XML Data Stream Processing: Extensions to YFilter

Shaolei Feng and Giridhar Kumaran

January 31, 2007

Abstract

Running XPath queries on XML data steams is a challenge. Current approaches that store the entire document tree in memory are more suited for static environments. We modify the existing YFilter architecture to support online execution of XPath queries. This is achieved by minimal buffering of the data stream and dynamic pruning of the document tree in memory. These modifications enable us to efficiently process XPath queries on streaming XML data.

1 Introduction

XML data streams [2] have become ubiquitous modes for information exchange over the internet. Examples abound from streaming news to stock market updates. More recently, 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.

To understand the unique aspects and requirements of XML data stream processing it is illustrative to compare it with a traditional DBMS (Figure 1).

In a traditional DBMS, the data is more or less static and stored in some medium to allow easy querying and updates. The queries are intermittent or periodic and can vary to cover a wide range of information needs. The entire process can be thought of as being query-driven, i.e. humans generate different queries about the stored data. However the streaming data environment is data-driven – static queries are used to track or analyze unending streams of data. The answers generated for a particular query can quickly become obsolete or redundant in a very short period of time. The limits on the amount of data that can be cached also puts constraints on the type and nature of queries that can be posed.

Queries on streaming data are specified in languages like XQuery. The read-once nature of streaming data as well as XQuery features like predicates and closures make certain queries very memory-intensive. The need to buffer the stream arises when a decision about one target item (a node or a set of nodes in the query) requires more information about other necessary conditions (e.g. aggregations, predicates). This need for additional storage often leads to unacceptable buffer overheads.

Our contributions include changes to the execution model to support online evaluation of queries. We also incorporated memory management techniques that ensured minimal memory usage. We summarize our contributions below.

1. Query Evaluation

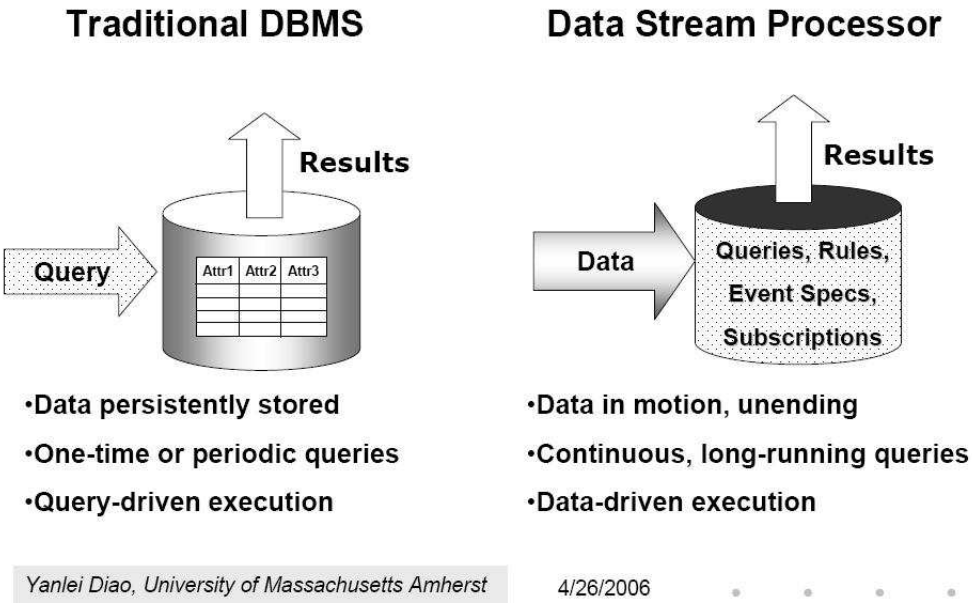

Figure 1: Comparison between a traditional DBMS system and a data stream processor

- Modified YFilter to perform event-based predicate processing
- Changed nested path processing from end-of-document to event-based
- Output results after every match.

2. Memory management
   - Included new data structures to perform n-way hash joins
   - Dynamically pruned the structures in memory to attain optimal memory usage
   - Included a memory measurement module to track memory usage

2 Analysis of memory usage

The characteristics of certain *types* of queries potentially makes them susceptible to causing memory overhead. To understand and study the reasons why certain queries cause memory overhead, we used a single large XML document to simulate a data stream. By running different types of queries we were able to obtain an understanding of the reasons for memory overhead. Our investigations enabled us to create a taxonomy, by no means complete, of the types of queries and their memory usage characteristics. These observations helped inform the techniques we developed to handle XML data streams.

We provide below the taxonomy we created along with illustrative examples of queries that led to memory overhead.

1. Self-predicates on text nodes
   - Queries with conditions on predicates

Example: /author[child::text()=”Heckmann”]
Analysis: This query is designed to find author nodes that have children text nodes and at least one of these children is ”Heckmann”. In a procedure similar to that for the previous query, the system has to store the target node in memory until it finds such a child.

- Queries with aggregation operators
Example: /title[pageno>10]/author
Analysis: The system has to store all the authors until it finds the child node ’pageno’ with value larger than 10.

2. Self-predicates on the position

   - Queries with positional operators
   Example: /para[position()=last()]
   Analysis: This simple query asks the system to return the last paragraph. Until the system has checked all the child nodes of the context node, it has to store the last encountered ’para’ child node in memory. This is an example of position functions in the query requiring buffering.

3. Nested Paths

   - Queries with nested paths
   Example: /title[publisher]
   Analysis: This query has a predicate on node ’title’, and requires that the returned node have a child node called publisher. This query represents a class of queries which requires a specific child node for the target item. In this case, the system must store the target node(s) in memory until it finds an appropriate child.
   - Queries with multiple nested paths
   Example: /title[publisher]/field[definition]
   Analysis: This represents a class of queries that require multiple hash joins in an online fashion. This potentially expensive operation can slow down the response time of the system.

4. Descendent-or-self operators

   - Queries with multiple matches
   Example: //reference//title[date]//author
   Analysis: This query has both descendent as well as predicate specifications. In addition to checking each matching path, the system has to store the target item and check the child nodes of the *title* node.

In summary, we can observe that predicates on text nodes, predicates on positions, nested paths, and descendent-or-self operators are XQuery features that potentially lead to buffering overheads. The main constraint in our proposed solutions to the buffering problem is the single-pass nature of XML streaming data. While we cannot do anything about the document structure itself, we can attempt processing techniques that will mitigate the amount of data from the stream we need to buffer.

3 YFilter

We used YFilter [1][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 XQuery language.

[1] `http://yfilter.cs.berkeley.edu/`

3.1 Current limitations of YFilter

YFilter ver. 1.0 has no support for buffering large amounts of streaming XML data, and is thus inefficient for the classes of queries we have discussed in the Section 2.

Aspects of YFilter implementation that impede its use in an online setting include

1. Complete buffering

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

2. Delayed processing of queries

   Predicate processing and system output is done only after the entire document has been observed - something unacceptable in an online environment. Results should be generated on the fly, i.e. whenever conditions are satisfied for queries.

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.

4. Measuring throughput

   YFilter also doesn't have modules to measure throughput and memory usage which are essential to testing the quality of potential solutions to handling data streams.

Our goal was to add functionality to YFilter so that it can be deployed effectively and efficiently in an online environment.

4 Execution Model

In YFilter ver. 1.0, predicate and nested-path processing was performed after the entire document tree was built in memory. To enable online processing, we introduced event-based execution. An event was triggered every time a closing tag was encountered. Such event-based processing also enabled us to keep track of the nodes that were required in memory for query processing. Once we identified the nodes that were required for query processing, we were able to determine which nodes to prune from memory.

1. Data structures

   The event-based processing was performed with the help of two auxiliary data structures for each query - a bit map and a set of multi-hash[2] tables. YFilter constructs an NFA for the complete set of input queries. Each input query is partitioned into a set of path steps consisting of the *main path* and every *nested path*.

   We used 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[3] as the value.

[2] Single key, multiple values

[3] An event ID was assigned to each event as it occurred

To keep track of which nested paths were satisfied for a query, we utilized 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

This setting provided the basis for our implementation of a n-way one-level hash join algorithm that was invoked each time an event occurred. Figure 2 depicts the complete architecture. The portion of the diagram depicting the bit map and hash tables is the set of data structures for *one* query. Additional queries will each have their own bit maps and hash tables.

2. Pseudocode

Each time an event is triggered, the corresponding tuple was passed through the YFilter NFA to determine if it matched any path step of any given query. If the tuple led to some accepting states being reached in the NFA, a set of eventID/queryID pairs were generated. The next step was to check if all the predicates of the accepting pats were satisfied. In case the predicates were satisfied, we continued with the following pseudocode. The implementation is included in the file `SHJforNestedPaths.java`.

Algorithm 4.1: HASHJOINFORNESTEDPATHS($queryIDs, eventID$)

```
for each queryID
  do
  comment: Update the data structures
  if path is the main path
    then { for each branching level
             do insert <branching level node,eventID> pair into main path multi-hash table
           for each nested path
             do insert <anchor node,eventID> pair into corresponding nested path multi-hash table
           create new entry in query bitmap with
           key ← eventID
           value ← bitset of size = |nested paths|, initialized to 0
    else { insert <anchor node,eventID> pair into corresponding nested path multi-hash table

  comment: Check if query is satisfied
  if path is the main path
    then { get bitset corresponding to eventID
           for each nested path
             do
           { if anchor exists in corresponding nested path multi-hash table
               then set corresponding bit in query bitset to 1
           if no. of bits sets to 1 in bitset = no. nested paths
             then output result and prune memory
           comment: See memory management in next section
    else { eventIDList ← main path multi-hash table[nested path anchor node]
           if |eventIDList| ≠ 0
             then for each e ∈ eventIDList
                  do { get bitset corresponding to e
                       if no. of bits sets to 1 in bitset = no. nested paths
                         then output result and prune memory
```

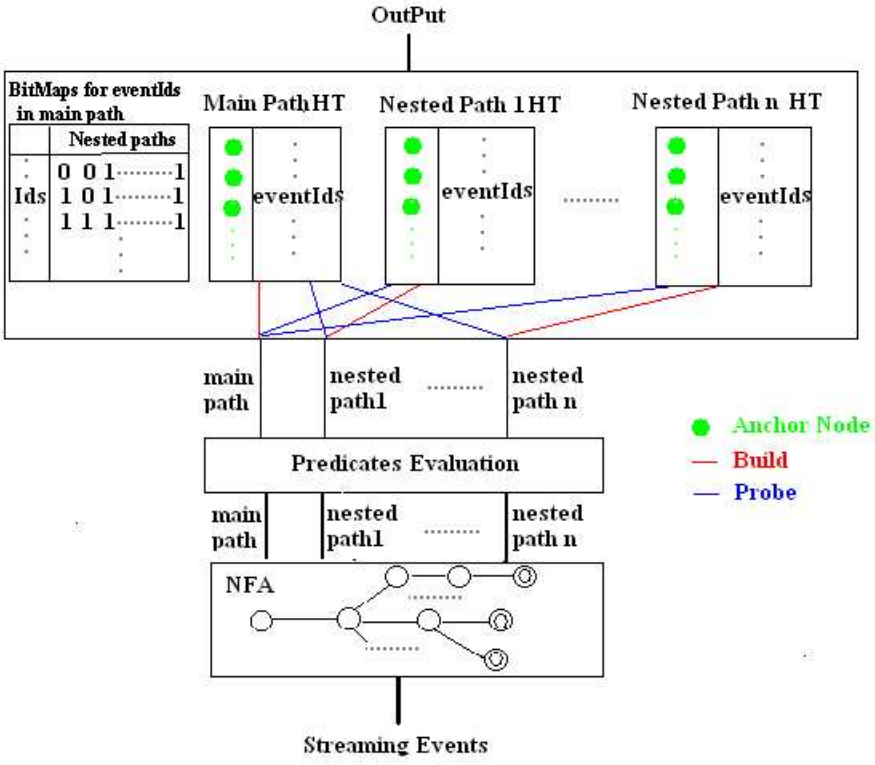

Figure 2: n-way single-level symmetric hash join

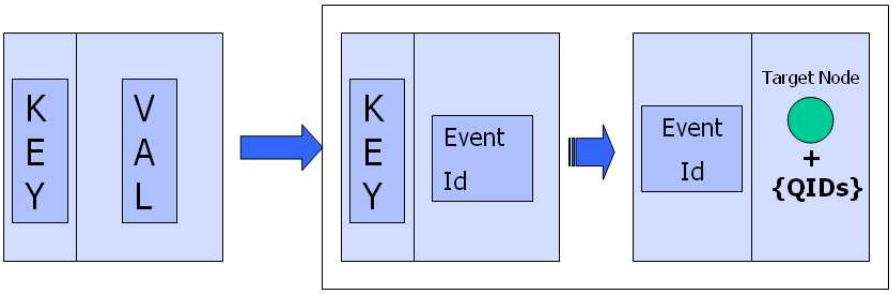

Figure 3: The global hash map to store output nodes

5 Memory Management

The required information stored in memory included not only the hash tables for the main path and nested path, but also the potential set of results associated with each event. The latter was necessary for returning the results when a query was satisfied. We present more details about the memory management techniques in following sections. Our first modification to YFilter involved disabling materialization, i.e. we disabled the creation of the entire XML document tree in memory. A comprehensive representation of the document was not suited for online processing of queries[4].

5.1 Result cache

1. Structure

   To manage memory more efficiently, we avoided storing the outputs for each event in the query hash tables. Instead, we stored the unique event id associated with each event and used a global hash table (Figure 3) to map the event ids in the query hash tables to target nodes *and* queries. This *query result buffer* not only avoided caching the full path information for each tuple in the hash tables but also helped us avoid storing duplicates as different queries could have the same output.

5.2 Hash Tables for Main and Nested Paths

1. Structure

   We used multi-hash tables for storing eventID and node information as reported in Section 4.

2. Pruning Strategy

   Each time a query was satisfied by an event, we removed the <key,value> pair associated with that eventID from the main path multi-hash table. Our decision to prune the contents of the hash table was motivated by the fact that once a match was found for a query, there was no need to store the associated values in memory.

   In addition to this, whenever an end-of-element tag was encountered, all the associated entries were cleared from the hash tables. This was valid as all the matching information about the tuples in the main path was recorded on the bit sets. These steps resulted in a significant reduction in the amount of data cached in memory.

[4]When materialization was left on, we couldn't even load the NASA data set, let alone process it

Name	Size(MB)	Test Size (MB)	Num. of Elements	Avg/Max Depth	Average Tag Length
NASA	25.0	4.94	180	5.77/7	5.03

Table 1: Characteristics of the NASA data set

5.3 Memory Usage Module

To determine the amount of memory used by the system, we measured the size of the query hash tables at the beginning of each event.

We adapted a Java module measuring memory usage downloaded from the internet [5].

6 Experimental Setup

We simulated streaming XML data by choosing a very large XML file, and running the online version of YFilter on it. Our online version of XML returning results as it processed the XML serially. We chose the NASA XML data set file available at the University of Washington XML Data Repository. The characteristics of the data set are tabulated in Table 1.

Our experiments were performed on an IBM ThinkPad^{TM}laptop with an Intel Pentium M 1.5GHz processor and 512MB of RAM. The JVM settings we used were *-server -ea -mx100m -ms20m*.

7 Results

To test our solutions we used the NASA XML data set (25 MB) available at the University of Washington XML Data Repository. We wrote our own queries to test the various classes of queries we have described in Section 2. We conducted a series of tests to verify the modifications we made to YFilter. The objectives of the tests were to determine if the results were output in an online fashion and the effectiveness of memory management techniques we incorporated.

To test the online nature of the system, we selected a query that didn't require buffering. For such a query, we would expect not only online output of results, but also constant memory usage. Figure 4 is the memory trace for the query */datasets/dataset/altname[@type="ADC"]*

Following this, we compared the performance on two queries – *//other/title* and *//other[publisher]/title* , i.e. one without predicates and one with predicates. We would expect the former to result in constant memory usage and the latter to cause buffering overhead. Figures 5 and 6 tracks the memory used by these query, and validate our hypothesis. The memory usage varies in the query with predicates, indicating that buffering is required for this query. The dips in memory usage indicate that pruning is being done to avoid storing unnecessary nodes.

We further ran two queries */datasets/dataset[keywords]/altname[@type="ADC"]*
and */datasets/dataset/altname[@type="ADC"]* separately. While the second query did not require buffering, the first did (Figure 7). The reason for the peaks in memory usage was that we needed to buffer the entire anchor node (dataset) until all the conditions are satisfied. Further analysis revealed a design oversight on our part while determining the *key* values for the query

[5] `http://www.javaworld.com/javaworld/javaqa/2003-12/02-qa-1226-sizeof.html#resources`

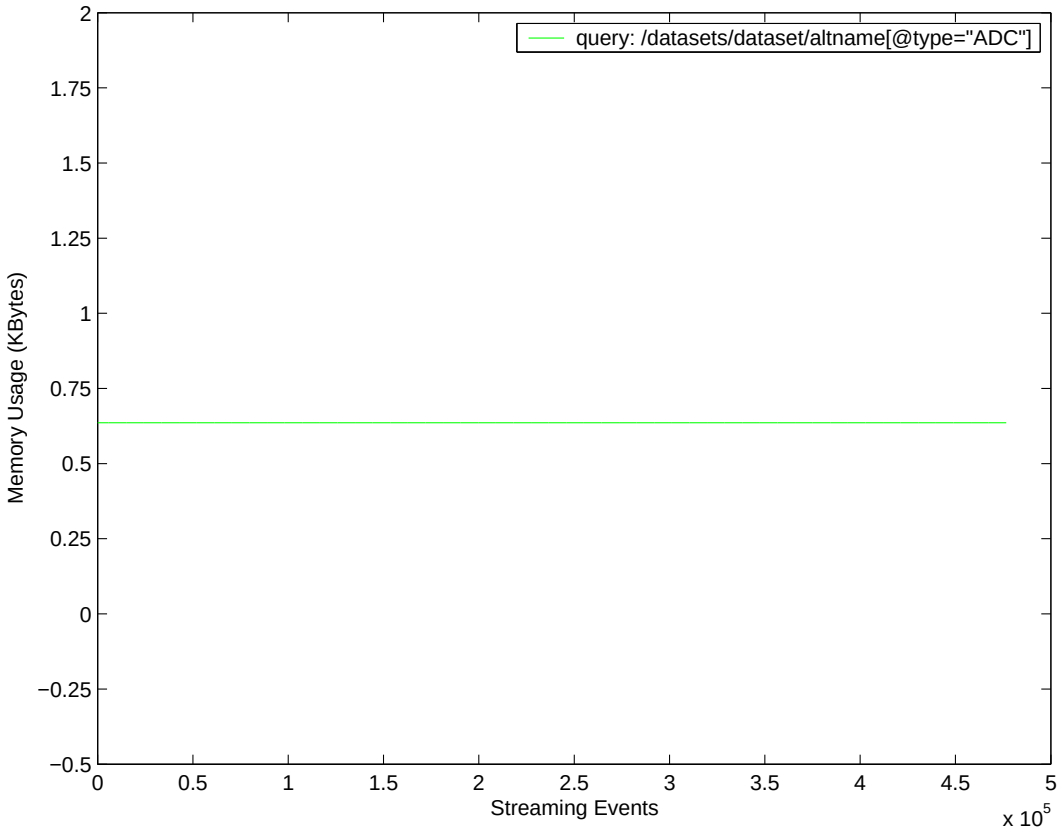

Figure 4: The memory usage is constant, indicating that the output node is returned as and when it occurs in the stream - there is no need for buffering due the absence of nested paths

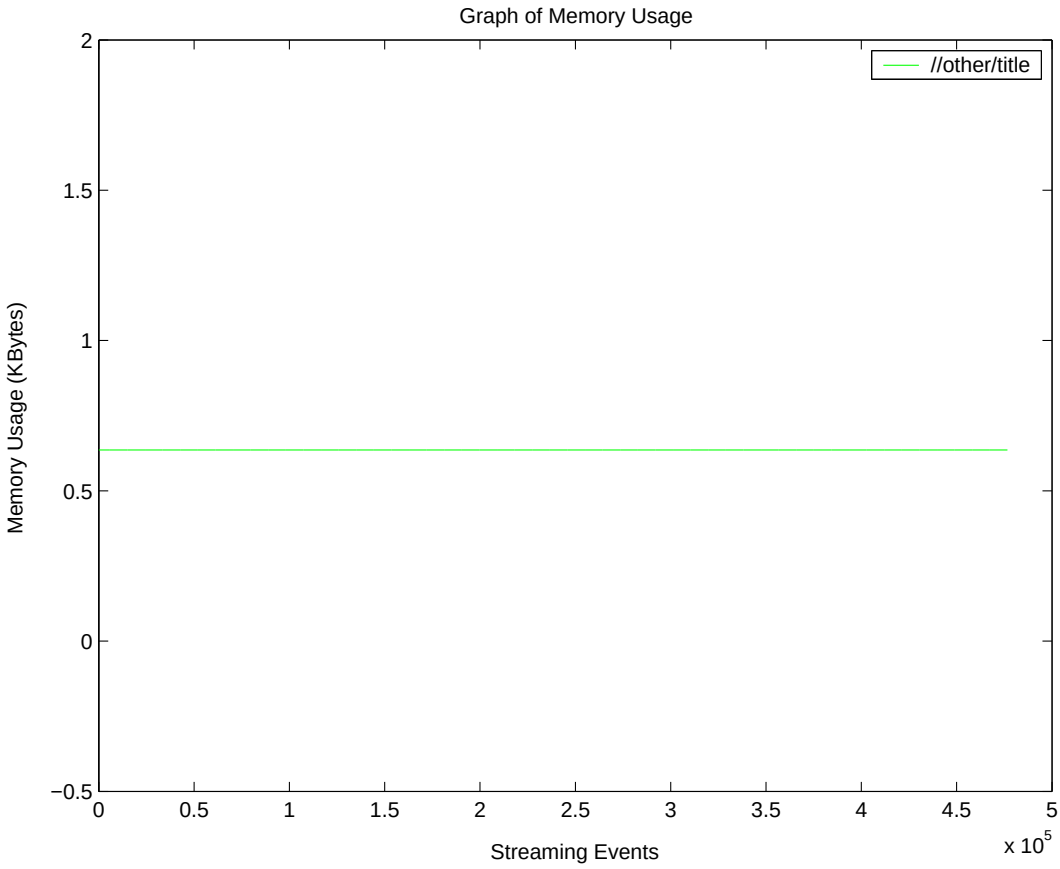

Figure 5: A query without predicates

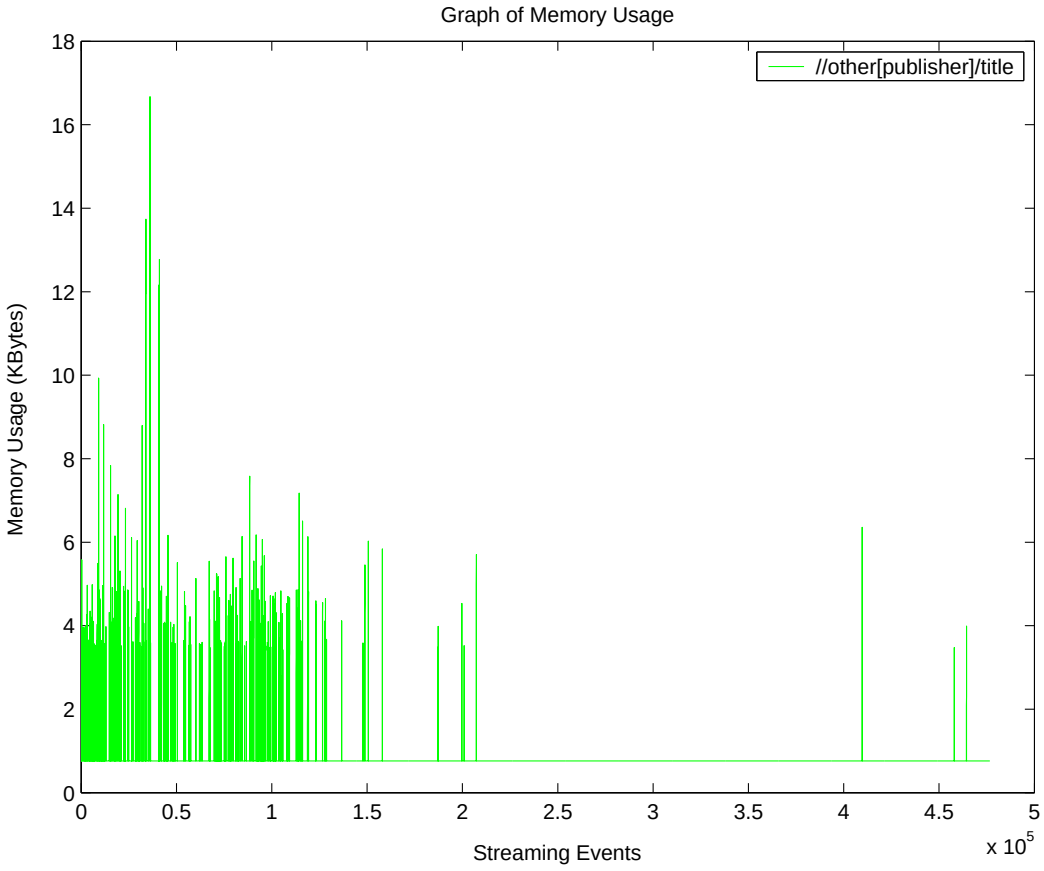

Figure 6: A query with predicates

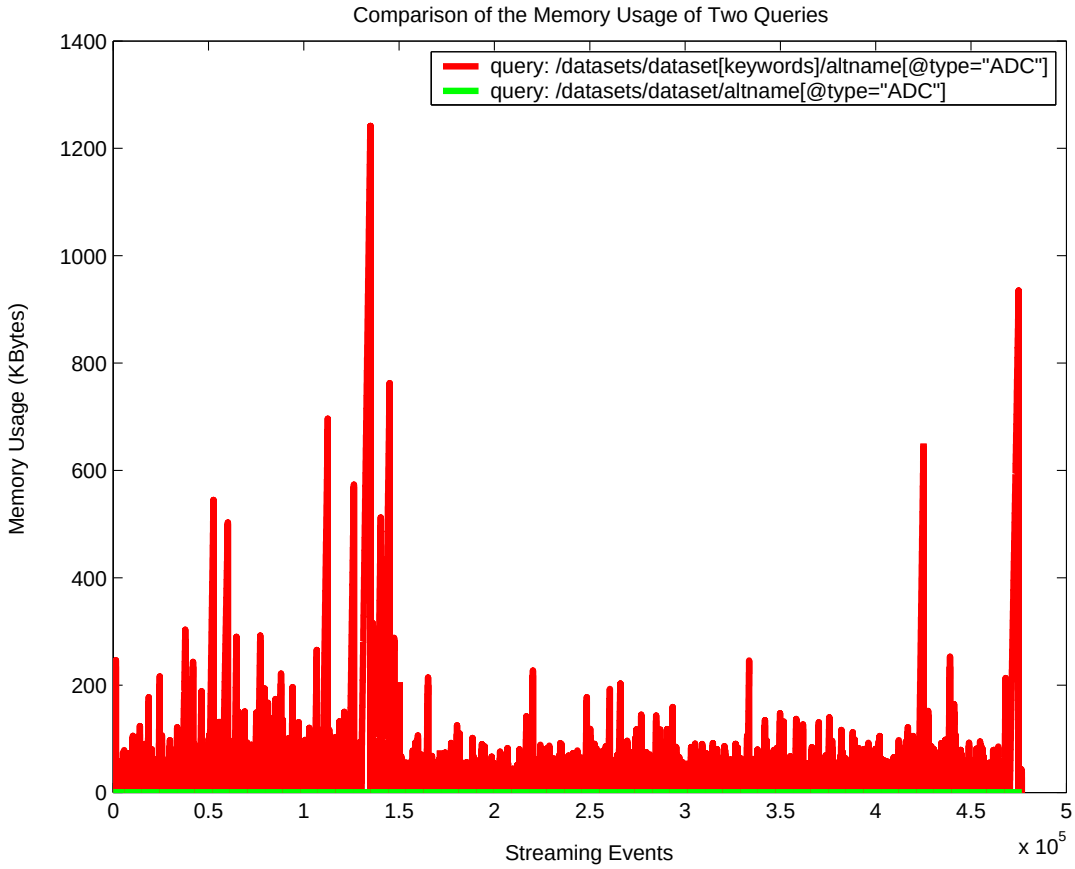

Figure 7: While memory usage is constant for one query, the usage for the second varies as expected

hash tables. We used the anchor nodes (represented as entire node sets in the YFilter code; `ParsingContext`) as keys for the query hash table. In this particular query, the node set for the node *dataset*, which has 1200 children occupied close to 1 MB in memory and 215 KB on disk. This observation provided us an opportunity to further improve our memory management techniques as part of future work.

Since YFilter is designed to handle multiple queries simultaneously, to stretch the system further, we tested on a set of four different queries and observed the total memory usage. The results are shown in Figure 8.

8 Conclusions

We have devised extensions to YFilter to enable it to operate in a streaming data environment. Using a novel hash join algorithm and simple memory management techniques we have provided YFilter the capability to produce real time outputs to sets of queries without the hindrance of memory overflow.

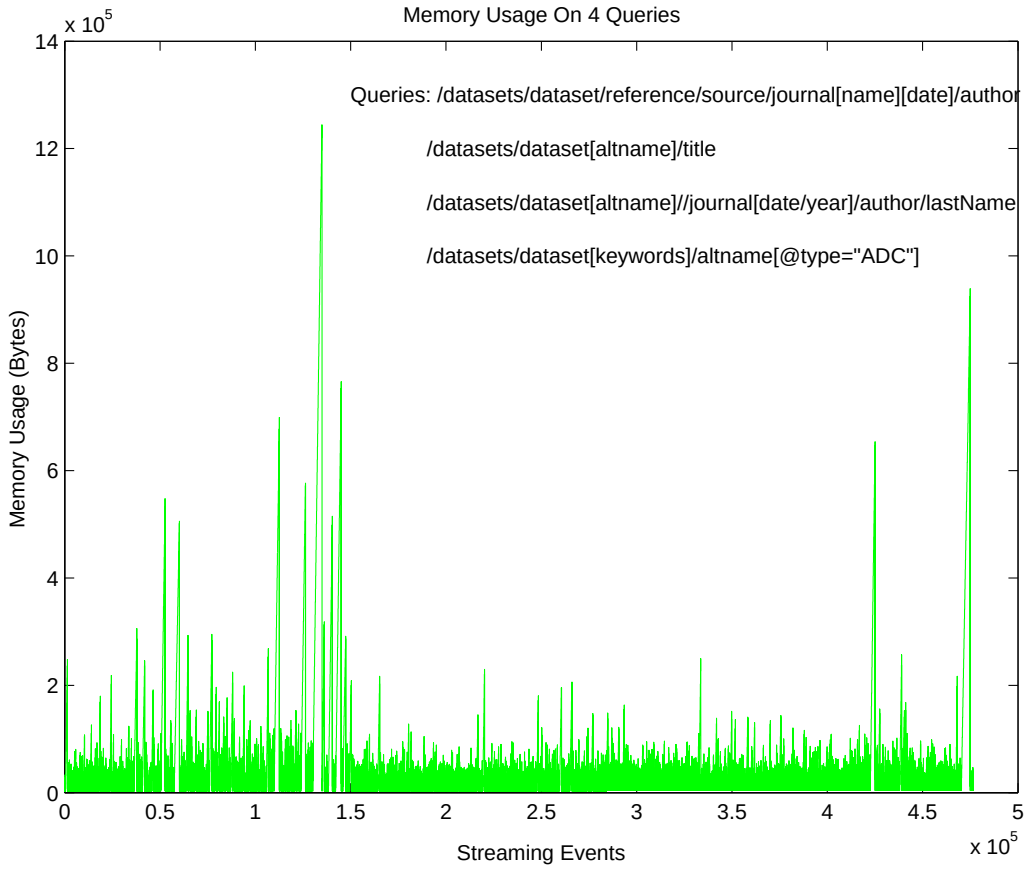

Figure 8: Total memory usage statistics for a set of four queries

9 Limitations

While we developed a taxonomy of the query types that led to memory overhead, our list is far from exhaustive. Given the limited scope of our experiments, we can only hypothesize that the techniques we have developed will extend to other query types as well.

Our memory management techniques make use of standard insert and delete operations on data structures available as part of the JVM. We do not explicitly manage memory usage - that is left to the Java garbage collector. We approximated memory usage by measuring the size of the hash tables and bit maps using third-party software.

10 Future Work

The discovery of the use of inefficent key value is one of the first of a set on improved memory management techniques we plan to pursue. Further testing of the system on a broader range of queries and steaming environments is warranted.

Acknowledgements

This work was supported in part by the Center for Intelligent Information Retrieval. We thank Prof. Yanlei Diao at the University of Massachusetts Amherst for her guidance through this work.

References

[1] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. Yfilter: Efficient and scalable filtering of xml documents. In *ICDE*, pages 341–, 2002.

[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.