

Optimizing XPath Queries on Streaming XML Data

Keerati Jitrawong Raymond K. Wong

University of New South Wales
and National ICT Australia
NSW 2052, Australia

wong@cse.unsw.edu.au

Abstract

XML stream processing has recently become popular for many applications such as selective dissemination of information. Several approaches have been proposed and most of them are based on the idea of finite automata. Different from these approaches, this paper presents a novel and efficient method for evaluating XPath with predicates on XML streaming data. For linear XPath expressions, our approach is at least as fast as the best method to date, i.e., with the cost of $O(1)$ for each SAX event. For XPath with predicates, experiments have shown that our approach is efficient and scalable.

Keywords: DFA, Stream Data, XML, XPath Optimization.

1 INTRODUCTION

The popularity of XML as a standard for information representation and exchange has created a wave of new applications as well as challenges. In particular, due to the demands from sensor network applications [20], information dissemination [1], content based routing [28], and processing of scientific data [23], efficient evaluation of XPath expressions on XML stream data has attracted lots of attentions very recently. Most of these recent proposals are based on some form of finite automata and can be categorized into approaches either based on Nondeterministic Finite Automaton (NFA) or Deterministic Finite Automaton (DFA). In general, although DFA-based approach uses constant processing time independent of the XPath query workload, there are no space guarantees. Alternatively NFA-based approaches can guarantee space requirements, but they require more processing time. Example NFA-based approaches include XFilter, YFilter, and XTrie [1, 9, 4], and DFA-based approaches include Lazy DFA, and the XPush Machine [15, 17]. Those approaches represent XPath queries as finite automata, which can then be used to process against the incoming streaming XML data. Out of these

approaches, the recent Lazy DFA [15] proposal addressed some of the above issues. First, it showed that the number of DFA states is small and will only be constructed lazily at run time. We observed from their result that the structure of input XML is usually small even though its schema may allow data instances with infinite structures. Based on their results and this observation, this paper proposes a novel approach by first extracting the structure of input XML documents and then using it for preprocessing XPath query workload. We term this special structure the *Structure Index*, due to the reason that the XPath queries are preprocessed according to the structure of the documents. While having better scalability, this approach is also based on the structure of the input XML documents, it thus will share some basic characteristics of the DFA-based approaches. Therefore, we also analyze and discuss the DFA-based approaches in detail in this paper.

Our contributions can be summarized as follows:

- We propose a novel approach based on the *Structure Index* which can be used to efficiently evaluate a large number of XPath queries on XML streaming data. We analyze the complexity of our approach both in time and space, and also verify it by experiments. After its short warm-up phase, it achieves constant processing time per SAX event independent of the query workload for linear XPath expressions.
- We propose a method called *Trigger Tree*, which is used to augment the Structure Index to efficiently evaluate XPath expressions with nested paths. This method is scalable with respect to both the number of nested paths and the number of value-based predicates when the queries have at least one value-based predicate. The central idea behind is to utilize the selectivity information of the value-based predicates.
- We provide detailed analysis for the DFA-based approaches, which has not been revealed in previous papers.

2 RELATED WORK

Related work on evaluating XPath expressions on XML data streams can be classified into DFA-based approaches and NFA-based approaches. We first discuss some recent NFA-based approaches below, followed by DFA-based approaches and other alternatives.

YFilter [9], a successor of XFilter [1], improves XFilter by using path sharing concept. It also separates the filtering problem of XPath expression into structure matching (using path expressions) and content matching (using value-based predicates). However, its performance still depends on the query workload.

XTrie [4] was designed to support large-scale filtering of streaming XML data. Unlike XFilter, XTrie supports complex XPath queries with predicates. Its basic idea is to use the *trie* to detect occurrences of substring matches for each event that it receives.

For DFA-based approach, the Lazy DFA [15], the XPush machine [17], and the recent work from Onizuka [26] (which is an improvement of the lazy DFA approach) are summarized below.

The XPush machine extends the lazy DFA approach to handle complex queries with nested paths using deterministic pushdown automaton in a bottom up fashion. Although this approach uses constant processing time in theory, it hardly achieves its theoretical performance due to its huge memory usage, and its efficiency in practice is about linear with the size of the query workload.

The recent work by Onizuka consists of two parts. The first part focuses on using lazy DFA with document-oriented XML, which usually has a complex schema. This class of data usually causes a problem in the lazy DFA approach because of its large data guide [14]. Onizuka clustered a large query workload into a number of smaller query workloads. Although this can reduce the soft upper bound of lazy DFA, it does not reduce the hard upper bound. Thus, the upper bound of lazy DFA still depends on a query workload and can be large. Therefore, this clustered lazy DFA approach can hardly achieve its stable phase, which can be observed from its experimental results.

Other approaches such as WebFilter [12] uses different approach based on the technique in [11], which treats an XML document as a set of attribute value pairs and an XPath expression as a collection of predicate pairs. [16] uses views of XML documents to speed up the processing time, which means that it needs to augment the query processor. Those approaches are interesting options and need further investigation to be compared with automata based approach.

3 THE STRUCTURE INDEX

In this section, we introduce the Structure Index, which is document structure derived from the input XML documents. It can be used to preprocess the structure navigation part of XPath queries. At runtime, the Structure Index can be used to efficiently find queries that match a given XML document by traversing its structure, and perform additional computation for predicate evaluation.

Example 3.1 Given two linear XPath queries:

Q1 = /a/*/*c//d[text()='a2z']

Q2 = //d[text()='t']

This example will be used as an example in subsequent sections.

3.1 Document Structure

Document structure is a minimal structure of an XML document which is sufficient to answer structure navigation part of an XPath expression. It ignores element values, attribute values, duplicate elements, and document order since those information are not essential for structure navigation part. This is similar to an index structure for XML data such as DataGuides [14], the Index Fabric [7], and ViST [31]. However, those index structures have a goal to index the data to facilitate efficient query processing, while our document structure has a goal to extract the structure of data that can be used to preprocess queries. An example of an XML document and its document structure is shown in Figure 1.

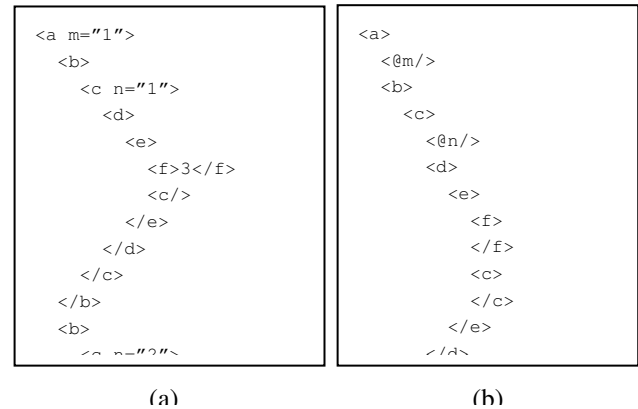


Figure 1. An XML document (a) and its document structure (b).

3.2 The Structure Index and Preprocessing

Structure Index is constructed by representing each element or attribute node as a node in a Structure Index, called *Index Node*. The relationships between Index Nodes are the same as parent-child relationship in nodes of document structure. This index will be used (as an XML document tree) to preprocess the structural navigation part of XPath queries. It will attempt to find all possible ways that the structural parts of the queries can be matched with this XML document tree. Subsequently, each result from structure matching is stored at the Index Node that it matches. Figure 2 is a Structure Index resulting from document structure in Figure 1 (b), after queries from Example 3.1 are preprocessed. Each oval represents an Index Node where a text box shows queries that match at this Index Node. The algorithm to construct the Structure Index is described in Section 3.6.

of the Structure Index depends on the schema of the input XML documents. A definition of document structure results in the data guide [14] of input XML data. Therefore, the upper bound of the size of the Structure Index is the size of DataGuides, and does not depend on the size of a query workload. An empirical observation in [15] reveals that the size of DataGuides is usually small in real data regardless of its schema for data-oriented XML, because real data tends not to exploit all possible patterns allowed by its schema. On the other hand, for datasets where its data guide is large, the size of the Structure Index can be large. This affects the performance of the Structure Index significantly both in time (due to its Index Node construction), and space (due to the space usage of Index Nodes). This becomes the same characteristic as with lazy DFA.

Additionally, to validate the size of the Structure Index, we also show, in Figure 4, the number of Index Nodes in Structure Indexes of three different datasets, which are Protein [27], NASA [23] and NITF [8] datasets which is used in [15] and [9]. Table 1 shows some characteristics of the schemas of these three datasets.

Table 1. Characteristics of the schemas of three XML datasets.

Dataset	Number of element names	Number of attributes	Recursive schema
Protein	64	13	No
NASA	142	197	Yes
NITF	123	510	Yes

For each dataset, we extract the first 200 XML documents from the real dataset. In comparison, we also try to generate synthetic data to be similar to the real data. Since the Protein dataset has a maximum depth of 7, we generate its synthetic using $D=7$ and $RP=2$. NASA has a maximum depth of 8, so we generate its synthetic data using $D=8$ and $RP=2$. However, we do not have real data for NITF, thus, we generate it using the same parameter as in synthetic NASA dataset. Table 2 show some characteristics of three XML datasets. The results are shown in Figure 4.

Table 2. Characteristics of three XML datasets.

Dataset	Number of elements	Number of attributes	No. of elements and attributes
Protein – Real	19336	1254	20590
Protein – Synthetic	19521	8891	28412
NASA – Real	39574	4152	43762
NASA – Synthetic	25860	25703	51563
NITF – Synthetic	7652	29365	37017

From Figure 4, it can be seen that there is a huge difference in the number of Index Nodes between real and synthetic data. In this result, the number of Index Nodes of synthetic data continues to increase since its upper bound is large, and it cannot reach its upper bound. In contrast, the upper bound of real dataset is very small. After a short period, the number of Index Nodes almost reaches the upper bound, and becomes almost constant. In addition, it

can be seen that synthetic data of the Protein dataset has the same characteristic as real data because its schema is non-recursive. Thus, its synthetic data cannot be much different from its real data

Nevertheless, another major space usage in the Structure Index is the XPath expression table, which contains pointers to the XPath expressions that kept at each Index Node (for fast maintenance of the index in case of document updates). The size of this table grows linearly with the size of queries in a workload. However, this XPath expression table is used to improve the performance of Index Node construction, and can be removed to save more space.

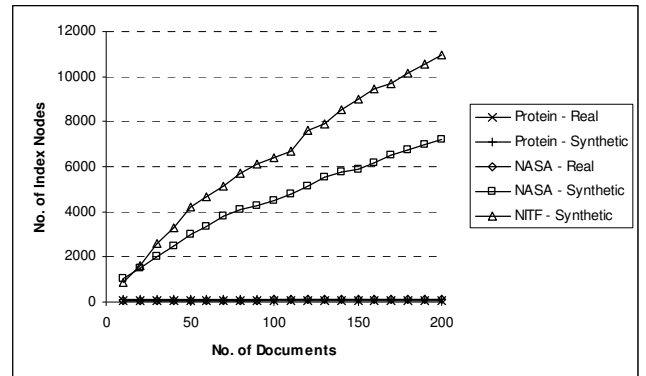


Figure 4. Number of Index Nodes of three different datasets for real data and synthetic data.

3.5 Warm-up and Stable Phase

The processing of Structure Index can be divided into two phases, which are *warm-up phase* and *stable phase*. The warm-up phase is a phase where most of Index Nodes of Structure Index are constructed. This Index Node construction accounts for most of the processing time in the warm-up phase. However, this processing overhead only occurs at the beginning of the process. After it reaches its stable phase, there is virtually no processing overhead because construction of Index Nodes does not occur or rarely occurs. At this stage, the processing time becomes approximately constant as analyzed previously. Nevertheless, under certain circumstance where data is synthetic and their schemas are recursive, it is possible that its stable phase will not be reached, and the processing overhead from constructing an index is unavoidable. However, this situation hardly occurs in practice for data-oriented XML. To verify this analysis, we show the number of Index Node construction, using the same datasets from previous section, in Figure 5. The reported number is an average number for groups of ten XML documents.

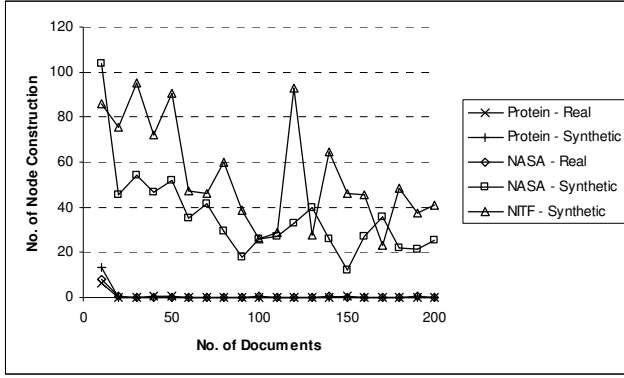


Figure 5. The number of Index Node construction from three different datasets for real and synthetic data.

From Figure 5, it can be seen that, for real data, major constructions of Index Nodes occur at the very beginning of the processing, and becomes zero, or a few after that. On the other hand, for synthetic data, constructions of Index Node are unavoidable because of its large upper bound. Again, synthetic data of protein dataset has the same characteristics as real data because of its non-recursive schema.

3.6 Maintaining the Structure Index

3.6.1 Structure Update

Structure update occurs when the Structure Index encounters unknown document structure and needs to construct Index Nodes for this new document structure. This process can be viewed as a learning process of the Structure Index. An algorithm to construct a new Index Node is given in Algorithm 3.1.

Algorithm 3.1 An algorithm to construct an Index Node

Method UPDATE-STRUCTURE (P, t)

Input: P is a parent Index Node where an update occur
 t is a name of found element or attribute.

Data: XPT is an XPath expression table of Index Node P

Output: N is a new Index Node which is a child of P

```

1 Create new Index Node  $N$ ;
2 Find XPath expression in  $XPT$  that has a name test  $t$ ;
3 for (each expression  $e$  with name test  $t$ )
4     INSERT-EXPR ( $N, e$ );
5 Find XPath expression in  $XPT$  that has a name test  $*$ ;
6 for (each expression  $e$  with name test  $*$ )
7     INSERT-EXPR ( $N, e$ );
8  $p =$  parent of  $P$ ;
9 while ( $p$  is not the root node)
10    Find XPath expression in  $XPT$  of  $p$  that has a
        name test  $t$  with a descendant axis;
11    for (each expression  $e$  with name test  $t$ )
12        INSERT-EXPR ( $N, e$ );
13    Find XPath expression in  $XPT$  of  $p$  that has a
        name test  $*$  with a descendant axis;
```

```

14    for (each expression  $e$  with name test  $*$ )
15        INSERT-EXPR ( $N, e$ );
16     $p =$  parent of  $p$ ;
17    set  $N$  as a children of  $P$ ;
18    return  $N$ ;
```

Method INSERT-EXPR (N, e)

Input: N is an Index Node

e is a name of found element or attribute.

Data: XPT is an XPath expression table of Index Node N

MQ is a list of XPath queries that match at Index Node N

VPI is a value-based predicate index for XPath queries that its structure navigation part match at Index Node N

```

1  $c =$  child of the main path of  $e$ ;
2 if ( $c$  is NULL)
3     Insert  $c$  in  $MQ$ ;
4 else if ( $c$  is a valued-based predicate)
5     Insert  $c$  in  $VPI$ ;
6 else
7     Insert  $c$  in  $XPT$ ;
8 for (each predicate  $p$  of  $e$ )
9     if ( $p$  is a valued-based predicate)
10        Insert  $p$  in  $VPI$ ;
11    else
12        Insert  $p$  in  $XPT$ ;
```

Intuitively, this algorithm simply evaluates queries with the Structure Index. In this algorithm, structure update occurs at an Index Node P with a new element or attribute t . In each Index Node, important information are kept, which are XPT , an XPath expression table of XPath fragments of queries that match at this node, MQ , a list of queries that match at this node, and VPI , a value-based predicate index which is used to evaluate value-based predicates of queries that match at this node. Then, a new Index Node is constructed, and all XPath fragments that have node test t and $*$ need to insert in this new Index Node. In addition, we also need to look for XPath fragments that have node test t and $*$ at each ancestor, because the descendant axis can match at any level. Alternatively, we can insert XPath fragments with descendant axis at all of its descendant nodes, but this requires more space, and the former method makes space usage of the Structure Index less sensitive to the descendant axis.

3.6.2 Query Update

Query update occurs when there is an update in the query workload. There are two types of query update, insertion and deletion.

Query insertion can be viewed as evaluating one query with one XML document, since the Structure Index is simply a structural part of XML documents. Generally,

query insertion is very efficient because the Structure Index is usually very small in data-oriented XML. Additional work is required to insert value-based predicates into a valued-based predicate index at each corresponding Index Node. This is also very efficient because valued-based predicate index is implemented as a binary search tree.

Query deletion can be done by traversing the whole Structure Index and removing the query from each Index Node. This is also efficient because the Structure Index is usually very small in data-oriented XML.

3.7 Nested Paths

In this section, we extend the Structure Index to handle more complex XPath queries, queries with nested paths. This kind of queries is more complex and quite different from the problem of linear XPath queries. One general approach that can be used to deal with this problem is to decompose an XPath query with nested paths as multiple linear XPath queries, and combine results from each linear XPath query to answer the query with nested paths. This approach is demonstrated in Example 3.2.

Example 3.2 Given the XPath query from Example 1.1:

$Q_0 = /a/*[c/@n>1][//d=3]//c$

It can be decomposed into the following linear XPath queries:

$Q_{3.0}$ (main) = $/a/*//c$

$Q_{3.1}$ (nested) = $/a/*[c/@n>1]$

$Q_{3.2}$ (nested) = $/a/*[//d=3]$

However, this approach becomes complicated since each nested path must match under the same element of a document. To avoid this problem, we use a slightly different approach by viewing an XPath query with nested paths as a tree of linear queries.

In addition, from an empirical observation with data-oriented XML, a major difference between queries in the query workload is not in their structure navigation part, but in their predicate evaluation part. This is because possible patterns of linear XPath queries without predicates is quite limited. For example, consider different the XPath queries in the form:

$//keyword[text()="database"]$

$//keyword[text()="XML"]$

$//keyword[text()="XPath"]$

...

Those queries have the same structure navigation part, but are different in their predicate evaluation part. This kind of query is likely to happen and the domain values which can be used in the predicate evaluation part is very large. As a result, in a large query workload, there will be a large number of value-based predicates to be evaluated, and only a small amount of those value-based predicate will evaluate to true. Therefore, we take advantage of this observation by using the concept of trigger, where each query has its *Trigger Tree*, and the processing of a trigger

occurs only when value-based predicates are evaluated to true. This Trigger Tree allows a processing of arbitrary nested paths and can be extended to support an XPath query with boolean connectors. This approach to handle XPath queries with nested paths are further described in the following subsections.

3.7.1 Query Tree

A query tree is a tree that represents an XPath query. A query tree of the query from Example 3.2 is shown in Figure 6. It should be noticed that both nested paths $c/@n>1$ and $//d=3$ are represented as $c/@n[text()>1]$ and $//d[text()=3]$ respectively since those forms can also represent main path with value-based predicate, and it evaluate to the same result.

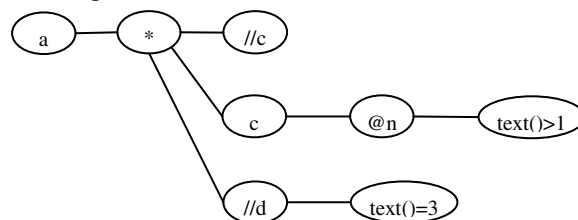


Figure 6. A query tree represents a query from Example 3.2.

3.7.2 Trigger Tree

A Trigger Tree is a tree of trigger node that represents a structure of an XPath query. A query tree of a query from Example 3.2 is shown in Figure 7.

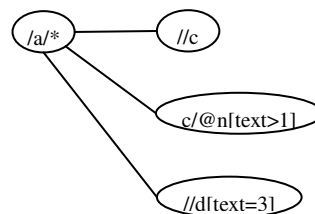


Figure 7. A trigger tree of a query from Example 3.2.

Later on, a Trigger Tree is preprocessed with the Structure Index. This is illustrated in Figure 8 where the Structure Index from document structure in Figure 1 (b) is preprocessed with a Trigger Tree in Figure 7.

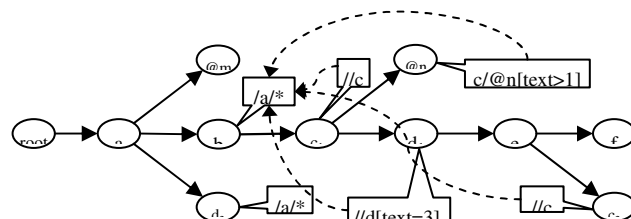


Figure 8. A Structure Index from document structure in Figure 1(b) after a Trigger Tree in Figure 7 is preprocessed.

3.7.3 Trigger Tree at Run time

At run time, input XML documents are processed similarly to the processing described in Section 3.3. Additional work is that when a nested predicate matches, it needs to trigger its parent; and if all children of the parent match, the parent becomes true, and the following condition is checked. If this trigger node (parent) also has a parent, again, it needs to trigger its parent. If it does not have a parent, it means that this node is a root node of a Trigger Tree and a query corresponding to this node match with this document. Nevertheless, extra work to clean up a Trigger Tree, and some duplicate triggers has to be handled carefully.

4 EXPERIMENTS

This section presents various experiments of the Structure Index. Our execution environment consists of a Pentium III 600 Mhz processor with 512MB memory running JVM 1.4.1 on Linux kernel 2.4.22. The SAX parser we used is Xerces2 Java Parser 2.5.0 [2] in non-validating mode.

The NASA XML dataset [23] is chosen as an experiment dataset because it is a data-oriented XML with recursive schema. We used the first 200 XML documents concatenated into a single file. We use a modified version of the XPath generator in [9] to generate synthetic XPath queries in distinct mode. The modification is to generate value-based predicates using data values from the NASA XML dataset.

All the numbers reported are averages of this dataset unless stated otherwise. The probability of wildcard (*) and descendant axis (//) is set to 20%. All reported processing times are in milliseconds, including document parsing time.

4.1 Experiment 1: Linear XPath Queries

The first experiment shows the performance of the Structure Index when the query workload consist of linear XPath queries. We varied the size of a query workload from 1,000 to 1,000,000. The processing time reported in the graph is the processing time for each XML document. The reported time is an average time for each ten XML documents. The results are shown in Figure 10.

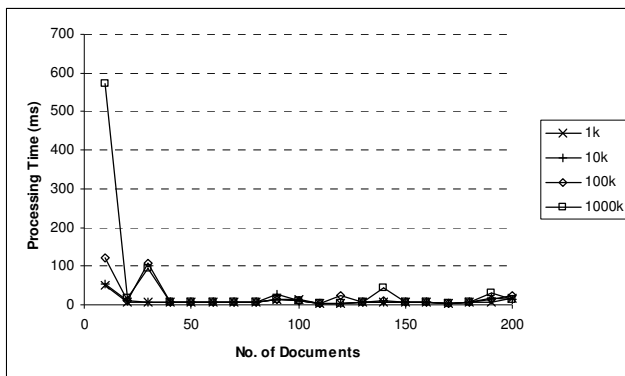


Figure 9. The processing time of real NASA data for 1k, 10k, 100k, 1000k queries.

This experimental result has shown that, after the short warm-up phase, our approach is extremely efficient. It uses constant processing time independent of the query workload. A little fluctuation comes from a few Index Node constructions that rarely occur after the short warm-up phase. Also, it should be noted that the processing time of the warm-up phase depends on the size of queries. This result comes from the construction cost of Index Nodes, which is more expensive when the query workload is large.

In comparison, we also use synthetic NASA data from Section 3.4. To ease the comparison, we plotted one result from real NASA dataset at the query size of 100,000. We ran the experiment with the same above setting. The results are shown in Figure 11.

From this result, there is a huge difference between the processing time of synthetic data and real data as can be compare in the case of 100,000 queries. Synthetic data takes much more processing time, and depends on the query workload, because it cannot achieve its stable phase.

In brief, for real data-oriented XML, the processing time of the Structure Index is about constant independent of the query workload, while, for synthetic data, it cannot achieve its stable phase and results in decreased performance.

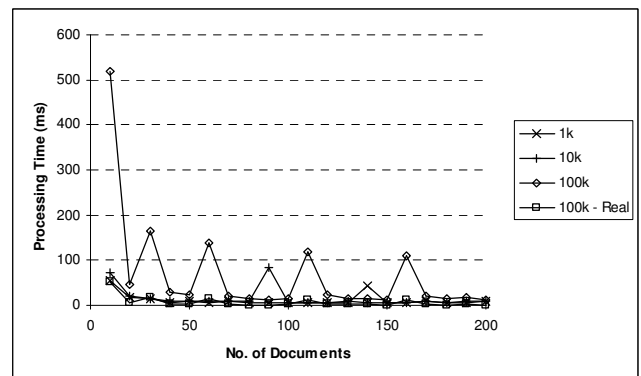


Figure 10. The processing time of synthetic NASA data for 1k, 10k, 100k queries.

4.2 Experiment 2: XPath Queries with Nested Paths

In this experiment, we evaluate the performance of the Structure Index with Trigger Trees when the query workload consists of XPath queries with nested paths. We varied the size of a query workload from 10,000 to 50,000. The probability of equal, range, and not equal operators are set to 80%, 10% and 10% respectively. All reported time is the processing time in stable phase. In the first experiment, we show that the processing time of Trigger Tree depends largely on selectivity of value-based

predicates. All queries have 3 nested paths. We vary percentage of nested paths without value-based from 0, 25, 50, 75, and 100 percent. The result is shown in Figure 12.

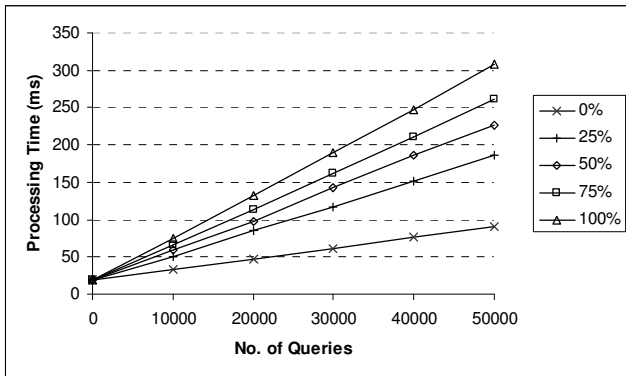


Figure 11. A comparison of the processing time for queries with nested paths with/without value-based predicates.

It can be seen from this figure that the scalability depends on the percentage of nested paths without value-based predicates, where pure nested paths with value-based predicates (0%) has the best scalability, and pure nested paths without value-based predicates (100%) has the worst scalability. Also, the scalability of pure nested paths with value-based predicates is quite distinct from the others. Obviously, this comes from the selectivity of value-based predicates because a nested path with value-based predicate has a much higher selectivity. To gain more understanding on this result, the number of triggers is shown in Figure 13.

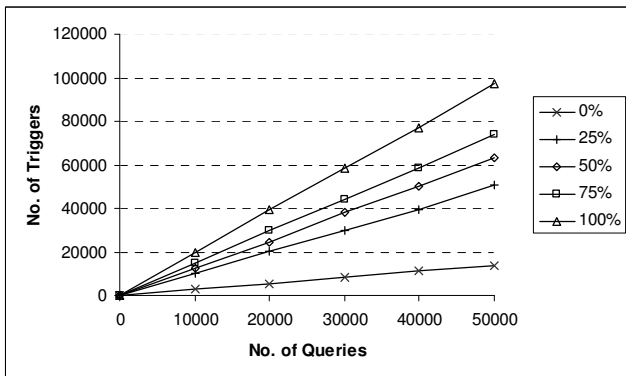


Figure 12. A comparison of the number of triggers for queries with nested paths with/without value-based predicate.

This figure is very similar to the previous figure. This result verifies that the processing time of Trigger Tree depends primarily on the number of triggers. In addition, when all nested paths have value-based predicates (0%), Trigger Tree is very efficient. This result leads to the concept of post-processing where less selective value-based predicates are processed after more selective value-based predicates has been satisfied.

4.3 Experiment 3: XPath Queries with Nested Paths using Post-processing Concept

This experiment demonstrates the performance of the Structure Index with Trigger Tree using post-processing concept, when the query workload consists of XPath queries with nested paths. The first result in Figure 14 shows the processing time of a Trigger Tree using post-processing concept.

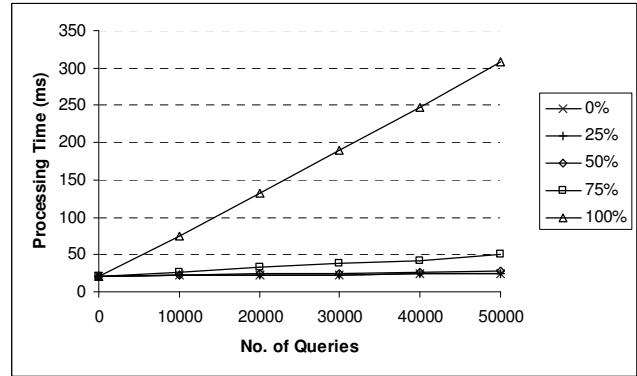


Figure 13. A comparison of the processing time for queries with nested paths with/without value-based predicates using post-processing concept.

From this figure, it can be seen that post-processing significantly improves the performance because it post process value-based predicates with low selectivity. Nevertheless, for the case of pure nested paths without value-based predicates (0%), the result is the same as in the previous experiment. In this case, post-processing cannot help because all its paths have low selectivity. To gain more understanding, we also show the number of triggers in Figure 15.

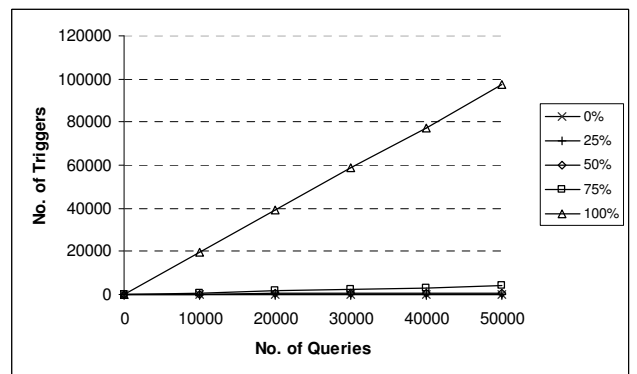


Figure 14. A comparison of the number of triggers for queries with nested paths with/without value-based predicates using post-processing concept.

As predicted, post-processing greatly reduces the number of triggers. In fact, the number of triggers is even less than the number of triggers of pure nested paths with value-based predicates from the previous experiment. This comes from that post-processing also postpone range and not equal operators since it has lower selectivity than

an equal operator. Again, the number of trigger of pure nested paths without value-based predicates far exceeds the others.

Next, we show the performance of post-processing concept while varying the number of nested paths from 2, 5, and 8 nested paths. In this experiment, we vary the number of queries from 20000 to 100000. We set the percentage of nested paths without value-based predicate to 67 percent in all queries since nested path tends to end with value-based predicates. Other parameters are the same as in the experiment 2. The result is shown in Figure 16.

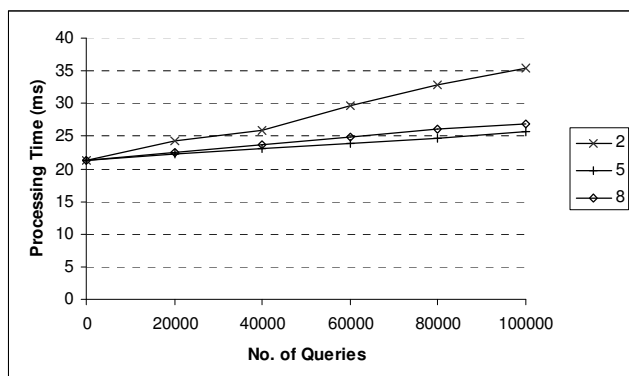


Figure 15. The processing time for queries with nested paths varying number of nested paths.

From this result, the performance of Trigger Tree with post-processing is highly scalable to a large number of nested paths. In fact, queries with 2 nested paths use more processing time than the others. Again, this comes from the selectivity of the value-based predicates since more nested paths means more chance for a query to have more selective value-based predicates. However, queries with 8 nested paths also use more processing time than queries with 5 nested paths. This comes from the number of decomposed linear queries, where 5 nested paths decompose to 600,000 linear queries, and 8 nested paths decompose to 900,000 linear queries. Thus, there is more chance for value-based predicates of queries with 8 nested paths to be evaluated to true, while queries with 5 nested paths is the most balance between its selectivity, and the number of decomposed linear queries.

5 CONCLUSIONS

In this paper, we proposed a novel approach called *Structure Index* for evaluating a large number of XPath queries on XML streaming data. Our focus is on data-oriented XML which has many practical applications. Our approach is based on the document structure, which can be easily derived from the input XML documents. After that, XPath queries are preprocessed and annotated into the Structure Index to facilitate efficient matching against future input XML documents. We analyzed the efficiency of the Structure Index both in time and space, and also by experiments. After a short warm-up phase, it achieves constant processing time, $O(1)$, per

SAX event independent of the query workload for linear XPath queries. In addition, we proposed a method called *Trigger Tree* to efficiently evaluate XPath queries with nested paths. This method is scalable in both the number of nested paths and the number of value-based predicates when the queries have at least one value-based predicate. For space usage, the space efficiency of the Structure Index depends on the size of the structure of the input XML, which is usually very small for data-oriented XML. The maintenance cost (query update) of our approach is also very efficient because it also depends on the size of (small) document structure.

6 REFERENCES

- [1] M. Altinel, and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of VLDB*, pages 53-64, Cairo, Egypt, September 2000.
- [2] The Apache Software Foundation. Xerces2 Java Parser. <http://xml.apache.org/xerces2-j>.
- [3] S. Babu, and J. Widom. Continuous Queries Over Data Streams. *SIGMOD Record*, 30(3):109-120, September 2001.
- [4] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of the International Conference on Data Engineering*, 2002.
- [5] J. Chen, D. DeWitt, F. Tian, Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 379-390, Dallas, TX, May 2000.
- [6] Y. Chen, S. B. Davidson, Y. Zheng. Validating Constraints in XML. Technical report, *University of Pennsylvania*, 2002. Technical Report MS-CIS-02-03.
- [7] B. F. Cooper, N. Sample, M. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *VLDB*, pages 341-350, September 2001.
- [8] R. Cover, The SGML/XML Web Page. <http://www.w3.org/TR/xslt>, 1999.
- [9] Y. Diao, M. Altinel, M. Franklin, and P. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. In *Proceedings of ACM Transactions on Database Systems*, December 2003.
- [10] Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of xml documents. In *proceeding of the International Conference on Data Engineering*, 2002.
- [11] F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, D. Shasha. Filtering Algorithms and Implementations for Very Fast Publish/Subscribe Systems. In *Proceedings of ACM SIGMOD*, pages 115-126, Santa Barbara, California, May 2001.
- [12] F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, D. Shasha. Webfilter: A High-throughput XML-based Publish and Subscribe System. In *Proceedings of The VLDB Journal*, pages 723-724, 2001.
- [13] L. Fegaras, D. Levine, S. Bose, V. Chaluvadi. Query Processing of Streamed XML Data. Submitted, November 2001.

- [14] R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of Very Large Data Bases*, pages 436-445, September 1997.
- [15] T. Green, G. Miklau, M. Onizuka, D. Suciu. Processing XML Streams with Deterministic Automata. Submitted to *The 9th International Conference on Database Theory*, Siena, Italy, 8-10 January 2003.
- [16] A. Gupta, A. Y. Halevy, D. Suciu. View Selection for Stream Processing. Submitted to *Fifth International Workshop on the Web and Databases (WebDB 2002)*, Madison, Wisconsin, June 2002.
- [17] Ashish Gupta, Dan Suciu. Stream Processing of XPath Queries with Predicates. In *Proceeding of ACM SIGMOD Conference on Management of Data*, 2003.
- [18] Z. Ives, A. Halevy, and D. Weld. An XML query engine for network-bound data. In *Proceedings of Very Large Data Bases*, December 2002.
- [19] Z. Ives, A. Halevy, and D. Weld. Efficient Evaluation of Regular Path Expressions on Streaming XML Data. Technical report, *University of Washington*, 2000. Technical Report UW-CSE-200-05-02.
- [20] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the 2002 Intl. Conf. on Data Engineering*, Feb 2002.
- [21] G. Miklau, and D. Suciu. Containment and equivalence of an XPath fragment. In *Proceedings of the ACM SIGMOD/SIGART*, pages 65-76, June 2002.
- [22] T. Milo, and D. Suciu. Index Structures for Path Expressions. In *ICDT '99, 7th International Conference*, Jerusalem, Israel, January 10-12, 1999, pages 277-295, 1999.
- [23] NASA's astronomical data center. ADC XML resource page. <http://xml.gsfc.nasa.gov/>.
- [24] B. Nguyen, S. Abiteboul, and G. Cobena. Monitoring XML Data on the Web. In *Proceedings of the 2001 ACM SIGMOD International Conference On Management of Data*, pages 437-448, May 2001.
- [25] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proceedings of Workshop on XML Data Management (XMLDM)*, LNCS. Springer, 2002.
- [26] M. Onizuka. Light-weight XPath Processing of XML Stream with Deterministic Automata. In *Proceedings of CIKM*, 2003.
- [27] Protein Information Resources. PIR International Protein Sequence Database. <http://pir.georgetown.edu>.
- [28] A. Snoeren, K. Conley, and D. Gifford. Mesh-based Content Routing Using XML. In *Proceedings of the 18th Symposium on Operating Systems Principles*, 2001.
- [29] E. Viglas, and J. Naughton. Rate-based Query Optimization for Streaming Information Sources. In *Proceedings of SIGMOD*, 2002.
- [30] W3C (1999) XML path language (XPath) 1.0. <http://www.w3.org/TR/xpath>
- [31] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proceedings of SIGMOD*, 2003.