute any result, it also does not suffer from the disadvantage of the Continuous Sliding View. Algorithm 2 shows the steps for view usage.

Algorithm 2 Balanced Continuous Sliding View Usage

```
Require: View-Interval VI, Query-Interval QI
Interval maxInterval = r.get(0).extractQueryInterval();
for Interval QI:r.extractQueryInterval() do
    if maxInterval < QI then
        maxInterval := QI
    end if
end for
if maxInterval.containedIn(VI) then
    computeNeeded = false
    if computeNeeded == true then
        PartialCompute(maxInterval)
    else
        Reuse()
    end if
end if
```

6.4 View Maintenance

By the way the Max-Interval is computed, it is guaranteed that the View-Interval can only move forward in time as the window slides forward. Thus, View Maintenance will require only a delta compute which are inserted into the view.

Claim: *The paired values in the View-Intervals always increases.*

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 View-Interval be [t_m,t_n]. If the View-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 View-Interval should have been [t_x,t_n] which is a contradiction.

7. COST ANALYSIS

We will now show a simple cost analysis to theoretically show the advantages of using view maintenance techniques. For that we must first introduce certain terminologies in Table 1 that will be used in the cost model. **Cost Model for Flat Stack-Based Execution.** For

Table 1: Terminology Used in Cost Estimation

Term	Definition
$C_{compute(qi)}$	The cost of computing results for a query q_i independently
P_E	Selectivity of all single-class predicates for event class E
$P_{Ei,Ej}$	Selectivity of predicates between event class E_i and E_j
$Pt_{Ei,Ej}$	Selectivity of the implicit time predicate of subsequence (E_i, E_j)
$\|S_i\|$	Number of tuples of type E_i in time window TW_P
$\|S_{qi}\|$	The number of results for a query q_i
Ov	The fraction of overlap between the View-Interval and the Query-Interval q_i

an event pattern query q_i = SEQ(E_1, E_2 ,..., E_i ,..., E_n), E_i is an event for 1 < i < n. Using stack-based pattern evaluation, the cost of computation $C_{compute(qi)}$ is formulated in Equation 1.

$$C_{compute(q_i)} = \sum_{i=1}^{n-1} |S_{i+1}| * [\prod_{j=1}^{i} |S_j| * Pt_{E_j,E_{j+1}} * P_{E_j,E_{j+1}}] \quad (1)$$

Cost Model of Iterative Nested Execution. For a nested expression q_i, qi_{root} represents the outer most query and qi_{child_j} represents its jth child. $C_{compute_{qi}}$ consists of computation costs for qi_{root}, computation costs for qi_{child_j} and joining costs as captured Equation 2.

$$C_{comp(q_i)} = C_{comp(qi.root)} + |qi.root| * (\sum_{j=1}^{n} C_{comp(qi.childj))} \quad (2)$$

Iterative Nested Execution with Continuous Sliding Viewing Technique. For a nested expression q_i, qi_{root} represents the outer event expression and qi_{child_j} represents its jth child. $C_{compute_{qi}}$ mainly consists of computation costs for qi_{root}, view maintenance costs for qi_{child_j} as shown in Equations 3 and 4.

$$C_{comp(q_i)} = C_{comp_{qi.root}} + |S_{qi.root}| * (\sum_{j=1}^{n} C_{VA} + C_{upd_{qj.child}}) \quad (3)$$

$$C_{upd(qi)} = (1 - Ov) * \sum_{i=1}^{n-1} |S_{i+1}| * P_{E_{i-1},E_i} * [\prod_{j=1}^{i} |S_j| * Pt_{E_j,E_{j+1}}] \quad (4)$$

$$C_{VA} = Size\ of\ the\ View \quad (5)$$

Iterative Nested Execution with Continuous Semantic Viewing technique. The computation cost using Semantic View is similar to

the cost for Continuous Sliding View as shown in Equation 6 except the cost for View access is constant and has thus been ignored.

$$C_{comp(q_i)} = C_{comp_{qi.root}} + |S_{qi.root}| * (\sum_{j=1}^{n} C_{upd_{qj.child}}) \quad (6)$$

From the above set of equations, it can be seen that the outer query results are joined with the inner query results by a cross product. Hence the factor $|S_{qi.root}|$ which denote the number of outer query results, is multiplied with the cost of computing the inner query. However in caching techniques, this computing cost of an inner query is replaced by the view access and maintenance cost which are much less compared to computing the inner queries from scratch.

8. PERFORMANCE EVALUATION

We now demonstrate the effectiveness of the Balanced Continuous Sliding Views over Aggressive Continuous Sliding Views and Passive Continuous Sliding Views. The experiments measure the execution time for the queries, which include the time for view construction, look-up and purging. Memory sizes have been

8.1 Varying Length of Subqueries

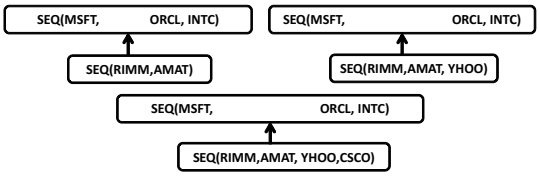


Figure 18: Varying Length of Children Queries

Figure 18 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. Figure 19 (a), (b) and (c) show the comparison between the three continuous view strategies with varying sub-query length. Clearly the length of the sub-query has an effect on the relative performances. The Balanced Continuous Sliding Views and the Aggressive Continuous Sliding Views techniques take significantly less execution time compared to the Passive Continuous Sliding Views as the length of the sub-queries in increased. This is due to the fact that the Passive Continuous Sliding View computes minimally and hence results in too many calls to the view maintenance function which slows it down significantly. Also as seen from our further analysis of the data, the occasions when Continuous Sliding View computes un-used results arises very rarely and hence proves better than Passive Continuous Sliding Views. However, Balanced Continuous Sliding View maintenance technique not only makes minimal calls to the view maintenance function but also maintains a granular view look-up structure thus making it perform better than both the other techniques. As the length of the sub-query in increased, the maintenance cost is increased and this maintenance happens much more frequently in Passive Continuous Sliding Viewing and hence its performance deteriorates compared to the other techniques.

8.2 Varying Number of Subqueries

Figure 20 shows 3 queries with varying number of the inner subqueries from one sub-query to three sub-queries. The Window size is kept constant at 500 and the number of sub-queries is kept constant. Figures 21 (a), (b) and (c) show a comparison between the

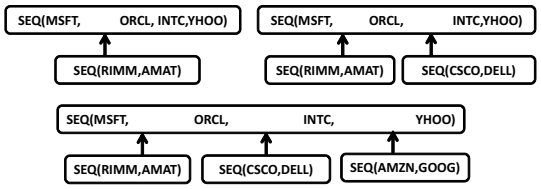


Figure 20: Varying Number of Positive Children Queries

three view maintenace techniques as we vary the number of subqueries shown in Figure 20. Again we see that the Balanced Continuous Sliding View Maintenance takes a much less execution time compared to Continuous View Maintenance or Passive Continuous Sliding View Maintenance. As the number of subqueries is increased, the difference in performance between Passive Continuous Sliding Viewing and Continuous Sliding Views increases as expected because for every subquery the number of recomputations is much larger for Passive Continuous Sliding Views than Continuous Sliding Views.

8.3 Rewriting v/s View Maintenance

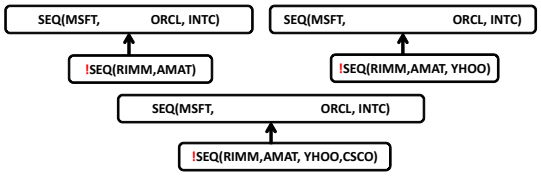


Figure 22: Varying Lengths of Negative Sub Queries

Figure 22 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 subquery has two event types in all of them. Figure 23 (a), (b) and (c) 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 Balanced CSV technique takes a much shorter execution time compared to the Rewriting technique [13]. This is due to the fact that the Rewriting technique results in extremely complex queries when the length of the negative sub-query is increased.

8.4 Varying Selectivity for Predicate Correlation

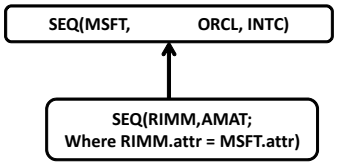


Figure 24: Query with Predicate Correlation

In the following set of experiments, we will use queries with predicate correlations between different levels of nested queries. We will compare how the optimized CSV for predicate correlation performs against general CSV without optimization. The queries used for the experiments are shown in Figure 24. We will vary the

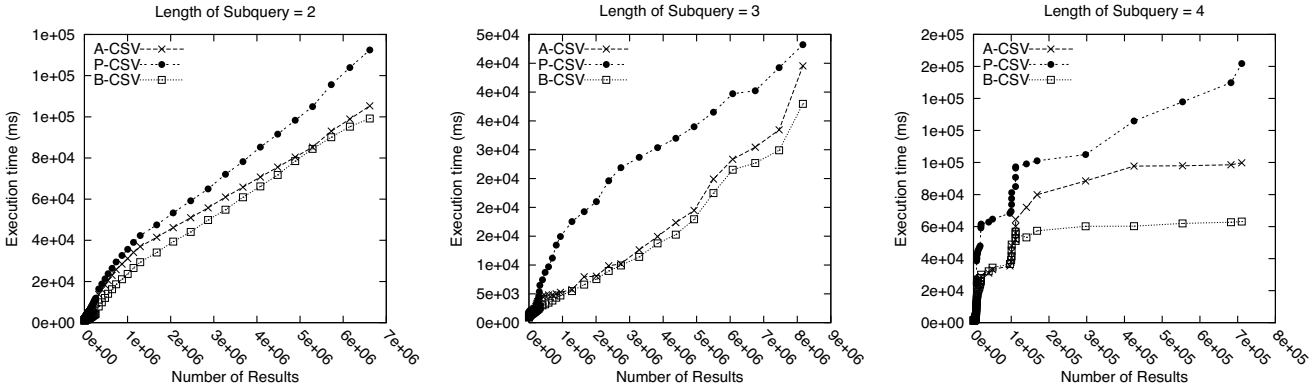


Figure 19: Comparing Execution Time with Aggressive Continuous Sliding View (A-CSV), Passive Continuous Sliding View (P-CSV) and Balanced Continuous Sliding View (B-CSV) with Varying Length of Sub-queries

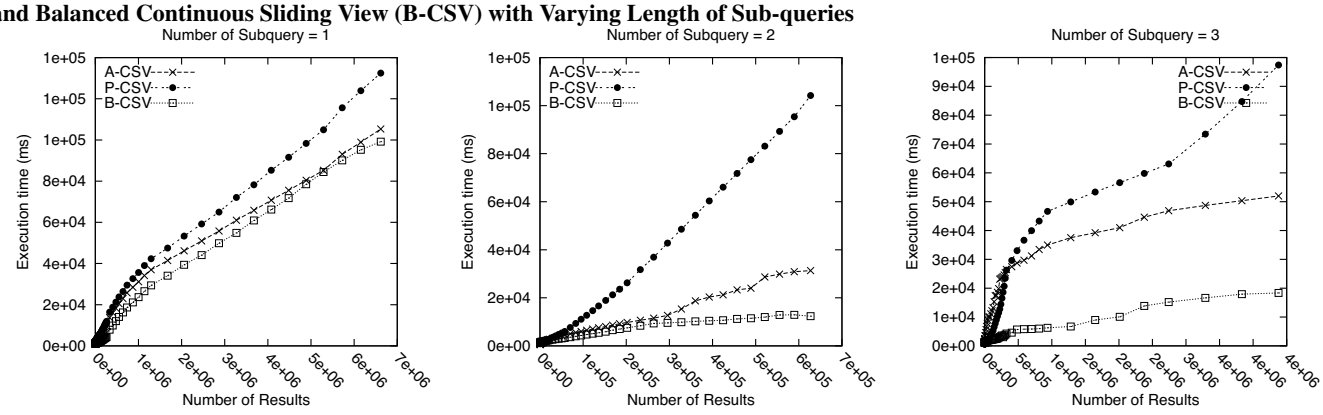


Figure 21: Comparing Execution Time for Aggressive Continuous Sliding View (A-CSV), Passive Continuous Sliding View (P-CSV) and Balanced Continuous Sliding View (B-CSV) with Varying Number of Sub-queries

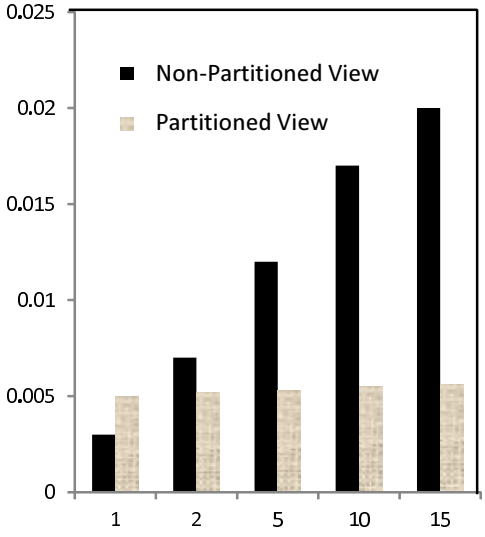


Figure 25: Varying Selectivity of Correlated Predicate

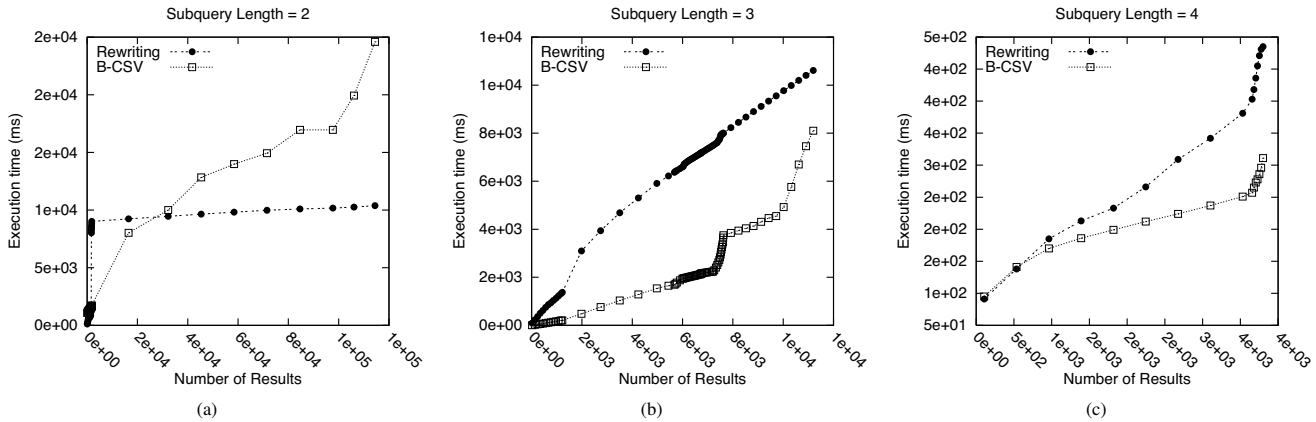
domain size of the attributes on which there is an equivalence correlation between the outer and inner query. Thus the selectivity is increased. The chart in Figure 23 (d) reflects the effect of increasing selectivity on the average time for the computation of a single result. It shows that with the increase in selectivity the partitioned view takes almost the same time for different selectiveness, while for the non-partitioned view the average execution time goes on increasing with increasing selectivity. The reason for this increase is that although the same number of outer results scan the same number of inner ones thus keeping the total time of computation constant, however when it is averaged over the smaller number of actual results produced, the average execution time increases.

8.5 Memory Usage of Viewing Techniques

Figure 26 shows the memory utilizations of the various continuous view strategies based on the size of the view at a given instance of time. We use the query depicted in Figure 9 with a fixed window size of 100. Even in terms of memory the Balanced Continuous Sliding View consumes minimum memory. It neither stores extra unused results like the Aggressive Continuous Sliding View nor has a huge list of intervals like the Passive Continuous Sliding View. However between Continuous Sliding View and Passive Continuous Sliding View, the number of unused results computed in the A-CSV and size of the list of intervals of the P-CSV keep varying.

9. CONCLUSIONS

This paper focuses on optimizing the processing of Nested Complex Event Processing queries by designing the Continuous Sliding View structures for inner sub-queries. In particular, we designed a Continuous Sliding Viewing methodology for storing intermediate results. We described techniques for incrementally loading, purging and exploiting the view content. We proposed three alternate paradigms of continuous sliding view management, namely aggressive, passive and balanced. Our experiments compare the above mentioned viewing methodologies against standard iterative



(a) (b) (c)

Figure 23: Comparing Execution Time for Rewriting Technique and Balanced Continuous Sliding Viewing with Varying Number of Sub-queries

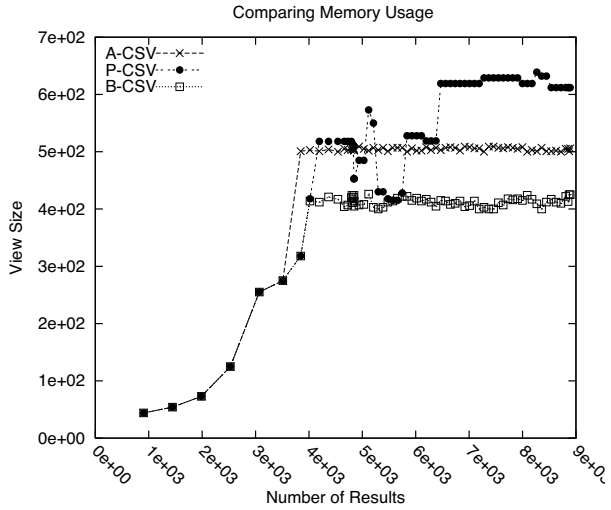


Figure 26: Comparing Memory Usage of Aggressive Continuous Sliding View (A-CSV), Passive Continuous Sliding View (P-CSV) and Balanced Continuous Sliding View (B-CSV)

processing technique for nested CEP query execution time. Our strategies significantly outperform iterative processing method by reducing execution time by up to 89 percent. The Aggressive CSV is found to out perform the Passive one in most cases, however the Passive CSV consumes less memory. The Balanced CSV however achieves both minimum execution time as well as memory usage. We also compared the performance of our caching technique against state-of-the-art method of processing Nested CEP queries which is Rewriting [8]. In [25] they authors observe that with increasing stream arrival rates and large join states, the CPU typically becomes strained before the memory does. Temporary data flushing [26] and compressed data representations further counteract the chances of a memory-limited scenario. If under duress complete results can no longer be produced at run-time, then the DSMS must employ the available resources to ensure the production of maximal run-time throughput (output rate). Therefore, in this work, we achieved optimizing the throughput of nested CEP queries in CPU-limited cases.

10. REFERENCES

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

[2] A. 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*, 2009, pp. 193–206.

[4] M. Liu, E. A. Rundensteiner, D. J. Dougherty, C. Gupta, S. Wang, I. Ari, and A. Mehta, "NEEL: The nested complex event language for real-time event analytics," in *BIRTE, VLDB WOrkshop*, 2010, pp. 116–132.

[5] 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.

[6] M. Liu, M. Ray, E. A. Rundensteiner, D. J. Dougherty, C. Gupta, S. Wang, I. Ari, and A. Mehta, "Processing nested complex sequence pattern queries over event streams," in *DMSN, VLDB Workshop*, 2010, pp. 14–19.

[7] "Esper 2009, http://esper.codehaus.org/. accessed july 2009."

[8] M. Liu, E. A. Rundensteiner, D. Dougherty, C. Gupta, S. Wang, I. Ari, and A. Mehta, "High-performance nested CEP query processing over event streams," in *ICDE*, April, 2011.

[9] W. Kim, "On optimizing an sql-like nested query," *ACM Trans. Database Syst.*, vol. 7, pp. 443–469, 1982.

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

[11] Mumick, IS. and Finkelstein, S. and Pirahesh, H. and Ramakrishnan. R, "Magic is relevant." in *SIGMOD*, 1990.

[12] A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross, "Implementing incremental view maintenance in nested data models," in *Database Programming Languages*, 1998.

[13] M. Liu, E. A. Rundensteiner, D. J. Dougherty, C. Gupta, S. Wang, and I. Ari, "E-Cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing," in *SIGMOD*, 2011.

[14] 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.

[15] B. Mozafari, K. Zeng, and C. Zaniolo, "Ik*sql: A unifying engine for sequence patterns and xml."

[16] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing queries with materialized views," in *ICDE*, 1995.

[17] L. A.Y., R. A., and O. J.J., "Query answering algorithms for information agents," in *Proc. National Conference on Artificial Intelligence*, 1996, pp. 270–294.

[18] P. Seshadri, M. Livny, and R. Ramakrishnan, "Sequence query processing," in *SIGMOD*, 1994, pp. 430–441.

[19] F. M. Dar S., J. B., S. D., and T. M., "Semantic data caching and replacement," in *VLDB*, 1996, pp. 330–341.

[20] KellerA.M. and B. J., "Apredicate-based caching scheme for client-server database architectures," in *VLDB Journal*, 1996, pp. 330–341.

[21] B. Cao and A. Badia, "A nested relational approach to processing sql subqueries," in *SIGMOD*, 2005, pp. 191–202.

[22] Dayal, U, "A unified approach to processing queries that contain nested subqueries aggregates and quantifiers." in *VLDB*, 1987.

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

[24] M. A. Nascimento and M. H. Dunham, "Indexing valid time databases via b+-trees," *IEEE Trans. on Knowl. and Data Eng.*, pp. 929–947, 1999.

[25] B. Gedik and et al., "Adaptive load shedding for windowed stream joins," in *CIKM*, 2005.

[26] B. Liu, Y. Zhu, and E. Rundensteiner, "Run-time operator state spilling for memory intensive long-running queries," in *SIGMOD*, 2006.