Improved memory management for XML Data Stream Processing

Ravishankar Rajamony, Yanlei Diao
Department of Computer Science
University of Massachusetts Amherst
{rrvai,yanlei}@cs.umass.edu

ABSTRACT

Running XPath queries on XML documents with minimum memory usage is a challenge. YFilter 1.0 stores the entire document in memory. The extensions to YFilter applied in [3] are limited as they discuss memory management techniques for a limited taxonomy of queries. They do not handle cases where data is being shared between queries. We implemented a basic algorithm which extended [3] to incorporate sharing of data among queries. However, this implementation also faced a problem of delayed pruning of data from the memory along with the overhead involved in maintaining additional data structures. We propose a novel algorithm YCompact which has a buffer manager that handles pruning of irrelevant information. This is achieved with reduced overhead in terms of data structures. This resulted in significant memory improvement as compared to the previously implemented algorithms with no extra overhead.

Keywords

Data Stream, YFilter [1], XPath, Sampling, pruning, filtering, YCompact

1. INTRODUCTION

XML data streams have become ubiquitous for information exchange over the internet. Examples abound from news feeds, financial data feeds and network monitoring events. More recently, sensor technologies such as RFID have appeared as new sources of XML data streams, bringing along with them challenges to manage and utilize the data effectively.

In a traditional DBMS, the data is resident: it is stored in some medium to allow easy querying and updates. The queries are one-shot and can vary to cover a wide range of information needs. The data processing can be thought of as being query-driven, i.e. query results are generated as frequently as queries are invoked. However the streaming data environment is data-driven i.e. continuous queries are used to process unending streams of data. The answers generated for a particular query can quickly become obsolete in a very short period of time. Hence, continuous results need to be generated in a timely fashion.

YFilter [1] aims to provide fast on the fly matching of XML data to a large number of user interest specification. However, it stores the entire document in the memory. There is no pruning of data after query processing. FisT [10] provides a XML filtering system. FisT [10] performs holistic matching of twig patterns with incoming XML documents by transforming them into Prufer sequences. Fist evaluates its algorithm for different query sets and compares the algorithm's performance with YFilter [1]. However, it has not studied memory management techniques for processing queries over XML documents.

Improvements to YFilter were provided by Feng and Kumaran [3]. They improved the YFilter code base to incorporate memory management techniques with pruning of irrelevant nodes from the memory. However if node information was being shared between queries, there was no provision to reduce memory consumption by having a single copy of the shared node information in memory. We first implemented a basic algorithm that extended Feng and Kumaran's work [3] in improving memory usage by storing a single copy of node information in a separate shared buffer if there is sharing between queries. YCompact extended this idea by using a single global buffer and using pointers to keep track of shared node information. The different issues that we addressed in YCompact were to minimize memory usage by pruning of unnecessary nodes and extending the concept of sharing node information across a wide variety of queries like the basic algorithm but with lesser overhead. We evaluated the memory usage across 4 different datasets and different types of queries. The results show reduced memory usage by YCompact than any of the other techniques with no overhead involved.

We present a taxonomy in section 2 for the different types of queries for which buffering is required. We describe YFilter implementation in section 3 and describe Extensions to YFilter by Feng and Kumaran [3]. In section 4 we propose a basic algorithm which incorporates sharing of node information between queries. We extend the algorithm in section 5 to YCompact which reduces overhead in terms of data structures and provides better memory management as compared to the basic implementation. We show the results we have obtained for different query sets and data sets in section 6. Finally we summarize the work with conclusion and related work.

2. TAXONOMY

In this section, we consider a subset of XPath queries where nodes have self-predicates nested paths, self-predicates on position and multiple predicates as described in [1] and [3]. The characteristics of certain *types* of queries potentially make them susceptible to causing memory overhead. We have highlighted different types of queries for which buffering might be required and illustrate those categories with examples. We use the NASA dataset as reference for all the examples. Before we describe the queries, we need to define few terms:

We need to define few terms:

Output_node: Node, which finally needs to be output

Predicate_node(s) : Node (or a set of nodes) on which predicates are to be applied. Lets call it predicate_node(s).

2.1 Queries with no predicate_node

Such queries have no predicate_node, but only an output_node. Hence, we can start outputting the output_node, as and when we encounter it, without having to buffer it. Hence this query takes constant amount of memory: O (1).

Example: We illustrate an example that is a query with no predicate.

/datasets/dataset/title: All <title> elements that are children of a <dataset> element.

2.2 Queries with predicate_node before or the same as output_node

In this case, predicate nodes occur before or are the same as the output nodes. Hence we don't really need to buffer anything here and the memory usage is constant: O (1).

Examples: We illustrate couple of examples in which output node occurs same as a predicate node.

1. /datasets/dataset/altname [@type="ADC"]: Selects all altname children of the dataset node that have a type attribute with value ADC

2. //dataset/altname [@type="brief"]: Selects all altname children of dataset node that have a type attribute brief.

2.3 Queries with predicate_node within output_node

In this case, we should keep on buffering the output_ node, till we encounter predicate_nodes. When we encounter a predicate_node, we check if the predicate is satisfied. If predicates are satisfied, we can write out the buffered output_node, and continue directly outputting the remaining of output_node, as and when we encounter it, without having to buffer it. If the predicate_nodes are not satisfied, then we discard the so far buffered output_node. From memory management point of view, it's important to note here that after having seen all the relevant predicate nodes, we can prune out the buffered data. This is important, as we want to minimize the amount of data that needs to be stored in the system at any point of time during the execution. In other words, till the predicate is evaluated to true/false the output node has to be kept in the memory.

Example: We illustrate an example that is a query with an output node occurs before a predicate node.

//dataset/reference/source/other//title [author]: The output node title has to be buffered till the predicate author is satisfied.

2.4 Queries where predicate_node occurs after output_node

In such queries, it depends on the order in which we encounter the nodes. If we encounter predicate_node first, and output_node later, then we don't really need to buffer anything. But if we first encounter the output_node and the predicate_node later, then we do have to buffer the output_node, and wait to check if the predicate_node is satisfied.

Example: We illustrate an example that is a query where an output node is a sibling and appears before of the predicate node.

//dataset/title//source/other [author]/title: The output node title has to be buffered till predicate node author is checked to be satisfied or not.

2.5 Queries with positional operators

The position functions in the query may or may not require buffering.

Examples: We illustrate examples that are queries with a positional parameter on the output node.

1. /datasets/dataset//source/other/author [position () =2]: For this query, we don't need to buffer anything. We can directly output the 2nd author node.

2. /datasets/dataset//source/other/author [position () =last]: For this query, the last author has to be returned as an output. The last encountered 'author' child node has to be stored in memory until all the child nodes of author are checked. This is an example where buffering is needed for queries with positional parameters.

3. YFILTER

We used YFilter [1] as an environment for developing our query processing techniques. ¨*YFilter aims to provide fast, on-the fly matching of XML encoded data to a large number of interest specifications, and transformation of the matching XML data based on recipient-specific requirements.*¨ YFilter processes data at the granularity of an event level, where an event is the arrival of the start or end of an element. Standing queries, i.e. those that are continuously applied to incoming data streams, can be issued to YFilter in the XPath language.

3.1 YFilter overview

In order to explain YFilter technology [1], we need to define a few terms, namely

1. Main path: The part of the query without any predicates is the main path.

2. Nested path: The part path expression in a predicate is the nested path.

YFilter [1] explains query decomposition where the nested paths are extracted from the main path expressions and processed individually. The different stages of query processing can be described as follows:

YFilter [1] shares the processing work for multiple queries of matching the paths mentioned above by building a single NFA over all path expressions.

After the path matching by building a single NFA structure, simple predicates e.g. over attributes are evaluated. We shall discuss about the join between the main path and the nested path in section 3.3. The following example illustrates the query decomposition into a main path and a nested path.

Example: //dataset/reference/source/other//title [author]

Here, the main path is //dataset/reference/source/other//title. The nested path is //dataset/reference/source/other//title/author.

3.2 Current limitations of YFilter

Aspects of YFilter implementation that impede its use in an online setting for very long documents include:

1. Delayed processing of queries including nested paths

YFilter processes queries including nested paths only after the entire document is seen. Results should be generated on the fly, i.e. whenever conditions are satisfied for different paths of the queries.

2. Complete buffering

YFilter incrementally builds and stores the entire XML document into memory. This is unsuited for processing of large XML documents as the document can be potentially infinitely long, and storing even a fraction of the stream will quickly overwhelm the available memory.

3. No dynamic pruning

In addition to the wasteful storage of the entire document in memory, YFilter currently doesn't have functionality to prune the tree of unnecessary nodes.

3.3 Extensions to YFilter [3]

Feng and Kumaran [3] improved YFilter [1] for better memory management with two extensions. First, YFilter [1] processes queries including nested paths only after entire document is seen. This inevitably requires lots of data to be buffered until the end of the parse of the document. Feng and Kumaran [3] modified this to process nested paths on-the-fly. If multiple nested paths are satisfied for a query with single nested path, the results are written out. If a nested path is not satisfied, then they waited till the end of element tag of the last common ancestor of the output node is encountered. They used the concept of a join where a main path was joined with each nested path of the last common node in a query as illustrated in figure 1.

Join Algorithm

They implemented an n-way in-memory **symmetric hash join** algorithm that is invoked each time a main path or a nested path is satisfied. They used one hash table per main and path. There are two phases in the join, namely the probe and the build phase. In the build phase, all the anchor nodes from main path and the nested path are stored in the corresponding hash tables. In the probe phase, all anchor nodes are used in the join with no duplicates. The anchor node here is the last common ancestor in case of multiple nested paths. For example,

/datasets/dataset [keywords]/altname [@type="ADC"]

In this case, the anchor node is dataset.

Data structures

The event-based processing is performed with the help of two auxiliary data structures for each query - **a bit map and a set of multi-hash tables** in [3]. They use multi-hash tables to store information pertaining to each nested path as well as the main path. For each path, the corresponding multi-hash table stored the anchor node as the key, and the event ID as the value. To keep track of which nested paths were satisfied for a query, they utilize a bit map. The bit map contained a set of *main path* event ids along with a corresponding set of bits - one for each nested path, initialized to zero. If all sets of bits of the nested paths are set to 1 then a result is produced as all the nested paths are satisfied.

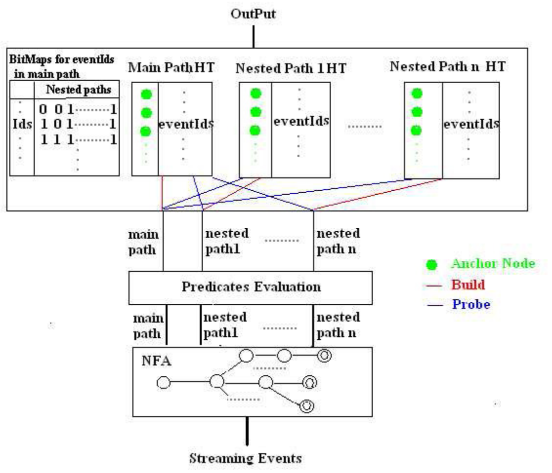

Figure 1: n-way single level symmetric hash join used in [3]

Buffer Manager

To manage memory more efficiently, they avoid storing the outputs for each event in the query hash tables. Instead, they store the unique event id associated with each event. This *query result buffer* not only avoids caching the full path information for each tuple in the hash tables but also helps them avoid storing duplicates as different queries could have the same output. Their decision to prune the contents of the hash table is motivated by the fact that once a match was found for a query, there is no need to store the associated values in memory. In addition to this, whenever an end-of-element tag of the last common ancestor is encountered; all the associated entries were cleared from the hash tables. Thus, once they identify the nodes that are required for query processing, they are able to determine which nodes to prune from memory.

However, they do not handle sharing of node information among queries. Since they store information per query, there is no technique to handle shared information among different queries. In other words, if there is node information that is shared among queries, their implementation will store all the information separately leading to extra memory usage. This will also result in delayed pruning of the nodes as there will be more information in memory than that is required.

There still persists a problem of delayed pruning in case of queries where there are multiple nested paths. The last common ancestor/anchor node has to be stored in memory till the predicates are satisfied. To summarize, the extensions to YFilter

[3] reduce memory usage by pruning of irrelevant information from memory. However, since their implementation does not handle sharing of node information among queries, memory usage is higher.

4. Basic Implementation to incorporate sharing of node information

The buffer manager of Feng and Kumaran [3] is did not handle sharing of node information across queries. If there are different queries sharing node information, then multiple copies of that information will be stored in memory till the queries are processed. The data structures that they used did not include any information on the common node information among different queries. In case of multiple nested paths, their algorithm stores the anchor node till all nested paths are satisfied. This results in delayed pruning of node information from the buffer. The algorithm mentioned in this section extends the algorithm proposed in [3] to queries where different queries share node information. By maintaining a single copy of share node information, memory usage can be reduced and queries can be processed more efficiently.

4.1 Idea

We modified the algorithm presented in [3] to include a shared buffer which holds node information that is shared across different queries. The idea was to store node information common across different queries only once and this was achieved by maintaining a single buffer containing the output nodes of each query *i*. We also maintained a separate shared buffer to store the node information that was common across all the queries. This enabled processing of the queries with better memory utilization as there is less information to be stored in memory.

4.2 Data Structures

The data structures used are a Hash Map for the shared buffer, Hash map for the pin count and Array List for the output buffer which stored the output nodes and characters for each query. The individual output node and corresponding characters are stored in the output buffer. If there is any information that is shared across queries, then a pointer from the output buffer points to the shared buffer which is linked to the pin_counter hash map by the EventID which is the key. The different data structures used with their contents are described in figure 2.

4.3 Dry Run of an example

Consider the group of queries below in order to illustrate the basic implementation on sharing of node information.

```
a/b/d[c]
/a/b/d[f]/e
/a//d[g]
```

on the input document,

```
<a>
<b><d><e>
<c>xyz</c>
<f>123</f>
<g>567</g>
</e></d></b>
</a>
```

The Output Buffer will contain:

<d> POINTER_TO_SHARED_BUFFER …<g>567</g> </d>

The Shared buffer will contain:

Shared Buffer

Event_ID	**Output_Node**
1	<e>
2	<c>
3	Xyz
4	</c>
5	<f>
6	123
7	</f>
8	</e>

The Pin_Counter will contain:

Pin_Counter(Hash Map)

Event_ID	**Pin_Count**
1	2
2	1

Thus, as seen in the example the sharing of nodes among different queries is handled over Extensions to YFilter [3].

4.4 Algorithm

The algorithm is spread across different methods startelement, endelelement and characters in EXfilterBasic.java. The buffer manager handles pruning of node information from memory. The different steps below indicate the flow of the algorithm across these different methods.

a. The output nodes which are common across queries are shared in the shared buffer. A corresponding pin count is maintained to keep track of which output node is required for which query.

b. The number of queries that share the node information (Pin_count) and the output nodes is found at the beginning of the endelement method mentioned above.

c. The output buffer has a pointer which points to the shared buffer in case of sharing data. The shared buffer has an Event_ID as a key and the Output_Node as value.

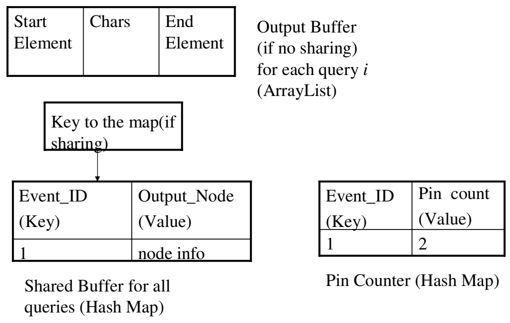

Figure 2: Basic layout of data structures in Buffer Manager

d. The output buffer is populated in the order of the start element, characters and end element. If there is no sharing of any node information among queries, then the output buffer has the Output_Node as its contents.

e. If there is sharing, then the Output_node has a pointer and the shared buffer holds all the node information.

e. The pin counter hashmap has an Event_ID as a key and a Pin_count as value. The Pin_count is used to keep track of how many queries share node information.

e. The pin_count is decremented after each query is processed. Once pin count reaches 0, the contents of the Shared Buffer are removed from the memory.

Thus, as seen from the above basic algorithm, sharing of node information among different queries is included as compared to [3]. This results in reduced memory usage as compared to YFilter [1] and Extensions to YFilter [3]. However, the use of an additional data structure for the shared buffer causes overhead and increased memory usage as there have to be buffers maintained for each query(Output buffer) and for all the queries(Shared buffer).

There is also the problem of the order in which queries are processed. In a group of 3 queries, if a query arrives before another query having 0 pin count this information will be stored in the buffer till the second query is processed. i.e. the information from the first query is not required by the second query, but is still kept in the shared buffer till the second query is processed resulting in delayed pruning from the buffer. This can be defined as a fragmentation (smaller chunks of a large chunk being shared) issue and is illustrated with an example in the next section.

5. YCompact

The YCompact algorithm acts as a superset to all the other implementations. It ensures efficient memory utilization with minimum buffering for a single query as well as a group of queries. There are issues of fragmentation and delayed pruning in the algorithm mentioned as described in section 4. There are overhead problems in maintaining an additional shared buffer. In order to handle fragmentation, after processing one query if a part of the buffer is not required by the other queries, then that is removed from the buffer.

5.1 Idea

The idea is to maintain one global buffer that the basic implementation does not provide; that holds the output node information for all the queries. In order to handle sharing of information between queries, we use the same buffer with start and end indices to keep track of where the output node of one query starts and where it ends. If there is overlap of output between queries, then we use a pin count value to prune the node information from the buffer after each query is processed. If the node information is not required by any other query, then output is flushed and buffer is cleared. After each query is matched and output is written out, we attempt to prune the buffer such that no information that is not required by any of the other queries is stored in the buffer.

5.2 Data Structures

The data structures used are Linked List as the output buffer and HashMap as the pin_counter. Pin_counter is used as a hashmap to map element_name from the linked list to the Object (start_index, end_index and pin_count_value). The main advantage of this approach over the basic implementation is that it avoids the overhead of maintaining an additional data structure. This keeps track of the output nodes in the buffer. The basic layout of the data structures is given in the figure below. The variables si, ei and pin stand for start_index, end_index and pin_count_value respectively for any query *i*. These names are described in the algorithm in the next subsection.

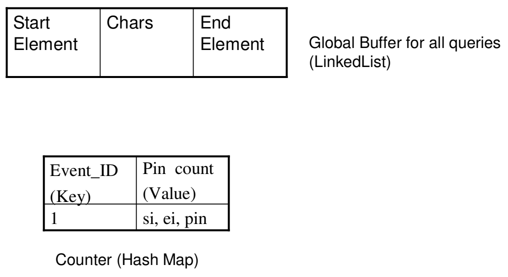

Figure 3: Basic layout of data structures in Buffer Manager

5.3 Dry run of an example

For the same set of queries and input document as mentioned in section 4.3, the data structures are populated as follows:

Global Buffer

<d>, 1, <e>,2, <c>,3, xyz, </c>,4, <f>,5, 123, </f>, 6, <g>, 7, 567, </g>,8, </e>,9, </d>,10.

Counter

Event_ID	**Si, ei, pin**
1	6, 8, 3
2	2, 12, 2
3	1, 13, 0

If query 3 is processed first, then Counter will be:

Event_ID	**Si, ei, pin**
3	1, 13, 0
2	2, 12, 2
1	6, 8, 3

Comparing next.start_index with current.start_index and next.end_index with current.end_index and a non-zero next.pin_count_value, start_index and end_index are assigned to 2, 12. The element <d> is pruned from the buffer as it is no longer required. Similarly after end of e is encountered, pruning takes place of the elements upto f and after f as they are no longer required.

5.4 Algorithm

This algorithm highlights the main steps to be handled in the different methods of EXfilterbasic.java as described in section 4. Similarly, the buffer manager handles pruning of node information from the memory. The different steps indicate the flow of the algorithm across these methods and classes.

a. Maintain a single buffer for nodes and characters i.e. start element, characters and end element. It is a linked list.

b. Add to the buffer

As an output node is encountered add it to the buffer. Use a start index to keep track of the start of the output node as well as an end index to keep track of the end of the output node for each query. In order to keep track of these indices and pin_count_value there is an object with three members, namely start_index, end_index and pin_count_value.

c. Output of the query

Check if the predicates evaluate to true or false for a query with multiple predicates. If all the predicates evaluate to true and output node is not required by any other query then decrement pin count and output the result.

If all the predicates evaluate to true and output node is required by another query, then just output the result.

If all the predicates evaluate to false, move on to the next query and wait till end of element of parent is encountered. This is done by checking the flag is_buffering_on. This is true and the output_on is false, which means that data still needs to be buffered and there is nothing to output.

d. Prune from the buffer when all predicates are satisfied

If the pin count value is reduced to zero, prune node information from the buffer. Whenever an output node is not needed by any other query, it is removed from the buffer.

If a query arrives before another query having 0 pin count, then in order to handle fragmentation pruning is done as follows:

Counter is a hashmap having name of the element as key and an Object (start_index, end_index, pin_count_value) as value. Using an iterator get the keys from the hashmap. corresponding to the keys, get the values.

If there is a case when start_index of second entry is greater than start index of the first and end_index of the first is greater than that of the second and the pin_count_value of the second entry is non-zero, then reset the start_index and end_index to the next values. Else, the pin count is zero and the fragment is not required by other queries, prune the data from the buffer. This prunes only the fragment that is no longer needed by the buffer. Also, if predicate is false, wait for the parent's end element to prune from the buffer.

Thus, from the algorithm and an example we can see that, the fragmentation issue mentioned in section 4 can be removed by keeping track of the start_index and end_index for output_node in the buffer. Thus, the problem due to delayed pruning described in section 4 is eliminated.

The overhead involved in maintaining an additional data structure in the basic implementation is also removed resulting in improved memory usage.

6 Performance Evaluations

In this section, we evaluate 4 different datasets for the YFilter algorithm [1], Feng and Kumaran's extension to YFilter system [3], Basic implementation and YCompact Algorithm. The metrics for evaluation are Memory Usage and CPU usage. The objectives of the tests are to determine if the memory usage is less in the YCompact algorithm than any of the earlier used techniques and to check whether there is any overhead involved. The tests on the different datasets are mentioned below.

We analyzed different **DTD's, datasets and query sets** to test the effectiveness of the algorithm. The query sets are described in detail along-with the data sets. The basis of selection for the query sets was based on the following criteria:
1. Manually generated queries that seem to have shared nodes between them. The degree of sharing is different across the different queries.
2. Automatically generated queries from the YFilter code base to test the memory usage for smaller/larger datasets.

The datasets were selected based common uses in literature. The details of the different datasets are mentioned in the coming subsections.

6.1 Experimental Setup

The experiments were performed on a Linux Desktop with an Intel Pentium 2.1GHz processor and 2GB of RAM. The JVM settings we used were *–Xms 500m –Xmx 1000m*. The Java version used was Java SDK 1.5. We used Profiling in Java to measure the memory usage.

The filter package in YFilter code base contains the main class EXfilter. The different algorithms were tested in different modes as described below:

1. **Result output mode:** The result was written to a file using the knob –result=ALL. The result size along with the output for each query was written to the file.
2. **Memory usage mode:** The memory usage in bytes was written to the file using the knob –result=ALL and --outputMemoryUse=TRUE. The other knob –longDocument is set to TRUE if we need to measure the memory usage of YCompact. In order to measure the memory usage for YFilter, longDocument is set to FALSE and original_streaming is set to TRUE. For extensions to YFilter [3], the longDocument is set to FALSE and symmetric_hash is set to TRUE. We used the memory usage module that was downloaded from the internet [12] by Feng and Kumaran [3].
3. **CPU usage mode:** The CPU usage was measured in Java in milliseconds using System time. The only knob to be used is --result=NONE. The --outputMemoryUse knob is set to FALSE as there is no memory usage to be measured. For the different algorithms the knobs to be set are same as mentioned in the memory usage mode.

6.2 Algorithms

We ran our tests on 4 different algorithms. The results of memory usage and CPU usage are described for each algorithm across different datasets in the section.

YFilter: YFilter handles queries such that the entire document is stored in the memory. The maximum memory usage is quite high and almost equal to the document size.

YFilter + Symmetric Hash Join [3]: Extensions added to YFilter by Feng and Kumaran [3] pruned unnecessary node information from the buffer once queries were processed.

YFilter + Basic Implementation for sharing: The basic implementation described in section 4 improves significantly over YFilter and the one suggested by [3]. However, due to overhead of maintaining an extra data structure, namely Shared buffer the memory usage is still not less.

YFilter + YCompact: The YCompact algorithm described in section 5 ensures pruning of irrelevant information from the buffer and fast execution of queries. Thus, the memory usage is minimal across different data sets and the previous algorithms.

6.3 Tests on NASA Dataset

The NASA dataset was used to conduct experiments on 4 different groups of queries. The tests were to identify characteristics of results, measure memory usage across different algorithms and find CPU usage for all the algorithms.

Data Characteristics

The NASA XML dataset is available at the University of Washington XML data repository [5]. The table below indicates the characteristics of the data used.

Table 1: Data Characteristics for NASA dataset

Name	**Size**	**Num. of Unique Elements**	**Avg/Max Depth**
NASA	25.0 MB	180	5.77/7

Query characteristics

We manually selected query sets for the NASA dataset [5]. The query sets are classified into four different groups mentioned below.

a. Query Set 1 (Group 1): In this category, queries are selected such that output node of certain queries are a subset of the output node of other queries. In this example, query 4 has an output node which is a subset of query 2. The queries are illustrated in Figure 4.

```
1./datasets/dataset[title]/author[lastName]/title
2. /datasets/dataset/title/altname/reference[name]/author
3. /datasets/dataset/title/altname/reference[lastName]/title
4. /datasets/dataset/title/altname/reference[name]/author
/lastName
```

Figure 4: Query Set 1

b. Query Set 2 (Group 2): In this category, queries are selected such that different predicate nodes act on the same output nodes. In this example, queries 1 and 2 share the same output node, whereas queries 3 and 4 share the same output node. The queries are described in the figure below.

```
1. /datasets/dataset[altname]/title
2. /datasets/dataset[reference]/title
3. /datasets/dataset[lastName]/author/title
4. /datasets/dataset[initial]/author/title
```

Figure 5: Query Set 2

c. Query Set 3 (Group 3): In this category, queries are selected such that queries have different nested paths but same main paths as seen in figure 6. In this example, queries 2 and 4 can be grouped together as well as queries 1 and 3 can be grouped.

```
1. //dataset/title/altname/reference[name]/author/initial
2. //dataset/title/altname/reference[intitial]/title
3. //dataset/title/altname/reference[name]/author
4. //dataset/title/altname/reference[lastName]/title
5. /datasets/dataset[altname]/title
```

Figure 6: Query Set 3

d. Query Set 4 (Group 4): In this case, there are 5 different queries passed simultaneously to parse over the input document. The group of randomly selected different queries is shown in Figure 8.

```
1. /datasets/dataset/reference/source/other/author
/lastName[text()="Jackson"]
2. /datasets/dataset/reference/source/journal[name][date]
/author
3. /datasets/dataset[altname]/title
4. /datasets/dataset[altname]//journal[date/year]/author
/lastName
5. /datasets/dataset[keywords]/altname[@type="ADC"]
```

Figure 7: Query Set 4

Result Characteristics

The cumulative size is an upper bound of the storage cost. It gives us the total cost of storing the node information separately for each query in memory. We calculate the percentage shared in order to observe the trends of memory usage with respect to degree of sharing across different datasets. The percentage shared is given by the formula:

No of bytes Shared/Cumulative Size * 100

Table 2: Result Characteristics for NASA dataset

Result Characteristics	**Group 1**	**Group 2**	**Group 3**	**Group 4**
Cumulative Size	**508KB**	**556KB**	**572KB**	**604KB**
Percentage Shared	40%	56%	64%	22%

Memory Usage

In this section, we analyze the memory usage across different algorithms. We measure the memory usage of the entire XPath filtering engine. We calculate the memory usage in bytes after whenever an end of element is encountered. We compute the maximum and average values for every query set across different data sets and report them in the tables below.

The memory usage shown in row 2 of Table 3 gives the maximum and the average values for YFilter implementation. Since, YFilter stores entire document in memory the maximum memory usage value is quite close to the size of the input document. An interesting observation is that the average and the maximum values increase as the size of the result (from table 2) increases.

The memory usage in [3] consists of the memory used by the Join algorithms and memory used as a result of buffering of data.

The memory usage values are shown in row 3 of Table 3. Since the basic implementation for sharing of node information and YCompact are extensions to [3], the reduction in the memory usage values are as a result of a more effective buffer manager. For the NASA dataset [5], and group 3 class of queries mentioned in section 2, the buffering cost was around 62% as compared to the join cost which was around 38%. The overall improvement of the YCompact algorithm is on this 62% as buffering is significantly improved, while the join still continues to be used in the YCompact algorithm as it is an extension of [3].

The memory usage for the basic implementation is less as compared to the previous two algorithms. This is as a result of single buffering if there is sharing of data across queries.

The memory usage for YCompact is the least as compared to the other algorithms. This is as a result of pruning of irrelevant information and maintaining a single global buffer which ensures no data duplication.

The memory usage values are sampled using a simple Java program where every 100th memory usage measurement is written to file for plotting figures 9 and 10.

Table 3: Memory Usage (KB) for NASA dataset

Memory Usage	**Group 1** **Max, Average**	**Group 2** **Max, Average**	**Group 3** **Max, Average**	**Group 4** **Max, Average**
YFilter	23929, 10966	24441, 11166	24839, 11312	24929, 11984
YFilter + SHJ[3]	3041, 966	4489, 1066	4639 1096	4829, 988
YFilter + Basic Implementation	984, 66	1034, 68	1066, 73	1208 87
YFilter + YCompact	862, 57	871, 59	898, 62	1007 76
Percentage improvement of YCompact over YFilter	99%	99%	99%	99%
Percentage improvement of YCompact over YFilter + SHJ	94%	94%	94%	92%
Percentage improvement of YCompact over Basic Implementation	11– 13%	14 - 18%	15-19%	9 – 11%

The Memory usage shown in Figure 8 shows the amount of memory used to process the 25MB NASA xml document using the original YFilter implementation for the Group 3 class of queries which is chosen as an example to represent the memory usage. As a result of no pruning and caching of all nodes, the maximum memory usage is close to the actual size of the document.

The Memory usage shown in Figure 9 shows the amount of memory used by the YCompact algorithm which is significantly lower than any of the other algorithms. The dips in the figure are as a result of pruning of data from the memory for effective memory management.

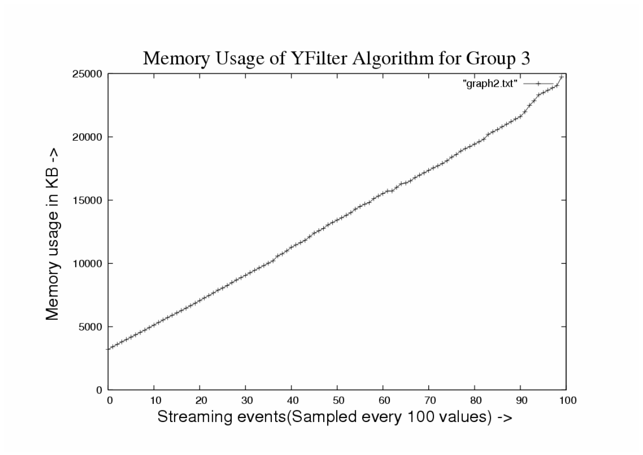

Figure89: Memory usage (YFilter) for Query Set 3 queries of NASA dataset

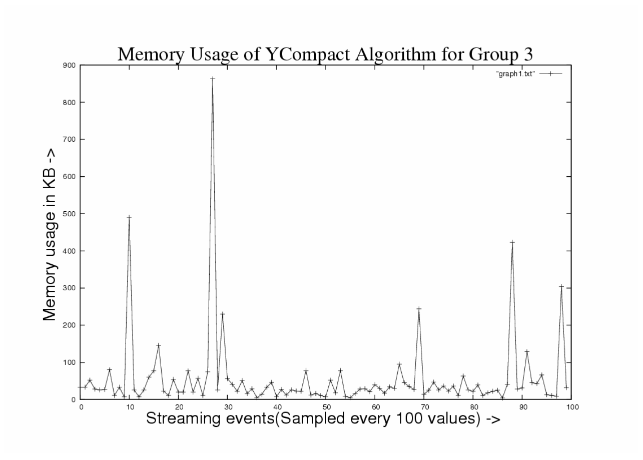

Figure 9: Memory usage (YCompact algorithm) for Query Set 3 queries of NASA dataset

CPU Usage

We ran an empty query set on the NASA dataset to measure the parsing cost incurred by YFilter in the class parser.java. The total CPU usage time consists of parsing and processing cost. The parsing cost for the NASA dataset is **5188 ms.**

The CPU usage time is calculated by disabling any I/O operations to speedup the time required to process the queries. The CPU usage of the YCompact algorithm is less than that of the original YFilter implementation. The CPU usage is less for all the other algorithms as compared to YFilter because of a couple of reasons. First, unlike YFilter which stores the entire document in memory and evaluates the nested paths at the end, all the other algorithms evaluate nested paths on the fly and generate results on-the-fly without storing the entire document in memory. Second, since YFilter does not prune any node from memory, a lot of unwanted information is stored; which adds to the processing time of the queries.

YCompact has the least CPU usage as compared to all the other algorithms as a result of handling of fragmentation and delayed pruning as described in section 5. The use of simplified data structures and garbage collection in Java enable faster processing of queries. The CPU usage time increases as the result size increases as seen in tables 2 and 4.

Table 4: CPU Usage Characteristics for NASA dataset

CPU Usage	**Group 1** (ms)	**Group 2** (ms)	**Group 3** (ms)	**Group 4** (ms)
YFilter	13144	15535	16828	18240
YFilter + SHJ [3]	12487	14308	15757	17381
YFilter + Basic implementation	11679	13481	14145	16594
YFilter + YCompact	10781	12147	12760	15406

6.4 Tests on News Article Dataset

The News Article dataset was used to conduct experiments on 2 different groups of queries. The first group consisted of queries having simple predicates. The second group consisted of a collection of mixed queries. The different metrics of evaluation are same as mentioned for the NASA dataset.

Data Characteristics

The News Article XML dataset is available at the YFilter homepage [13]. We used a concatenated dataset of 400 XML documents to test our results. The table below indicates the characteristics of the data used.

Table 5: Data Characteristics for News Article dataset

Name	**Size of Concatenated document(MB)**
News Article	0.8

Query Characteristics

The queries are selected from the YFilter homepage [13] on simple predicates and mixed queries. The two different types of query sets are described below:

a. Query Set 1 (Group 1): In this category, 200 queries on simple predicates were used on a concatenated News Article dataset of 0.8 MB. Example of simple predicates on queries is described below. In this example using the terminologies described in section 2, the predicate is id on the node body.end and bibliography is the output node.

//body/body.end[@id=12]//bibliography

b. Query Set 2 (Group 2): In this category, 1000 mixed queries were used on a concatenated News Article dataset of 0.8 MB. Using the terminologies described in section 3, mixed queries are collection of queries with single nested paths, multiple nested paths etc. An example belonging to this group of queries is described below. In this case, there are multiple nested paths, namely /*version//col and /*body/body.end/bibliography//col

/*[@version=1][body/body.end/bibliography]//col

Result Characteristics

The metrics are the same as described for the NASA dataset.

Table 6: Result Characteristics for News Article dataset

Result metrics	**Group 1**	**Group 2**
Cumulative Size	**62KB**	**234KB**
Percentage Shared	52%	64%

Memory Usage

We analyze the memory usage similar to the NASA dataset. The memory usage of YFilter is similar to the NASA dataset with the maximum value being around the size of the document.

The Memory usage of the YCompact algorithm is significantly less than any other algorithm. The Memory usage for queries on simple predicates is constant for the YCompact algorithm as the predicates are encountered before the queries in all of the queries and there is no buffering.

The Memory usage values for YCompact for simple predicates signify that a lot of memory can be saved as compared to YFilter.

The memory usage for the News Article dataset differs from the NASA dataset such that the document is small and YFilter frees the cached document at the end of the parse of each document.

The memory usage for the News Article dataset differs from the NASA dataset such that the document is small and YFilter frees the cached document at the end of the parse of each document.

The Memory usage shown in Figure 11 shows the amount of memory used to parse the 0.8MB concatenated News Article document using the original YFilter implementation for the 1000 mixed queries. The memory usage restarts from lower values after reaching the maximum value as a result of garbage collection in Java i.e. once the Java objects are no longer referenced by the program, memory is freed and again allocation takes place. The data is sampled into 100 intervals for the ease of readability.

The Memory usage shown in Figure 10 shows the amount of memory used to parse the 0.8MB concatenated News Article document using the original YFilter implementation for the 1000 mixed queries. The memory usage restarts from lower values after reaching the maximum value as a result of garbage collection in Java i.e. once the Java objects are no longer referenced by the program, memory is freed and again allocation takes place. The data is sampled into 100 intervals for the ease of readability.

Table 7: Memory Usage for News Article dataset (KB)

Memory usage	**Group 1** **MAX, Average**	**Group 2** **MAX, Average**
YFilter	746, 246	798, 428
YFilter + SHJ	328, 104	564, 230
YFilter + Basic Implementation	Almost constant ~ 2.45	228, 48
YFilter + YCompact	Almost constant ~ 2	171, 36
Percentage improvement of YCompact over YFilter	99%	92%
Percentage improvement of YCompact over YFilter + SHJ	97%	84%
Percentage improvement of YCompact over YFilter + Basic implementation	25%	22 – 24%

The Memory usage shown in Figure 11 shows the amount of memory used by the YCompact algorithm which is significantly lower than any other algorithm. Thus, an interesting observation over the previous dataset is as the number of queries increase the pattern of memory usage remains quite similar.

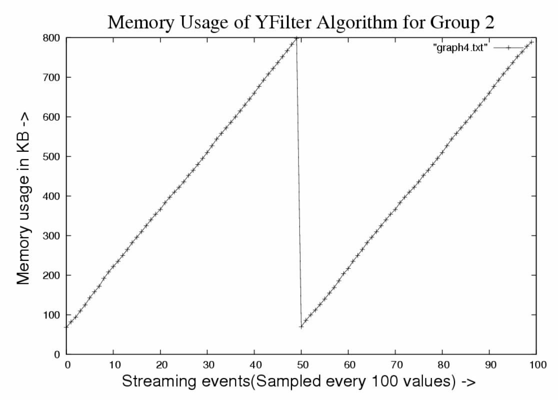

Figure 10: Memory usage (YFilter) for Query Set 2 queries of News Article dataset

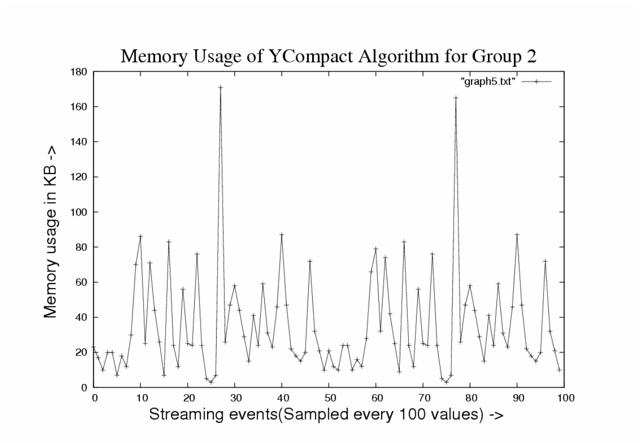

Figure 11: Memory usage (YCompact algorithm) for Query Set 2 queries of News Article dataset

CPU Usage

The parsing cost for the News Article dataset is **1063 ms.**
The CPU usage of the YCompact algorithm is less than that of the original YFilter implementation. Thus, without introducing overhead there is a significant reduction in memory usage. The CPU usage varies according to the result size similar to the NASA dataset as seen in tables 6 and 8. The other observations made for the NASA dataset hold true here too. The processing time for the queries is less if we subtract the parsing cost from the total CPU usage. This indicates that the algorithms work efficiently for larger number of queries on smaller documents.

Table 8: CPU Usage Characteristics for News Article dataset

CPU Usage	**Group 1** **(Milliseconds)**	**Group 2** **(Milliseconds)**
YFilter	2897	7826
YFilter + SHJ	2824	7762
YFilter + Basic Implementation	2798	7682
YFilter + YCompact Implementation	2712	7614

6.5 Tests on DBLP Dataset

The DBLP dataset was used to conduct experiments on 2 different groups of queries. The first group consisted of 10 randomly generated queries. The second group consisted of 100 randomly generated queries. The different metrics of evaluation are same as the ones described above.

Data Characteristics

The DBLP XML dataset is available at the University of Washington XML data repository [5]. The table below indicates the characteristics of the data used.

Table 9: Data Characteristics for DBLP dataset

Name	Size(MB)	Total Num. of Elements	Avg/Max Depth
DBLP	130.0	3332130	2.9/6

Query Characteristics

The queries are random query sets generated by YFilter for DBLP dataset [5]. The two groups consist of 10 and 100 randomly generated queries.

a. Query Set 1 (Group 1): 10 randomly generated queries were used to parse over the DBLP dataset to measure the result sizes, memory usage and CPU time.

b. Query Set 2 (Group 2): 100 randomly generated queries were used to parse over the two datasets to measure the result sizes, memory usage and CPU time.

Result Characteristics

The result metrics are the same as described for the previous two datasets.

Table 10: Result Characteristics for DBLP dataset

Result Characteristics	**Group 1** **(10 queries)**	**Group 2** **(100 queries)**
Cumulative Size	**978KB**	**2280KB**
Percentage Shared	54%	62%

Memory Usage

The Memory usage of the YCompact algorithm is significantly less than any other algorithm similar to the previous datasets. The DBLP document is around 6 times larger than the NASA dataset. The ratio of the memory usage of the YCompact algorithm to that of the basic implementation is more or less similar to that of NASA and News Article datasets as can be seen in the percentage of improvement rows.

The Memory usage for greater number of queries over the large document follows a similar pattern to the NASA dataset with highest values for YFilter and least for YCompact algorithm. The percentage improvement of the YCompact algorithm over the Basic implementation and the implementation proposed by Feng and Kumaran [3] is more or less consistent across DBLP, NASA and NEWS ARTICLE datasets.

CPU Usage

The parsing cost for the DBLP dataset is **29632 ms.** Table 12 gives the values of CPU usage for the different query sets. The CPU usage of the YCompact algorithm is less than that of the original YFilter implementation similar to the other datasets mentioned above. The CPU usage follows the same trend across different algorithms as mentioned for the first two datasets.

Table 11: Memory Usage for DBLP dataset (KB)

Memory Usage	**Group 1** **MAX, Average**	**Group 2** **MAX, Average**
YFilter(Unfinished after 30MB of document)	28678, 10091	29815, 10978
YFilter + SHJ	8041, 2098	14567, 8765
YFilter + Basic Implementation	1567, 86	4665, 374
YFilter + YCompact Implementation	1346, 73	3528, 309
Percentage improvement of YCompact over YFilter	99%	96%
Percentage improvement of YCompact over YFilter + SHJ	96%	93%
Percentage improvement of YCompact over YFilter + Basic implementation	17 – 20%	19 -22%

Table 12: CPU Usage for DBLP dataset

CPU Usage	**Group 1** **(Milliseconds)**	**Group 2** **(Milliseconds)**
YFilter	65187	104937
YFilter + SHJ	64289	103789
YFilter + Basic Implementation	63678	102145
YFilter + YCompact Implementation	62765	101876

6.6 Tests on Xmark Dataset [14]

The Xmark dataset was used to conduct experiments on 2 different groups of queries as mentioned in section 2. The first group consisted of 10 randomly generated queries. The second group consisted of 100 randomly generated queries. The different metrics of evaluation are mentioned below.

Data Characteristics

The Xmark XML dataset is available at the University of Washington XML data repository [5]. The table below indicates the characteristics of the data used.

Table 13: Data Characteristics for Xmark dataset

Name	**Size(MB)**
Xmark	114.0

Query Characteristics

The queries are random query sets generated by YFilter for Xmark dataset [14] similar to the DBLP dataset. The two groups consist of 10 and 100 randomly generated queries

a. Query Set 1 (Group 1): 10 randomly generated queries were used to parse over the Xmark dataset to measure the result sizes, memory usage and CPU time and to verify patterns of improvement

b. Query Set 2 (Group 2): 100 randomly generated queries were used to parse over the Xmark dataset to measure the result sizes, memory usage and CPU time and to verify patterns of improvement.

Result Characteristics

The result metrics are the same as described in the previous datasets.

Table 14: Data Characteristics for Xmark dataset

Result Characteristics	**Group 1**	**Group 2**
Cumulative Size	**924KB**	**1848KB**
Percentage Shared	44%	56%

Memory Usage

The Memory usage of the YCompact algorithm is significantly less than any other algorithm similar to the previous datasets.

The Xmark document is around 4.5 times larger than the NASA dataset. The ratio of the memory usage of the YCompact algorithm to that of the basic implementation is comparable to the other datasets as indicated by the percentage improvement row in the memory usage table.

The Memory usage for greater number of queries over the large document follows a similar pattern to the NASA dataset with highest values for YFilter and least for YCompact algorithm.

CPU Usage

The parsing cost for the News Article dataset is **22563 ms.**
The CPU usage of the YCompact algorithm is less than that of the original YFilter implementation. Thus, without introducing overhead there is a significant reduction in memory usage as seen across all the previous datasets.

The CPU usage follows similar patterns to all previous datasets.

Table 15: Memory Usage for Xmark dataset

Memory Usage	Group 1 MAX, Average	Group 2 MAX, Average
YFilter(Unfinished after 30MB of the document)	28962, 9208	29451. 10568
YFilter + SHJ	7568, 1872	12732, 5876
YFilter + Basic Implementation	1231, 79	4109, 298
YFilter + YCompact Implementation	1178, 63	3298, 277
Percentage improvement of YCompact over YFilter	99%	97%
Percentage improvement of YCompact over YFilter + SHJ	96%	94%
Percentage improvement of YCompact over Basic implementation	14 – 17%	17 -20%

Table 16 CPU Usage for Xmark dataset

CPU Usage for different algorithms	Group 1 (Milliseconds)	Group 2 (Milliseconds)
YFilter	40876	96782
YFilter + SHJ	38989	95678
YFilter + Basic Implementation	37243	94234
YFilter + YCompact Implementation	36244	93236

7 Related Work

We use YFilter [1] which is an environment for developing our query processing techniques. There are some extensions to YFilter already in place by Feng and Kumaran[3]. They identified limitations of YFilter where the entire document was being stored in the memory which led to buffering overhead. However, they did not check for sharing of node information among queries.

There has been some work done in XPath queries on streaming data using XSQ. A queue for buffering relevant data from a given XPath query was used as a buffer as described by Peng and Chawathe [2]. Buffering of XPath queries in order to store potential results was studied by them. In their analysis, the items in the buffer are marked separately so that after evaluation of predicates only those items that are affected by the predicates are evaluated.

Koch et al[6] developed a language called Flux with an algorithm for automatically translating a significant part of XPath into Flux queries. The main contribution of this paper is the Flux language together with an algorithm for automatically translating a significant fragment of XPath into equivalent Flux queries. Flux – while intended as an internal representation format for queries rather than a language for end-users – provides a intuition for buffers-conscious query processing on structured data streams. The algorithm uses schema information to schedule Flux queries so as to reduce the use of buffers.

Barton et al[7] developed a novel streaming algorithm for evaluating XPath expressions that use backward axes and forward axes of an XML document. The performance of their algorithm is efficient as well as shows memory utilization by retaining only relevant portions in memory. Li et al [8] developed a series of optimizations which transform Xpath queries so that they can correctly execute within a single pass of any dataset. This involved buffering of information. Fegaras et al [11] dealt with the buffer size and client's ability to optimize queries effectively. They presented a framework for processing streamed XML data and an algebraic optimization framework.

Guo and Chirkova [9] devised efficient algorithms for passing queries over large input documents. They measured the response times. FisT [10] provides a XML filtering system. FisT[10] performed holistic matching of twig patterns with incoming XML documents by transforming into Prufer sequences.

Feng and Kumaran[3] also discusses a taxonomy for XPath queries to gain insight into memory utilization of queries, as they implement execution of XPath queries on streaming XML data over the framework of YFilter, but our taxonomy presents facilitating theoretical memory utilization analysis of concurrent queries along with implementation ensuring minimum memory consumption. As seen in the related work, there has been lot of emphasis on faster processing of queries apart from Flux [6]. Our approach proposes a novel YCompact algorithm and compares memory usage and CPU usage across different datasets for different query sets.

8. Conclusion and Future Work

We have devised techniques that produce real time outputs with significantly less memory usage and no overhead over the existing YFilter implementation. The results show efficient memory management for various different datasets using different classes of queries. The interesting question is that whether these improvements to YFilter will still hold for different types of Xpath queries on a real system with memory and throughput constraints. Also, it could be possible to come up with a theoretical bound on amount of memory usage for different types of queries.

ACKNOWLEDGMENTS

I would like to thank Prof. Yanlei Diao for her continuous guidance throughout the project. I would also like to thank Prof. Gerome Miklau for being the second reader for this work.

References

[1] Y. Diao, P. M. Altinel, M. J. Franklin, and P. Fisher. Path Sharing and Predicate Evaluation for High- Performance XML Filtering. In TODS, 2003.

[2] F. Peng and S. S. Chawathe. XPath queries on streaming data. In SIGMOD ’03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pages 431–442, New York, NY, USA, 2003. ACM Press.

[3] S. Feng, G. Kumaran. XML Data Stream Processing: Extensions to YFilter. UMASS Technical Report, 2007.

[4] Filtering and Transformation for High-Volume XML Message Brokering. Online version at, http://yfilter.cs.umass.edu/

[5] University of Washington XML Repository, http://www.cs.washington.edu/research/xmldatasets/

[6] Cristoph Koch, Stefanie Scherzinger, Nicole Scweikardt and Bernhard Stegmaeir, Schema-based scheduling of event processes and buffer minimization for queries on structured data stream. In 30th VLDB conference' 04.

[7] Charles Barton, Phillipe Charles, Deepak Goyal, Mukund Raghavachari, Marcus Fontoura and Vanja Josifovski, Streaming XPath processing with forward and backward axes. In 19th ICDE'03, page 455.

[8] Xiaogang Li and Gagan Agrawal, Efficient evaluation of Xquery over streaming data, In VLDB'05.

[9] Gang Gou and Rada Chirkova, Efficient algorithms for evaluating XPath over streams, In Proceedings of the 2007 ACM SIGMOD international conference on Management of data

[10] Jonho Kwon, Praveen Rao, Bongki Moon and Sukho Lee, FisT: Scalable XML Document Filtering by Sequencing Twig Patterns, In VLDB'05

[11] Leonidas Fegaras, David Levine, Sujoe Bose and Vamsi Chaluvadi, Query processing of streamed XML data. In Proceedings CIKM'02

[12] Java memory Usage module, http://www.javaworld.com/javaworld/javaqa/2003-12/02-qa-1226-sizeof.html#resources

[13] YFilter homepage, http://yfilter.cs.umass.edu/

[14] Xmark dataset, http://www.xml-benchmark.org/