

# Light-weight Acceleration for Streaming XML Document Filtering

Shuichi Mitarai<sup>1</sup>

Akira Ishino<sup>2</sup>

Masayuki Takeda<sup>1,3</sup>

<sup>1</sup> Department of Informatics, Kyushu University, Fukuoka 819-0395, Japan

<sup>2</sup> Department of System Information Sciences, Tohoku University, Miyagi 980-8579, Japan

<sup>3</sup> SORST, Japan Science and Technology Agency (JST)

{mitarai, takeda}@i.kyushu-u.ac.jp

ishino@ecei.tohoku.ac.jp

## Abstract

We present a light-weight streaming XML document filtering tool named XAXEN, which is scalable with respect to the number of queries. XAXEN consists of (1) an XML file transformer on data-sending server, which transforms an XML file into a trie representing the tag-name sequences of the root-to-leaf paths in XML tree and the binary XML file where every start- and end-tag is replaced with a special symbol followed by the corresponding path trie node ID and with another symbol, and (2) a query processor on receivers. Computational experiments show that XAXEN is 2 ~ 6 times faster and 6 times space-efficient in comparison with XMLTK, a well-known scalable streaming XML document filter.

## 1 Introduction

XML has become an increasingly popular format for information exchange, and efficient processing of broadcast XML data on constrained devices such as PDAs and mobile phones is of great importance. Many researches have been undertaken on streaming XML query evaluation (see, e.g., [7, 4, 11]). YFilter [7] and XMLTK [4] are well-known streaming XML query processors, which efficiently process XPath queries. YFilter builds an NFA for the queries by merging common prefixes of the query paths, and it is fast and scalable with respect to the number of queries. XMLTK adopted the so-called *Lazy-DFA* technique: it converts the NFA for the queries into a DFA lazily. It is highly scalable and copes with huge number of queries, but the amount of memory usage could be very large due to the conversion.

In this paper, we present a light-weight streaming XML document filtering tool, named XAXEN (eXtreamly-Accelerated XML filtering ENgine). XAXEN consists of a transformer on a data-sending server and a query processor on data receivers. That is, the transformer on data-sending

server preprocesses the XML files before sending them in order to bring scalabilities and time- and space-efficiency to data receivers. The idea of transforming XML files into some convenient format before sending them can be found in [6], for instance. There are also several proposals on XML binary formats [2]. Moreover, the Stream Index (SIX) technique [4], which is used for speeding up stream processing by XMLTK, is a good example of data structures for acceleration obtained by preprocessing XML data.

The transform adopted in XAXEN has the following two properties:

1. The time for transform is reasonably small.
2. The transformed data is not larger than the original data.

XAXEN transforms the input XML file into a *path trie* and a *binary XML file* during one pass. The former is a trie representing all the label strings of the paths from the root, which is called the *path trie*, and the latter is obtained from the input XML file by replacing every occurrence of the start tags with a special code followed by a pointer indicating the corresponding path trie node, and every occurrence of the end tags with another special code. We note that in practice the path trie is sufficiently small and fits main memory for moderate XML files. Table 1 shows characteristics of the original XML files and the binary ones, the corresponding XML trees, and the path tries for two datasets: one is DBLP [9] and the other is a randomly generated data with xmlgen [12] from a DTD provided by XMark benchmark.

Data-sending server sends a path trie followed by a binary XML file, not the original XML file. Data receivers first perform path-pattern matching against the path trie for given queries, and add some information about the results to the path trie. By the information, the receivers do not need to do path-pattern matching during scan of the binary XML file: they obtain the results of path-pattern matching with the help of path-trie nodes embedded in the binary

**Table 1. Characteristics of two datasets.**

	DBLP	random
original file size (MB)	352	111
# XML tree nodes	8,632,812	1,666,310
# tag names	35	74
binary XML file size (MB)	208	84
# path trie nodes	138	515
transform time (sec)	100.41	73.26

XML file. The path trie is often sufficiently smaller than the original XML file, and therefore the processing time is drastically reduced. Moreover the SIX technique accelerates XAXEN at rate of  $1.6 \sim 7.2$ .

The path trie is a kind of DataGuide [8]. Although using DataGuide for efficient XML processing is not new, our usage is different from existing ones to our best knowledge.

The organization of this paper is as follows. Section 2 defines our problem which is the basis of more complex problems needed to XML document filtering. Section 3 gives an overview of XAXEN’s query evaluation strategy. Section 4 then estimates the performance of XAXEN in comparison with YFilter and XMLTK. Section 5 discusses how to extend XAXEN to cope with more complex queries: logical connectives, nested predicates, aggregations such as *count*, and so on. Section 6 concludes this work.

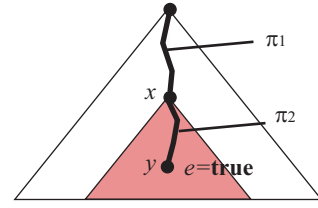
## 2 Notation and problem formulation

Let  $\Sigma$  be a finite set of characters, and  $\Sigma^*$  (resp.  $\Sigma^+$ ) denote the set of strings (resp. nonempty strings) over  $\Sigma$ . Let  $\mathcal{N}$  be a set of tag names. An *XML tree* is an ordered tree such that the interior nodes are labeled by tag names in  $\mathcal{N}$  and the leaves (called the *text nodes*) are labeled by strings in  $\Sigma^*$ .<sup>1</sup>

A *simple path pattern* is a sequence consisting of tag names in  $\mathcal{N}$  and  $*$ ’s, separated by “/” or “//”, where “/” and “//” denote the parent-child and the ancestor-descendant axes, respectively. A simple path pattern  $\pi$  is said to *match* a path in an XML tree if  $\pi$  matches the sequence of tag names spelled out by the path when regarding “\*” and “//” as a wildcard (that matches any tag name) and a variable-length-don’t-care (that matches any string of tag names), respectively.

A *path pattern* is an ordered pair of simple path patterns  $\pi_1$  and  $\pi_2$ , written as  $\pi_1[\pi_2]$ . For any node  $x$  and its descendant  $y$  (possibly  $x = y$ ) in an XML tree, a path pattern  $\pi_1[\pi_2]$  is said to *occur at locus*  $(x, y)$  if  $\pi_1$  and  $\pi_2$ , respectively, match the path from the root to  $x$  and the path from

<sup>1</sup>In this paper an attribute node corresponding to “name=value” is regarded as an interior node labeled “@name” having a unique child (leaf) labeled “value”.



**Figure 1. An illustration of occurrence of XPattern  $\pi_1[\pi_2 : e]$  at locus  $(x, y)$  of XML tree.**

$x'$  to  $y$ , where  $x'$  is the node preceded by  $x$  on the path from the root to  $y$ .

Let  $e = e(w_1, \dots, w_m)$  be a Boolean expression over keywords  $w_1, \dots, w_m$  in  $\Sigma^+$ . A text  $d$  in  $\Sigma^*$  is said to *satisfy*  $e$  if  $e$  is true under the truth-value assignment determined by whether or not the corresponding keywords  $w_1, \dots, w_m$  occur in the text  $d$ .

An *XPattern* is a pair of a path pattern  $\pi_1[\pi_2]$  and a Boolean expression  $e$  over keywords  $w_1, \dots, w_m$  in  $\Sigma^+$ , written as  $\pi_1[\pi_2 : e]$ . An XPattern  $\pi_1[\pi_2 : e]$  is said to *occur at node*  $x$  of an XML tree if  $x$  has a descendant  $y$  such that the path pattern  $\pi_1[\pi_2]$  occur at locus  $(x, y)$ , and the Boolean expression  $e$  is satisfied by at least one text node that is a child of  $y$ . We note that a path pattern  $\pi_1[\pi_2]$  can be viewed as the XPattern  $\pi_1[\pi_2 : \text{true}]$ .

Fig. 1 gives an illustration of occurrence of XPattern  $\pi_1[\pi_2 : e]$  at locus  $(x, y)$  of XML tree.

Now, we give a formal definition of a basic problem we address in this paper.

### Definition 1 (Basic problem)

**Given.** An XML data  $T$ .

**Query.** A set of XPatterns  $P_1, \dots, P_\ell$ .

**Answer.** A node  $x$  of the XML tree for  $T$  at which  $P_i$  occurs for every  $i = 1, \dots, \ell$ .

One may think that the class of XPatterns is too simple compared with XPath expressions. The problem is, however, the basis of problems for more complicated queries as stated in Section 5.

The input XML data  $T$  is assumed to be a sequence of XML documents, i.e., a sequence of XML trees. One may consider that there is no need for computing the nodes  $x$  since we have only to get the sets of documents satisfying the queries for XML document filtering. However, computing the nodes  $x$  is required when processing complex queries written as combinations of multiple XPatterns.

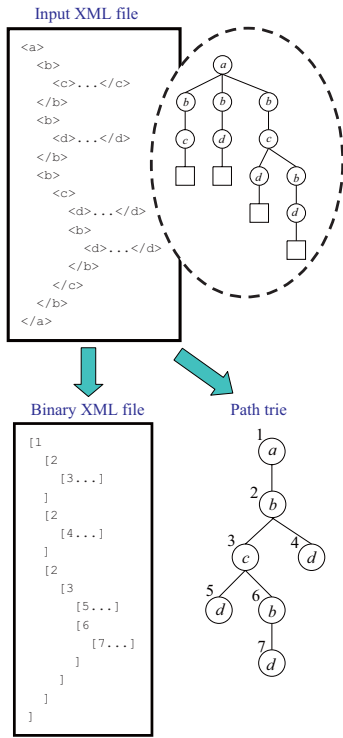


Figure 2. Preprocessing of XML data.

### 3 XAXEN: Fast filtering using path trie

#### 3.1 Path trie and binary XML file

Our method transforms the input XML file into a path trie and a binary XML file, as shown in Fig. 2. An XML file with its tree representation is displayed on the upper in Fig. 2, where the squares in the nodes represent the text nodes. The binary XML file and the path trie created from the XML file are shown on the lower-left and on the lower-right, respectively. The numbers adjacent to the nodes of the path trie imply their IDs. We note that the start tags and the end tags are, respectively, replaced with '[' followed by a node ID, and with ']' in the binary XML file. This transform process takes time  $O(n \log |\mathcal{N}| + |T|)$ , where  $n$  denotes the number of total number of occurrences of start tags,  $|\mathcal{N}|$  is the number of distinct tag names, and  $|T|$  is the input XML document size. In practice, it is not so time consuming as demonstrated in Table 1. We note that path tries are sufficiently smaller than XML document trees, and that the binary XML files are smaller than the original XML files.

It should be mentioned that the start tag occurrences are replaced with the corresponding path-trie node IDs, not with the tag IDs. Whenever reading a path-trie node ID in binary XML file, XAXEN checks the outputs of the path-

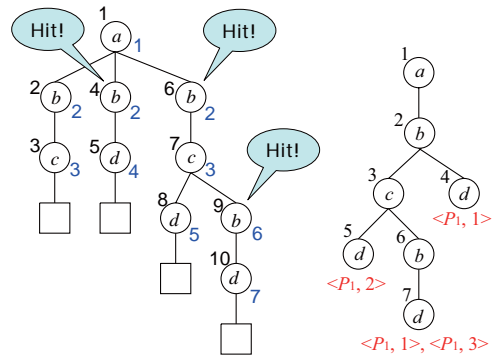


Figure 3. Illustration of path-pattern matching using the path trie of Fig. 2.

trie node to determine whether or not the corresponding node has a “match”.

#### 3.2 Path-pattern matching

Our method replaces the path-pattern matching for XML trees with the one for path tries. The idea is to add to a path trie node  $y$  an output  $(P_i, d)$  if and only if the path pattern  $P_i$  occurs at locus  $(x, y)$  of the path trie where  $x$  is the  $d$ -th ancestor of  $y$ . This enables us to obtain the loci of XML tree at which path patterns  $P_i$  occur.

Fig. 3 shows the occurrences of the path pattern  $P_1 = a//b[/d]$  in the path trie and the XML tree for the XML data used in Fig. 2. Since the path pattern  $P_1 = a//b[/d]$  occurs at loci  $(x, y) = (2, 5), (2, 7), (6, 7), (2, 4)$  in the path trie, the path trie nodes  $y = 5, 7,$  and  $4$  are associated with the outputs  $\{\langle P_1, 2 \rangle\}, \{\langle P_1, 1 \rangle, \langle P_1, 3 \rangle\},$  and  $\{\langle P_1, 1 \rangle\},$  respectively. On the other hand,  $P_1$  occurs at loci  $(x, y) = (4, 5), (6, 8), (6, 10), (9, 10)$  in the XML tree shown on the left in Fig. 3. These loci can be obtained from the outputs added to the path trie. For instance, the output  $\langle P_1, 2 \rangle$  of the path trie node 5 implies that the occurrence of  $P_1$  at locus  $(x, y)$  such that  $y = 8$  in the XML tree is associated with the path trie node 5 and  $x = 6$  in the XML tree is the 2nd ancestor of  $y$ .

Next, we briefly mention the path-pattern matching against a path trie. We compute the loci  $(x, y)$  of every path pattern in given XPatterns. We build NFAs from the path patterns and make them run along the paths of the path trie in a depth-first manner. (See [13] for detail). We adopt the bit-parallel technique [10] to implement the NFAs. Since the size of NFA for pattern  $\pi$  is the number of tag names and \*’s in  $\pi$  plus one and often fits the computer word size (32 or 64) in practice, we can perform the nondeterministic state transitions in parallel at high speed. Construction of NFAs can also be done very fast.

### 3.3 Keyword searching

XPath expressions such as `/a/b[@month=December]` and `/a/b[contains(name, "mickey")]` require attribute-value tests and keyword matching. Thus we need search in the contents of text nodes. We apply Aho-Corasick’s multikeyword search method to this problem. One advantage is space efficiency. Another important advantage is document-by-document processing based on simultaneous search of multiple keywords  $w_1, \dots, w_m$ . Thanks to the advantage, no set operations are needed: Boolean operations for every document are enough. In XML document filtering application, this advantage is of great importance as the set of keywords from a great deal of queries can grow very large. (This should be compared to the index-based methods based on text index techniques, such as suffix trees and suffix arrays. It proceeds pattern-by-pattern, and therefore a considerable amount of extra working space is needed for set operations.)

### 3.4 Query evaluation

We describe an algorithm of searching binary XML file for keywords and combining the keyword occurrences with the path pattern matching results associated to the path trie nodes in order to perform the query processing.

Let  $W = \{w_1, \dots, w_m\}$  be the set of keywords occurring in the Boolean expressions  $e$  of the XPatterns  $P_1, \dots, P_\ell$ . We build from  $W$  Aho-Corasick’s pattern matching machine  $M$ . We maintain variables  $offset$  and  $depth$  respectively representing the offset from the beginning of the binary XML file and the depth of the current node in XML tree during the scan of the binary XML file. We use a two-dimensional array  $Occ$  of Boolean values such that  $Occ[d][i]$  is **true** if  $w_i$  occurs in some text node that is a child of the current node of depth  $d$ . We use another two-dimensional array  $Q$  of Boolean values such that  $Q[d][q]$  is **true** if the XPattern  $P_q$  occurs at a node of depth  $d$  that is an ancestor of the current node. We repeat the following until reading the end of binary XML file: Read a one-byte character  $c$  from the binary XML file, and execute the following statements.

- If  $c$  is a special code implying a start tag, we read an integer  $v$ , and push  $(v, offset)$  into a stack  $S$ . Increment  $depth$  by one. Initialize the values of  $Occ[depth][1..m]$  and the values of  $Q[depth][1..\ell]$  by **false**. Set the current state of  $M$  to the initial state.
- If  $c$  is a special code implying an end tag, pop  $(v, start)$  from the stack  $S$ . If the path trie node  $v$  has outputs, then for each output  $\langle q, d \rangle$  evaluate the Boolean expression  $e$  of the XPattern  $P_q$  under the truth-value assignment determined by the values  $Occ[depth][1..m]$ . If  $e$  is **true**, then set  $Q[depth -$

$d][q]$  to **true**. For every  $q$  in  $\{1, \dots, \ell\}$  such that  $Q[depth][q]$  is **true**, outputs the node  $(start, offset)$ . Decrement  $depth$  by one. Set the current state of  $M$  to the initial state.

- Otherwise, make a state-transition of  $M$  on  $c$ . If the current state of  $M$  has outputs, then update the values  $Occ[depth][1..m]$  depending upon the keyword occurrences implied by the outputs.

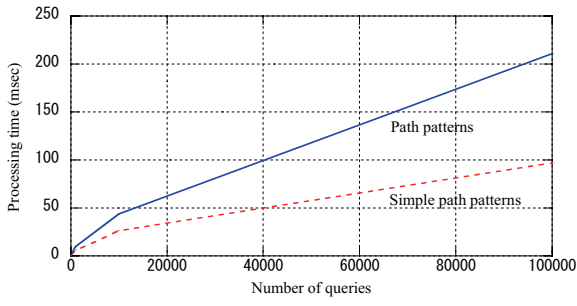
## 4 Computational experiments

We estimate the performance of XAXEN experimentally. The times for path pattern matching against the path trie and the times for scanning the binary XML file were measured. The performance is compared to a naive method without path trie. Then we compare the performance of XAXEN to those of YFilter and XMLTK. In addition, the speed-ups of XAXEN by SIX were compared to those of XMLTK by SIX.

All the experiments were carried out on an Personal Computer with an Intel Pentium 4 (CPU 2.4GHz and 2.0GB RAM) running RedHat Linux Advanced Server 2.1 operating system. The input XML file was randomly generated with `xmlgen` [12] from a DTD of XMark benchmark and is of size 111MB. The (simple) path patterns were randomly generated by using `pathgenerator` [1] where the occurring probabilities of `//` and `*` were set to (1%, 1%) and to (10%, 10%). We use the simple path patterns with  $(//, *) = (1\%, 1\%)$ , unless we do not mention about the parameters of the query sets.

**Times for path pattern matching.** Fig. 4 shows path pattern matching times (incl. NFA construction times) against varying number of queries. They are relatively small compared with the times for scanning binary XML file (shown in Table 4) even for a large number of queries. This is mainly due to the fact that the path trie size is sufficiently smaller than the XML tree size (see Table 1) and to practically fast implementation based on the bit-parallel technique.

**Times for scanning binary XML file.** Table 2 compares the performances of XAXEN and Naive in scanning binary XML file against varying number of simple path patterns, where “Naive” does path-pattern matching with NFAs during the scan. Naive builds NFAs for the queries, and performs state-transitions of the NFAs for each of the tag IDs embedded in a binary XML file, while XAXEN only refers to the outputs added to the path trie for each of the path-trie node IDs embedded in a binary XML file. Although the running time of Naive grows linearly proportional to the number of queries, that of XAXEN does not so: it depends



**Figure 4. Times for path pattern matching against path trie are displayed.**

upon the number of references to the outputs added to the path-trie nodes.

**Comparison with XMLTK and YFilter.** Table 3 displays the amounts of memory usage against 10,000 and 100,000 queries. We used simple path patterns as the queries since XMLTK does not allow queries with predicates completely. For the 10,000 queries with  $(//, *) = (10\%, 10\%)$ , the amount of memory usage of XAXEN is not greater than 1/6 of those of XMLTK and YFilter.

Table 4 compares the execution times of XAXEN, XMLTK and YFilter against varying number of simple path patterns with  $(//, *) = (1\%, 1\%)$ . The results show that XAXEN is 2 ~ 6 times faster than YFilter and XMLTK.

**Estimation of speedups with SIX.** Table 5 displays the execution times of XAXEN and XMLTK, both accelerated with SIX. Comparison with the results of Table 4 shows that XAXEN with SIX is 1.6 ~ 7.2 times faster than XAXEN without SIX and the speedup rate of XAXEN is higher than that of XMLTK.

**Table 2. Query execution time.**

	# queries	elapsed time (sec)	
		XAXEN	Naive
simple path pattern 1%	1	0.51	0.86
	10	0.52	1.29
	100	0.54	4.30
	1,000	0.53	21.81
path pattern 1%	1	0.51	0.86
	10	0.52	1.36
	100	0.52	5.51
	1,000	0.54	34.50

## 5 Extensions

### 5.1 Permitting references to other queries

Let the input queries in the XPatterns format be:

$$P_1 = \pi_1^1[\pi_2^1 : e^1], \dots, P_\ell = \pi_1^\ell[\pi_2^\ell : e^\ell],$$

where the expressions  $e^i$  are Boolean expressions over string patterns  $w_1, \dots, w_m$ . Now, we extend  $e^i$  to be Boolean expressions over variables  $w_1, \dots, w_m$  and  $P_1, \dots, P_{i-1}$ . The truth-value evaluation of the  $e^i$  parts can be done under the truth-value assignment determined by the values  $Occ[depth][1..m]$  and  $Q[depth][1..i-1]$  for the current value of  $depth$  in the algorithm stated in Section 3.4. This extension enables us to introduce the logical connectives and the nested predicates as follows.

**Table 3. Memory usage comparison. Input queries are simple path patterns, where the occurring probabilities of // and \* are (1) 1% and (2) 10%.**

	# queries	memory usage (KB)		
		XAXEN	YFilter	XMLTK
(1)	10,000	3,320	1,169,975	30,288
	100,000	18,632	(memory full)	285,328
(2)	10,000	4,952	1,494,845	34,412
	100,000	28,543	(memory full)	318,560

**Table 4. Execution time comparison.**

# queries	elapsed time (sec)		
	XAXEN	YFilter	XMLTK
1	0.57	39.24	2.27
10	0.57	42.54	2.57
100	0.57	45.22	3.09
1,000	0.67	61.30	4.14
10,000	1.85	155.30	11.83
100,000	142.04	(memory full)	270.81

**Table 5. Comparison of speedups with SIX.**

# queries	elapsed time (sec)	
	XAXEN	XMLTK
1	0.00	0.14
10	0.07	1.53
100	0.18	2.49
1,000	0.39	4.37
10,000	1.81	12.25
100,000	139.62	275.74

**Logical connectives.** Consider the XPath query  $\pi[\pi_1 \text{ and } \pi_2]$ , for instance. Let  $P_1 = \pi[\pi_1 : \mathbf{true}]$  and  $P_2 = \pi[\pi_2 : \mathbf{true}]$ , and let  $P_3 = \pi[\varepsilon : P_1 \wedge P_2]$ . Then the XPath query  $\pi[\pi_1 \text{ and } \pi_2]$  occurs if and only if  $Q[\text{depth}][3] = \mathbf{true}$ .

**Nested predicates.** Consider the XPath query  $\pi_1[\pi_2[\pi_3 \text{ and } \pi_4]]$ . Let  $P_1 = \pi_1\pi_2[\pi_3 : \mathbf{true}]$ ,  $P_2 = \pi_1\pi_2[\pi_4 : \mathbf{true}]$ , and  $P_3 = \pi_1[\pi_2 : (P_1 \wedge P_2)]$ . Then the XPath query  $\pi_1[\pi_2[\pi_3 \text{ and } \pi_4]]$  occurs if and only if the value  $Q[\text{depth}][3]$  is **true**.

We remark that, letting  $P_4 = \pi_1[\pi_2\pi_3 : \mathbf{true}]$ ,  $P_5 = \pi_1[\pi_2\pi_4 : \mathbf{true}]$ , and  $P_6 = \pi_1[\varepsilon : (P_4 \wedge P_5)]$ , the value  $Q[\text{depth}][6]$  can be **true** even when there is no occurrence of the XPath query  $\pi_1[\pi_2[\pi_3 \text{ and } \pi_4]]$ .

## 5.2 From Boolean to arithmetic functions

Each of the queries  $P_i$  can be viewed as a mapping that assigns truth-values to the XML-tree nodes. We extend this to assign integer values to the XML-tree nodes.

First, we extend the Boolean expressions  $e^i$  to the arithmetic expressions over the integers. The truth-values **true** and **false** are represented with 1 and 0, and the logical connectives  $\wedge$ ,  $\vee$ , and  $\neg$  are interpreted as appropriate arithmetic functions on the integers. Inequalities are also viewed as arithmetic functions that return 1 or 0.

Second, if the query  $P_i = \pi_1[\pi_2 : e^i]$  occurs at node  $x$ , the mapping  $P_i$  assigns to  $x$  the summation of the integer values obtained by evaluating at node  $y$  the arithmetic expressions  $e^i$  over all the nodes  $y$  such that the pattern  $P_i$  occurs at locus  $(x, y)$ . To the nodes  $x$  at which the query  $P_i$  does not occur, the mapping  $P_i$  assigns 0.

**Aggregations.** Consider the XPath query  $\pi_1[\text{count}(\pi_2) > 1]$ . We replace the statement “If  $e$  is **true**, then set  $Q[\text{depth} - d][q]$  to **true**” of the algorithm stated in Section 3.4 with “Increment  $Q[\text{depth} - d][q]$  by  $e$ ”. Let  $P_1 = \pi_1[\pi_2 : 1]$  and  $P_2 = \pi_1[\varepsilon : (P_1 > 1)]$ . Then the XPath query  $\pi_1[\text{count}(\pi_2) > 1]$  occurs if and only if  $Q[\text{depth}][2] > 0$ .

By a similar idea, we can process the other aggregation functions such as *sum*, *max*, *min*, and *avg* (average).

## 6 Conclusions

We have presented XAXEN, a light-weight streaming XML filtering tool, based on path pattern matching on path trie and keyword search using Aho-Corasick’s automaton over binary XML file. Experimental results show that XAXEN is approximately 2 ~ 6 times faster than XMLTK and YFilter and its memory requirement is not larger than

1/6 of those of XMLTK and YFilter. We note that XAXEN deals only with forward axes (i.e., the child and descendant axes) as YFilter and XMLTK. Recently, there are some proposals which deal with the ancestor axes and the sibling axes (see, e.g., [5]) at sacrifices of scalability and speed. XAXEN can be extended to process queries with ancestor axes and sibling axes by transforming the queries into the ones without such axes using path trie.

## References

- [1] Filtering and transformation for high-volume xml message brokering. [http://yfilter.cs.berkeley.edu/code\\_release.htm](http://yfilter.cs.berkeley.edu/code_release.htm).
- [2] Report from the w3c workshop on binary interchange of xml information item sets. <http://www.w3.org/2003/08/binary-interchange-workshop/Report.html>, 2003.
- [3] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination. In *VLDB’00*, pages 53–64, 2000.
- [4] I. Avila-Campillo, T. J. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suci. XMLTK: An XML toolkit for scalable XML processing. In *PLANX’02*, 2002.
- [5] C. Barton, P. Charles, D. Goyal, M. Raghavachari, V. Josifovski, and M. Fontoura. Streaming xpath processing with forward and backward axes. In *ICDE*, pages 455–466, 2003.
- [6] Y. Chen, G. A. Mihaila, S. B. Davidson, and S. Padmanabhan. EXPedite: A system for encoded XML processing. In *CIKM’04*, pages 108–117, 2004.
- [7] Y. Diao, H. Altinel, M. J. Franklin, H. Zhang, and P. M. Fischer. Path sharing and predicate evaluation for high-performance. In *ACMTOD*, 2003.
- [8] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB’97*, pages 436–445, 1997.
- [9] M. Ley. DBLP computer science bibliography. <http://dblp.uni-trier.de/>.
- [10] G. Navarro and M. Raffinot. *Flexible pattern matching in strings: Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.
- [11] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *SIGMOD’03*, pages 431–442, 2003.
- [12] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB’02*, pages 974–985, 2002.
- [13] M. Takeda, A. Ishino, and S. Mitarai. A light-weight xml query processor for a large number of structural and textual patterns. Technical Report DOI-TR-CS-226, Department of Informatics, Kyushu University, July 2006.