

YFilter++ – Efficient Declarative Querying Over Event Streams

Andy Carle & Eugene Wu

*Department of Electrical Engineering and Computer Science
University of California – Berkeley
acarle@cs.berkeley.edu, ewu@berkeley.edu*

Abstract

Event processing systems are growing in importance as more and more sequential data is becoming available from sensors and other real-time monitoring equipment. Much of this data must be processed quickly and efficiently in order to be useful. However, many current systems are drastically slowed by dependence on SQL over streams which is very inefficient for sequential data. We propose a new solution, an extension to the University of California – Berkeley’s YFilter system to handle event processing. Our new system, YFilter++, has shown much better performance than contemporary streaming database systems when run over event streams in initial testing. In addition, the event description language that we propose is more expressive than SQL over streams. There are still many optimizations that could be made to YFilter++, but overall we believe we have created a very viable system for event processing.

1. INTRODUCTION

Over the recent years, two communities of declarative querying systems have arisen for handling events. The events processing community focus on Event Notification Systems such as Siena[1], Le Subscribe[2], and Ready[3]. These systems concentrate on periodical evaluation of events and have not been known to process high rates of data very well. The primary focus this community has been language design of sequence based queries [4] rather than deployable systems.

The SQL community consists of database systems modified to support continuous streaming data sources. To this end are systems such as UC Berkeley’s TelegraphCQ [6]. These systems consist of SQL extensions to support continuous queries, and modified database engines to handle streaming inputs. Although they accept queries over streams, they have not been known to handle sequence queries well. In addition to streaming databases, there has been research done on temporal databases and active databases that extend SQL to support sequence based queries and database events, respectively. However we believe viewing this as an events processing problem allows for a more flexible solution.

In this paper, we present a method for using a Non-deterministic Finite Automaton (NFA) to continuously and efficiently evaluate declarative sequence queries over an event stream. We also describe the features necessary for any deployable events processing system and their implementation details in our system, YFilter++. Our system is able to run sequence based queries across input streams of high data rates. An additional feature we provide that is novel to the events community is aggressive sharing of evaluation of multiple queries.

Our initial results are very encouraging, and lead us to believe that YFilter++ is powerful enough to exist as a standalone system. For identical event streams and equivalent queries, processing time is orders of magnitude faster than equivalent queries run on the streaming database TelegraphCQ. We also believe that the language is highly extensible and reasonably expressive for most simple and complex queries.

2. THE PROBLEM

Over recent years the Computer Science community has continued to bridge the physical-virtual divide with new technologies such as virtual sensors[7] and other monitoring devices. This has led to an equivalent increase in data flow to monitoring systems. As the amount of data continues to increase, more and more efficient methods of processing it will be needed. These data streams are inherently sequential, so event-based techniques are a natural avenue to explore for these processing requirements.

One solution to this problem is to create custom code for every sensor array (virtual or physical) that is deployed. Such code tends to have high overhead, both in creation and maintenance. As project requirements change and sensors are added and removed from the system new queries must be added to the system. Custom code generally requires massive rewrites to already very complicated code. This leads to lots of time spent recoding and loss of program performance as optimizations are lost in subsequent rewrites. Generally, this is the argument for declarative systems and languages over fine-grained customized code.

```

With
  Yfilter.shelf as (select * from yfilter.XML [range by '100 sec' slide by '0.1 sec'] where event = 'shelf')
  Yfilter.exit as (select * from yfilter.XML [range by '100 sec' slide by '0.1 sec'] where event = 'exit')
  Yfilter.counter as (select * from yfilter.XML [range by '100 sec' slide by '0.1 sec'] where event = 'counter')
(select 'shoplifter', e.time
From   yfilter.shelf s [range by '100 sec' slide by '0.1 sec']
       yfilter.exit e [range by '100 sec' slide by '0.1 sec']
       yfilter.counter c [range by '100 sec' slide by '0.1 sec']
where  s.id = e.id AND c.id = e.id AND
       e.inputtime > c.inputtime AND c.inputtime > s.inputtime AND
       e.inputtime-s.inputtime < 1 hour)

```

Fig. 1, a purchasing query written in SQL for TelegraphCQ

Currently, the most common solution for handling this data is to use SQL-based streaming data processors, such as TelegraphCQ and the Stanford Stream Data Manager[8] (STREAM). However, there are limitations in the design of SQL that make event processing slow at best and difficult to implement. The main issue is that SQL as a language and the Relational Model of databases were designed to perform operations over sets of data. Event processing, on the other hand, is an operation over a list, or sequence, of data items. Temporal databases offer some potential in this area, but have yet to be well explored. The inefficiency of SQL over sequences is detailed in [9]. Such SQL blocks are difficult to write and even more difficult to decipher. See Fig. 1, a block of SQL written to detect purchase events on Telegraph CQ.

This slow performance is not acceptable in the domain of event processing for several reasons. First, sensors are quickly becoming powerful enough to send extremely fast streams of data. Second, as sensors become less expensive it will be feasible to have much larger arrays of sensors active at one time. This leads to a much faster event stream. Third, as virtual sensors prove their value we will see more and more input from virtual sources. The rate of input from such data sources is practically unlimited. Finally, event recognition systems are often used in real-time, critical situations where it would be unacceptable to delay processing or, much worse, drop events completely.

Given these problems it seems clear that a system that is much more efficient than SQL and much more flexible than application-specific, customized code is needed. It is to this end that we suggest the following event specification and our proposed solution, YFilter++.

3. EVENT DESCRIPTIONS

In the following section we describe the major data and language constructs necessary for the reader to understand the theoretical background of our system language. Reference [10] lays out a clear description of event types. These events include events that happen within the system itself (analogous to database triggers), events external to the system, and events triggered by the passing of time. For the current version of YFilter++ we chose to focus on external events as described by the event types below. Internal events are modelled in our system as an external event that is looped back as an input into the system. YFilter++ does not currently support events triggered strictly by the passing of time, but an external sensor could provide timed inputs and serve as a time-based alarm.

As the running example throughout this section, we present the “Shoplifter” scenario. In this scenario, there are

RFID tags attached to every product in a store, and RFID readers located at the product shelves, the checkout counters, and the exits. We define the shoplift query to be the sequence of RFID reader sensing of the same product RFID tag at the shelf, exit but not the counter. Row g of table 1 is an example shoplifting query in our language.

3.1. Event Type One – Simple Event

A simple event is one that can not be decomposed into any smaller parts recognizable by the event processing system. A simple event is instantaneous and atomic; therefore, it occurs at a discrete point in time. As in [10] we distinguish between the description of an event, called its event type (ET), and a specific occurrence in time of an event that holds all of the details of that event as parameters, called an event instance (EI).

Example – A product with product ID 17 is seen on a shelf at 12:47pm.

3.2. Event Type Two – Complex Event

A complex event is a collection of Simple Events, possibly filtered with some set of operators (which we will define next).

Example – A product with product ID 17 is seen on a shelf at 12:47pm and a product with product ID 17 is seen at the exit at 1:12pm.

References [11] and [12] describe sets of highly expressive event operators. These operators describe presence and absence, sequencing, and timing of events in input. Here we present a combination of operators taken from these two papers and our own ideas for what was needed to make our system successful. The language described here is more restrictive than those presented in [11] and [12] in that it does not allow for precise ordering of events. For example, you can not specify that you want to return a result when seeing the 4th event of one type five events after the 10th event of another type. However, we argue that this type of expressiveness is not needed in the majority of event processing applications and that many complex queries of this form could be built out of simpler operators and recursive querying (described in Section 6).

3.3. Operator One – Directly Proceeds

This operator indicates that the simple event before it must occur directly before the simple event after it. This allows for sequencing of events. This corresponds to the “/” character in our language.

Example – Return all incidents when a product with product ID 17 is seen on a shelf preceding a product with product ID 17 being seen at the exit, with no other events recognized by the system in between.

3.4. Operator Two – Proceeds

This operator indicates that the simple event before it must occur sometime before the simple event after it. This allows for sequencing of events with the potential for noise in the middle. This corresponds to the “/” characters in our language syntax.

Example – Return all incidents where a product with product ID 17 is seen on a shelf preceding a product with product ID 17 being seen at the exit. Any number of other events could come in between these two events.

3.5. Operator Three – Negation

This operator indicates an event type that is not allowed to occur between two other event types. This corresponds to the “!” character in our language.

Example – Return all incidents where a product with product ID 17 is seen on a shelf preceding a product with product ID 17 being seen at the exit without seeing a product with product ID 17 at the checkout counter in between.

3.6. Operator Four – Equivalence

This operator indicates that you are interested in determining complex events only across all events where a certain parameter is the same. This corresponds to the command separated list enclosed in brackets “[]” at the beginning of our language.

Example – Return all incidents where a product with the same ID was seen on the shelf and then at the exit, but was not seen at the checkout counter in between.

3.7. Operator Five – Windowing

This operator indicates a time frame in which the first and last event in a sequence must occur. This corresponds to the integer value within parentheses “()” in our language.

Example – Return all incidents where a product with the same ID was seen on the shelf and then at the exit within 1 hour, but was not seen at the checkout counter in between.

4. PROPOSED SOLUTION

4.1. Event Representation and Language

We believe that the most natural way of expressing events is through XML. This is due to several reasons. First, is the increasingly wide spread acceptance XML enjoys as de facto standard for information encoding and transmission. Because of this, we believe that the sensor community will soon describe their events in XML natively. In the mean time, XML is a very efficient mechanism for storing data. XML elements are particularly good storing event information consisting of an event name and a set of attributes about the event. The clean semantics of XML make parsing input and generating output a very efficient process.

When working with sequences in XML it is natural to look towards XPath[13] as a querying tool. XPath is used to specify constraints over both structure (ordering and sequence using path expressions) and content (using value-based predicates). In addition, the XPath specification already includes several of the operators and event

Semantics that we need to query our system. Table 1 lists each of the event definition components from Section 3, the symbolic notation in our extended XPath, and the example queries from Section 3 above. The bottom row of the table shows the completed shoplifter query from Section 3.7 in our extended XPath.

4.2. Query Representation during Processing

There are two distinctly different ways to process events. The lazy approach is to wait until recognizing an ending element of a query and then process backwards to determine whether or not a full event occurred.

The eager approach is to advance all possible query paths any time any input that may eventually satisfy a query enters the system. We argue that eager processing is an efficient way to process complex events in shared processing. This is because we are able to eliminate invalid sequences early, thus saving wasteful computation.

In this NFA, each element along the path expressions of the extended XPath queries is represented as a state. The final element of any given path expression is an accepting state in the NFA. The element names are the edges in the graph. See Fig. 2, an NFA representing a purchasing query, /Shelf//Counter//Exit.

To allow for efficient processing, at run time the currently active states for the NFA should be kept on a stack. As XML input elements enter the system, they

TABLE 1
EVENT COMPONENTS AND EXTENDED XPATH QUERIES

Event Component	Symbolic Notation	Sample Query
A. Simple Event	One-deep XPath path	//shelf-sensor//
B. Complex Event	Variable Depth XPath path	//shelf-sensor//exit-sensor
C. Directly Proceeds	XPath Single Slash	//shelf-sensor/exit-sensor
D. Proceeds	XPath Double Slash	//shelf-sensor//exit-sensor
E. Negation	!	//shelf-sensor/!counter-sensor//exit-sensor
F. Equivalence	[equivalence list]	[prodID]&&//shelf-sensor/!counter-sensor//exit-sensor
G. Windowing	(window size)	[prodID]&&//shelf-sensor/!counter-sensor//exit-sensor&&(1 hour)

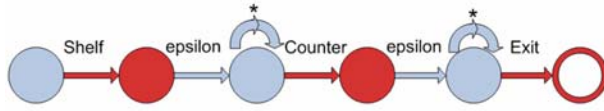


Fig. 2 NFA Representation of /Shelf//Counter//Exit

should probe all active, valid states, check for transitions on these states, take transitions as necessary, and move all states that are still active and valid to a new level on the stack. Any states that have not progressed should expire.

It is worth noting that when using this model to store the NFA it is possible to have a large number of states active at any one time. For instance, without a method of pruning, any active state that is blocked on a Proceeds operator (/) and waiting for input will continue to remain active and valid for the duration of the system’s run.

5. YFILTER

YFilter is an XML filtering system that provides the matching of XML documents with large numbers of XPath queries containing both structural and content constraints. YFilter encodes path expressions using an NFA based approach which enables highly efficient shared query processing. Its desirable features include:

- XPath parsing and evaluation
- XML input parsing
- NFA-based query evaluation
- Aggressive query with nearly no overhead
- Post processing
- Proper handling of a “wildcard” (*) elements

YFilter is modeled after the previous system XFilter[14] and inherits many of these excellent properties from that project. The main difference between XFilter and YFilter is that XFilter builds individual Finite State Machines (FSM) for each query in the system while YFilter takes a more general, nondeterministic approach which lets it combine all of the active queries in the system into a single NFA. This has the potential to reduce the total machine size in every case, and massively reduces the total machines size when there are high levels of sharing between queries. See Fig. 3, a sample set of queries and their corresponding YFilter NFA.

YFilter reads the active queries and creates the master NFA at startup time. Queries can be incrementally added or removed from the system with very low overhead, due to the Nondeterministic quality of the state machine. It then uses an event-based (SAX) parser to read through incoming XML documents one at a time and apply structural matching. Specifically, events raised during parsing are used to drive the transitions between states in the NFA. Predicate evaluation is done in a post processing step.

Finally, YFilter is an excellent core system that was designed to be easily extensible and has been shown [5] to be robust and efficient under a variety of work loads.

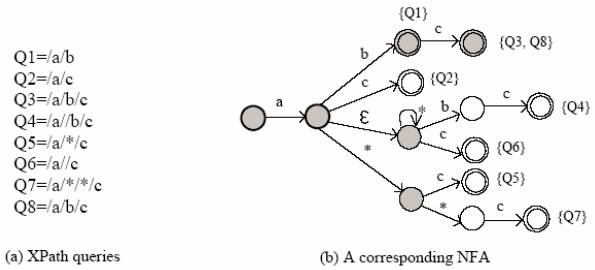


Fig. 3 Sample YFilter queries and corresponding NFA

6. YFILTER++ FEATURES

6.1. Handling Streaming Events

YFilter was originally designed to process well defined static XML documents; they follow proper XML conventions and adhere to the conventions set forth in a Document Type Definition (DTD). To handle streaming XML data in YFilter++ we treat the data stream as a potentially infinitely nested XML document. Multiple streams are merged and treated as a single data source.

Because we require the data stream to be strictly increasing in time, managing multiple unsynchronized streams will cause problems. In the current implementation we assign each input a timestamp as it is recognized by the system.

This could cause difficulties if the system is provided with inputs from a variety of sources with varying latency and throughput, as some inputs may be processed out of order. We plan to study this problem as future research.

6.2. Recursive Querying

To allow arbitrarily complex events in Yfilter++, we have implemented a mechanism for recursive querying. Using this system a user can specify both a transformation as well as a destination host and port to send the result event, which itself is an XML element. We also provide a mechanism to route the result event back into the system as the next event to be evaluated. This recursion allows the user to specify arbitrarily complex queries by feeding results back into the system. An example is a simple “count” aggregation to count the number of shelf events that you see in the system with an XML element and 2 queries:

```
<count total="[integer]">
1: //count/shelf
2: //count[@total>50]
```

Query 1 is fed back into the system with the total attribute incremented, whereas query 2 prints a value when the total attribute is incremented above 50. This system is started by inputting the element <count total="0">.

6.3. Equivalence Definitions

In Section 3 we introduced the notion of an equivalence definition that describes the attributes that must have the same values among the events satisfying a query.

In order to facilitate this, we added a table to each NFA state listing the starter events of the query(s) corresponding to the state. Starter events are the event IDs of the event that first triggered the evaluation of the given query.

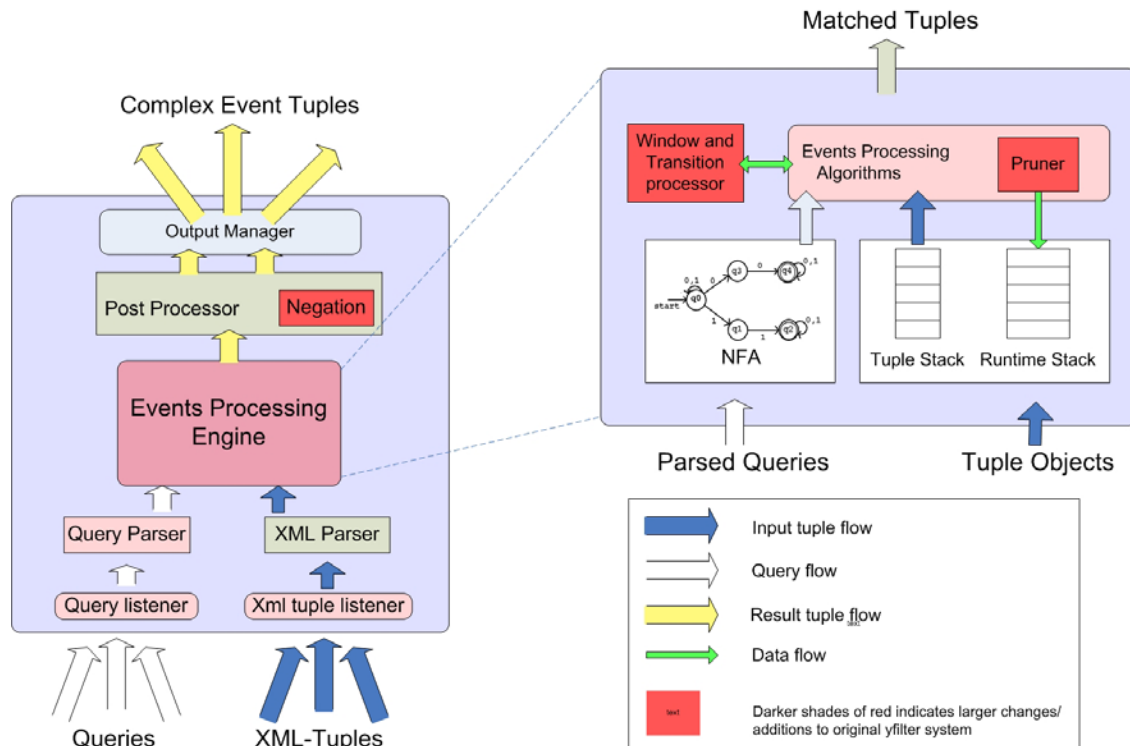


Fig. 4 System Architecture Diagram

On transition, the new event checks the table and ensures that its attributes match at least one starter event's attributes based on the query's equivalence definition. This is very effective at filtering out erroneous events in highly selective queries.

We also require a check on post processing due to shared NFA states. It is possible that two queries with different equivalence definitions match the same accepting states, in which case we must find only those events whose attributes are equal to those of the end event, depending on the equivalence definition.

Equivalence is necessary for event processing because users often are not interested in events that are strictly structural. (E.g. Product 1 on shelf, Product 2 at counter, Product 3 at exit is probably not attractive.) Interestingly, the notion of an equivalence definition that carries information across the components of a complex event is a novel idea in the field of event processing.

6.4. Negation

In Section 3 we defined the semantic meaning of negation to be an indication of an event type that is not allowed to occur between two other event types. We implement this in YFilter in two stages. First, during query parsing we strip out all negated elements and replace them with *, the wildcard element. We then store information about the element that was negated and its depth in the path expression in the negation table. The actual detection of a negation happens during post-processing. This is done for several reasons. First, detecting negations is generally computationally intensive. Our initial thought was that the relative infrequency of outputs that needed to be checked against the negation would lead to far less computation than checking negations on every transition through the NFA. Second, since we replace negations by a path element that

can be taken by any element and can loop on itself many times it becomes difficult to consistently check for the absence of an element. YFilter is well optimized to check for the presence of elements, but not as efficient at detecting when things have not occurred. For these reasons we decided to push negation detection to post processing in the current implementation of YFilter++.

To do this post-processing check we look at the query that is about to generate a result to see if its negation table has any entries. If it does, we check and see what XML element the wildcard actually transitioned on. If it is the element that was to be negated, we throw the result away. Negation is necessary for a good event processing system because the absence of an event is often just as important and the presence of an event. The shoplifting query is an obvious example of this; there would be no clear way to implement it without a negation operation.

6.5. Windowing

We have extended the XPath language to allow windows in the manner specified in Section 3. At the end of the XPath query, the system requires an additional integer, which defines the number of seconds between the first and last events in any sequence satisfying the query.

In order to enforce proper windowing semantics, we force all incoming events to include a strictly increasing timestamp attribute. We currently assign internal timestamps to each incoming event. In addition we keep a table within each NFA state of all starter events that began the evaluation path that to this state.

Our windowing recognition occurs in two phases. The first phase occurs during transitions across NFA states. When a new event enters the system and is a candidate for transitioning, we check to ensure the current time is within the window of the starter event's time. Thus we only transition if necessary.

The second phase is during post processing. Because of the aggressive sharing YFilter employs, multiple queries with different window specifications may exist on the same accepting state. In this case, it is vital for correctness to filter away the result for those queries that the window fails on.

Finally, as we will see next, windowing is necessary for any type of pruning.

6.6. Pruning

In modern streaming data processors, the solution to infinite streams of inputs is a sliding window over the stream that the query evaluates over. Because of the success of existing systems such as TelegraphCQ that implement windows, we decided to mirror this approach and implement pruning completely based on the user defined window.

The difficulty in pruning lies in removing only those elements that we can guarantee are not needed. For the scope of this project, we were only able to analyze two types: window based pruning and NFA state pruning.

The first type is relatively straight forward. The system keeps a queue of incoming events with the invariant that the time difference between the head of the queue and the tail be less than or equal to the largest window size amongst all queries.

The second type is performed at transition. When we check the timestamp of the new event against the table of starter events, we remove starter events from the table if the timestamp difference is greater than the largest window associated with that state. If the table becomes empty, the entire state may be safely removed. This prevents unnecessary transitions during query evaluation, and needless computation when generating result events at accept states.

Our current implementation is extremely conservative due to the difficulty of determining whether an event may be safely discarded, and because of the amount of state that must be updated to reflect the removed events. Even so, as we describe in section 8.3, the performance gain that the system enjoys from the mere existence of pruning is substantial.

7. The YFilter++ Language

YFilter++ and its event description language are clearly optimized for sequence based queries. As we mentioned before, SQL is optimized and highly expressive for set based queries over relations. However, as shown in Figure 1, it is not optimized for streaming, sequential sources. A sequence of N elements requires N-1 outer joins, which is both difficult to decipher, and computationally wasteful. While there has been work to address this issue [15], there have not been any deployable systems built upon these ideas.

The design goal of YFilter++'s language was for extensibility and ease of use, and towards this goal we believe we have succeeded. Although it is technically possible to write any event based query in SQL, it requires the use of User Defined Aggregates which border on the complexity of custom code. In order to express equivalent

queries in SQL, our language is orders of magnitude more compact and intuitive. In terms of extensions, we have provided an efficient framework so it is simple to add functionalities such as non-blocking aggregation (not possible in streaming databases), nested queries and result set transformations.

It is also interesting to note that the languages for systems such as TelegraphCQ are not able to support negations. This is due to the semantic difficulty in defining a logical endpoint at which to evaluate non membership in SQL based languages.

8. PERFORMANCE EVALUATION

In preliminary testing the current implementation of YFilter has performed very well. In this section we present several types of tests. First we present a performance comparison between YFilter++ and TelegraphCQ. Then we present a suite of tests designed to find the strengths and weaknesses of YFilter++ itself. These results will be used to guide future work on this system.

The YFilter++ vs. TelegraphCQ test was performed on an Intel Pentium 4 1.6 GHz desktop running the Fedora Core 2 operating system. The test was run on each system with identical input streams and semantically equivalent queries. It was not possible to run the same literal queries on each system due to the differences in the query specification languages of Y-Filter++ and TelegraphCQ. The data streams for this test were randomly generated and saved to disk by the XMLGenerator tool that we created for testing this project.

The YFilter++-only tests were run on an Intel Pentium 4 1.6 GHz laptop running Windows XP Professional. For each of these test suites, the individual tests were run under identical workloads to each other. However, both the overall system load and the input data streams were not consistent across the various test suites. For many of the tests this was necessary, as the tests were designed to test system performance on a very specific workload or a specific data schema. Unfortunately, this does mean that very few relevant comparisons can be made between test suites.

8.1. YFilter++ vs. TelegraphCQ

For this test we ran YFilter++ and TelegraphCQ over a randomly generated data file of new line separated tuples. Each tuple was a comma separated list of attribute values. We ran the purchase query on both systems. The query in SQL is shown in Fig. 1 at the beginning of this paper. The query in YFilter++'s language is:

```
[productID]&&/shelf//counter//exit&&(100)
```

For TelegraphCQ, we measured the time when the last tuple was parsed and entered the eddy, to when a result was generated and exiting the eddy. For YFilter++, we measured the average time to process an event, including parsing time. Fig. 5 shows the processing time per input of each system.

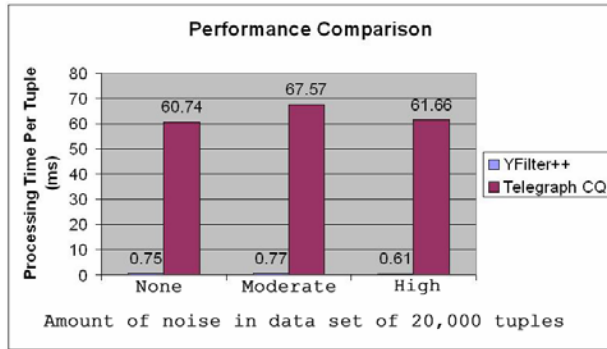


Fig 5 Performance Comparison between YFilter++ and TCQ

It is clear from this comparison that YFilter++ far outperforms TelegraphCQ for event processing. In fairness to TCQ, there is a great deal of overhead involved in processing inputs to a continuous query database system that is not needed in YFilter++. In other words, there are a great many operations that TelegraphCQ can perform that YFilter++ can not, but these all lay outside the realm of event processing. But, on the other hand, streaming database systems are indeed being used by some groups to do event processing currently. This is, in essence, our argument against using SQL over streams for event processing; it is just far too resource intensive and complicated.

8.2. Sharing Among Queries

For this test we ran 9 different test runs on YFilter++. The input data stream was the same for all tests, but the queries differed. For all tests the data input stream consisted of 313,594 XML inputs with randomly chosen elements ranging from a - c that were pre-buffered in to memory before beginning the performance test. The first 8 tests were for individual queries as shown in Table 2 below. The final test was run with all 8 queries in the system at the same time. The total processing time of each of these test runs is shown in Fig. 6.

Query	Query Expression
Query 1	[id]/a/b
Query 2	[id]/a/c
Query 3	[id]/a!/b/c
Query 4	[id]/a/!/b/c
Query 5	[id]/a/*/c
Query 6	[id]/a//c
Query 7	[id]/a*/*/c
Query 8	[id]/a/b/c

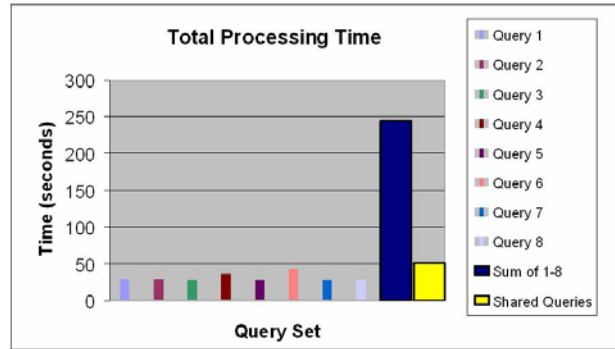


Fig 6 Isolated queries vs. Shared queries

These results show that much of YFilter's potential benefit does indeed come from sharing among queries in the system. This matches our expectation for YFilter and is a large part of why we chose it as the core of our extended system. Empirically, it appears likely that queries over event streams will have a high level of sharing, particularly if there is a large population of users and the event data is confined to a relatively small domain.

8.3. Pruning

For this test we ran the simple purchase query: [id]/shelf/counter/exit(50) over a small input stream of 5000 XML inputs with randomly chosen elements shelf, counter, or exit. We are comparing our current implementation of YFilter++ with pruning against a naïve system that does not prune. Fig. 7 shows the results of this test.

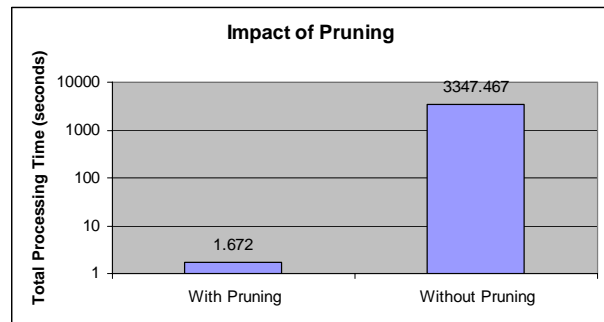


Fig 7 The affect of pruning on system performance (note the logarithmic scale on the Y-axis)

These results show that even the most conservative pruning makes giant leaps in performance. The difference in pruning and not pruning is the size of the NFA and the number of events buffered in the system. Thus we can see that the ability to do fine grained pruning is the key to high performance.

8.4. Impact of Number of Elements in Query

For this part of our analysis we ran a series of 5 tests. For each test the query [id]/<path>(50) was run where the number of elements in the path expression increased with each run. For example, the path for test 1 was /a, the path for test 2 was /a/b, the path for test 3 was /a/b/c, etc. The input stream for this test consisted of about 100,000 XML inputs with randomly chosen elements ranging from a to g. Fig. 8 shows the results of these tests.

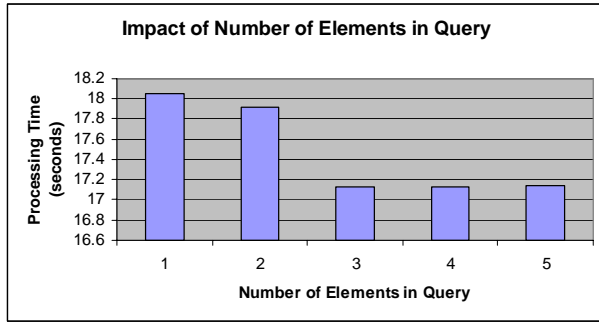


Fig 8 Impact of Number of Elements in Query on Processing Time

At first glance this graph may seem counter-intuitive. However it can be explained by the way our system is implemented. When we reach an accepting state, we do traverse through the buffer of events to compose the set of events that satisfy the query. As the elements in the query increase, more is filtered out during transition, and thus fewer results need to be generated. We expect the graph to asymptotically decrease to simply the system overhead.

8.5. Impact of Number of Negations in Query

In order to determine the impact of negations on YFilter++, we ran 5 queries through the system over identical inputs. The base query was `[id]&&/a//b//c//d//e//f//g&&(50)` where each subsequent query added an additional negation to an element in the middle of the path. The input stream for these tests consisted of about 100,000 XML inputs with randomly chosen elements ranging from a – g. The results of this test are shown in Fig. 9.

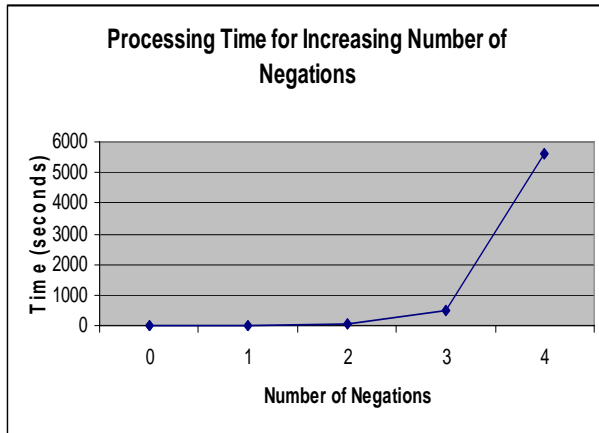


Fig. 9 Processing Time as a Function of Number of Negations

This spike in performance is due to the way negation is implemented in our system. We currently model the negation as a `*` and perform post processing to filter out invalid results. Out check for the non-existence of an element is done using brute force, in which case traverses all permutations of the event buffer.

8.6. Impact of Size of Window in Query

For this test we ran 10 different tests on YFilter++ for each of the follow two queries.

1	<code><id>&&/shelf//counter//exit&&(window)</code>
2	<code><id>&&/shelf//!counter//exit&&(window)</code>

For each of these queries we varied the size of the window and measured the total processing time. The same input

stream was used for all of these tests and consisted of approximately 20,000 XML inputs with randomly picked elements shelf, counter, or exit. Fig. 10 shows the results.

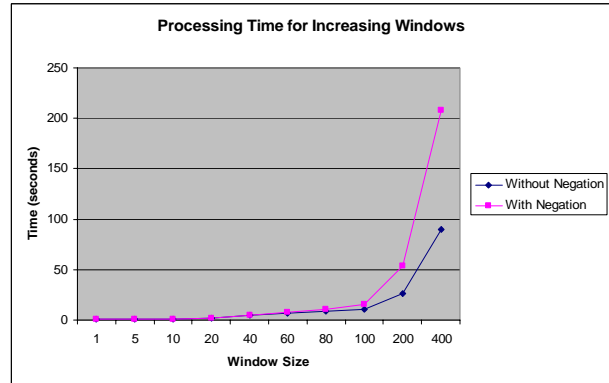


Fig 10 Processing Time for Increasing Windows

The performance of YFilter++ degrades to a point where it is no longer acceptably responsive as the window sizes get moderately large. This issue is addressed in part in the next section and in Section 6.6.

9. PROBLEMS WITH THE NFA-BASED APPROACH

As YFilter++ processes input it has to keep a very large amount of data about the state of the machine on hand at all times. For every active state in the system this is a complete history, including the elements that caused transitions and the attributes associated with those elements. The elements that caused transitions are needed to test negations in post-processing. The attributes are stored for YFilter’s predicate processing and equivalence evaluation. This information accumulates over time and is only removed during pruning. This buffer of stored XML inputs must be scanned many times to compute equivalence and negation. Therefore, as the buffer gets very large the time spent evaluating negation and equivalence grows proportionally. This is why YFilter++ performs poorly as window sizes get very large and pruning becomes less effective. We argue that this is an inherent flaw with an NFA-based execution model.

There are several optimizations to negation that could reduce the impact of this problem in future versions of YFilter++. First, the processing of negation could be pushed out of post-processing in to the processing of state transitions. This would increase the time it takes to process each input that but could dramatically speed up post-processing. Second, a running table of states that could be affected by negation could be stored. Only these states would need to be kept in the stack and only the XML inputs that affect those states would need to be kept in the input buffer. Third, in the current implementation there are many instances in the stack that describe the same state. These are used to keep track of `“/”` transitions, where a state loops back on itself. With a cleverer pruning algorithm, many of those duplicates could be removed. Finally, a table could be created to keep track of which paths through the runtime stack have been negated for each query. Once a path has been negated for all queries that use it the path could be safely pruned.

10. RELATED WORK

The current popularity of event processing has led to a wide array of literature on event processing systems and related concepts. These generally fall into a few categories, including the following:

10.1. Event Notification Systems (ENS)

This type of system includes the aforementioned Le Subscribe, Ready, and Siena along with other systems such as Herald[15] and Elvin[17]. These systems are generally based on the principles of Publish/Subscribe Systems (see [18] for more info on Publish/Subscribe). While these share a lot of common ground with our project, what sets YFilter++ apart from Event Notification Systems in general is the ability to handle continuous streaming event data. Most event notifications periodically check the state of some item of interest and report to their subscribers if that item has changed. For instance, a typical ENS could check a news website every half hour and notify subscribers if the top story has changed. This is a useful function, but does not map well to real-time or critical systems where every event must be processed as it happens rather than at a fixed interval. To use a common systems metaphor, ENS is based on polling while YFilter++ is essentially interrupt based.

10.2. Event Language Specifications

Many authors have written on the importance of specifying a robust and useful event language. We borrow heavily from [10, 11, 12] in Section 3 when we describe our own event language. It is interesting to note that the subject of event language specifications covers many fields within Computer Science and Electrical Engineering, including , which is about event specification for sequential circuits. Most of the pieces of our event description are shared with many other systems.

10.3. Active Databases

Many early attempts at event processing were modeled as triggers in active databases. Ode[20] combined standard database triggers with concepts from Object Oriented Programming to allow for the detection of more complex and flexible conditions. However, any sort of built-in database trigger is much more useful for detecting and handling internal database events than for processing external events. YFilter++ is very focused on external event processing and is much more efficient at doing this than an active database would be.

11. CONCLUSION

We have introduced a framework for complex event processing over streams based on NFAs. Our solution involves transitioning across states in an NFA based on incoming events.

In a prototype system called YFilter++, extended from UC Berkeley's YFilter, we show that this model is feasible and reasonably efficient. Our simple and extensible XPath-like language is suitable for the majority of simple queries, and can be easily extended for more complex queries. Our performance results show that YFilter++ runs 2 orders of magnitude faster than UC Berkeley's streaming database

TelegraphCQ for modest sized windows, which suggests that this system is fast enough to be considered as a viable standalone system.

We are interested in continuing work on this system. In our performance studies, we have found that window based pruning is wholly insufficient given the large sizes of windows in any practical query and the data rates we expect. Thus our main focus is a more aggressive pruning mechanism that strips out as much unnecessary NFA state as possible. Another area of interest is converting the system to an operator in a streaming database. We believe that a system that can efficiently query incoming streams in both SQL, and an event specific language is necessary in a world of sensors.

ACKNOWLEDGMENTS

Thanks to Yanlei Diao for both her feedback and her guidance in understanding YFilter, the UC Berkeley Telegraph group for their insight into TelegraphCQ and Professors Franklin and Brewer for teaching the class that inspired this project and many hours of deep thought.

REFERENCES

- [1] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332-383, August 2001.
- [2] J. Pereira, F. Fabret, H. Jacobsen, F. Llirbat, R. Preotiu-Prieto, K. Ross, and D. Shasha. LeSubscribe: Publish and Subscribe on the web at extreme speed. In *Proceedings of the ACM SIGMOD Conference*, 2001.
- [3] R. E. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the READY event notification service. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Middleware Workshop*, 1999.
- [4] A. Hinze. Efficient Filtering of Composite Events. In *Proceedings of the 20th British National Database Conference*, 2003.
- [5] Y. Diao, P. Fischer, M. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. Demo paper, in *Proceedings of ICDE 2002*, February 2002.
- [6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. *CIDR 2003*.
- [7] I. Stamos and P. Allen. Interactive sensor planning. In *Computer Vision and Pattern Recognition Conference (CVPR)*, pages 489-494, June 1998.
- [8] The STREAM Group. STREAM: The Stanford Stream Data Manager, *IEEE Data Engineering Bulletin*, Vol. 26 No. 1, March 2003
- [9] P. Seshadri, M. Livny, and R. Ramakrishnan, The Design and Implementation of a Sequence Database System, in *The VLDB Journal*, pages 99-110, 1996.
- [10] D. Zimmer, R. Unland. On the Semantics of Complex Events in Active Database Management Systems, in *Proceedings of the 15th International Conference on Data Engineering, IEEE Computer Society Press*, pages 392-399, 1999.
- [11] N. H. Gehani, H. V. Jagadish, O. Shmueli. Composite Event Specification in Active Databases: Model and Implementation. In *Proceedings of 18th International Conference on Very Large Data Bases*, Vancouver, August 92.
- [12] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 1994.
- [13] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0, November, 1999.
- [14] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of VLDB Conference*, 2000.
- [15] Carlo Zaniolo: Key Constraints and Monotonic Aggregates in Deductive Databases. *Computational Logic: Logic Programming and Beyond 2002*: 109-134. F. Cabrera, M. B. Jones and M. Theimer, Herald: Achieving a global event notification service. In *Proceedings of HotOS VIII*, Schloss Elmau, Germany, May 2001.
- [16] L. F. Cabrera, M. B. Jones and M. Theimer, Herald: Achieving a global event notification service. In *Proceedings of HotOS VIII*, Schloss Elmau, Germany, May 2001.
- [17] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the Australian UNIX and Open Systems User Group Conference (AUUG'97)*, 1997.
- [18] P. Th. Eugster, P. A. Felber, R. Guerraoui and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. In *ACM Computing Surveys*, June 2003.
- [19] C. P. Wu, C. L. Lee and W. Z. Shen. SEESIM – A Fast Synchronous Sequential Circuit Fault Simulator with Single Event Equivalence. In *Proceedings of the Conference on European Design Automation*, 1992
- [20] N. Gehani and H. V. Jagadish. Ode as and Active Database: Constraints and Triggers. In *Proceedings of the 17th International Conference on Very Large Data Bases*, 1991