Time/Space Efficient Filtering of Streaming XML Documents Using Incrementally Constructed c

Hagio, Kazuhito
Kyushu University

Mitarai, Shuichi
Kyushu University

Ishino, Akira
Tohoku University

Takeda, Masayuki
Kyushu University

出版情報：DOI Technical Report. 231, 2007-06. 九州大学大学院システム情報科学研究院情報理学部門
バージョン：
権利関係：

Time/Space Efficient Filtering of Streaming XML Documents Using Incrementally Constructed Path-trie

Kazuhito Hagio
Kyushu University
kazuhito.hagio@i.kyushu-u.ac.jp

Shuichi Mitarai
Kyushu University
mitarai@cc.kyushu-u.ac.jp

Akira Ishino
Tohoku University
ishino@ecei.tohoku.ac.jp

Masayuki Takeda
Kyushu University
takeda@i.kyushu-u.ac.jp

Abstract

In this paper, we present a streaming XML document filter named DXAXEN which is based on incremental construction of path-trie. It runs very fast, and processes a large number of XPath queries efficiently. Experimental comparison with XMLTK, a well-known streaming XML document filter, shows that DXAXEN is 2–5 times faster and needs only 5–20 percent of memory.

1 Introduction

Querying XML streams is receiving much attention due to its growing range of applications such as stock and sports tickers, traffic information systems, electronic personalized newspapers, and entertainment delivery. Existing approaches to querying XML streams assume that user interests are written as queries using XPath [7], a language for specifying the selection of element nodes within XML documents. It is central to most XML-related technologies such as XQuery, XSLT, and XML Schema, under the auspices of W3C.

The deterministic finite-state automata (DFAs) can be used effectively to evaluate a large collection of XPath expressions on streaming XML data at high throughput, compared with the nondeterministic finite-state automata (NFAs). However, most of existing XPath evaluators on XML streams such as XFilter [2], XTrie [5], YFilter [8] avoided using DFAs because their size grows exponentially with respect to the total size of XPath expressions. The lazy DFA technique [13] is a good solution to the state explosion problem. A *lazy* DFA is one whose states and transitions are computed from the corresponding NFA at runtime, not at compile time. Thanks to this technique, XMLTK [4] processes a large collection of XPath expressions on streaming XML data at guaranteed throughput.

Table 1. Examples of XPath expressions DXAXEN supports.

//receiver/name	linear path-pattern
/*/order[//sender]	primitive path-pattern
/order//[contains(name, "mickey")]	substring match
/order//address[street or region]	logical operation
//address[//region[//country and zipcode]]	nested predicates
/order[//address]//zipcode	predicate in any location step
//sender[count(//region) > 2]	aggregation

Contribution We present a new XPath filtering technique based on *path-trie*, a trie representing the set of sequences of tagnames on the root-to-leaf paths. It incrementally constructs path-trie, and using it as a DFA, it evaluates a large number of XPath queries in parallel during single pass through an XML data. We call our technique DXAXEN (dynamic XAXEN; XAXEN [18] is an abbreviation for "eXtremely-Accelerated XML filtering ENgine"). Compared with the lazy DFA, our DFA is much simpler and more time/space efficient. Although our DFA has often a larger number of states than the lazy DFA does, the number of states of the two DFAs are bounded by the path-trie size plus one, which is known to be typically very small for XML data that has a fairly regular structure.

XPath fragment DXAXEN supports A *linear* XPath expression is one where (i) wildcards ('*') are allowed in node test, (ii) only child and descendant axes (resp. '/' and '//') are allowed in location steps (parent, ancestor and other axes are forbidden) and (iii) no predicates are allowed in loca-

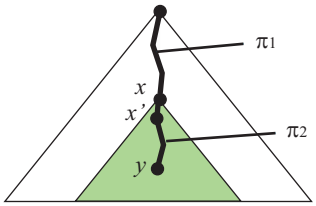


Figure 1. An occurrence of primitive path-pattern $\pi_1[\pi_2]$ at locus (x, y) of XML tree is illustrated, where x' is the child of x on the x-to-y path.

tion steps. Some of existing XPath evaluators on XML streams (e.g., XFilter, YFilter, and XMLTK) focus mainly on the linear fragment of XPath. XPath expressions with predicates are processed by decomposing them into a set of linear XPath expressions, evaluating them respectively, and then combining the results appropriately. In contrast, we focus on XPath expressions which possibly have predicates in a restricted way: A *primitive path-pattern* is a linear XPath expression π_1 followed by a single predicate containing a linear XPath expression π_2, namely it is of the form $\pi_1[\pi_2]$. The reason why we distinguish the predicate-containing XPath expressions of this type from the ones of general type will be clarified in Section 6.

For any node x of XML tree and any descendant y of x (possibly $x = y$), a primitive path-pattern $\pi_1[\pi_2]$ *occurs* at locus (x, y) if π_1 and π_2, respectively, match the tagname strings of the root-to-x path and the x'-to-y path, where x' is the child of x on the x-to-y path. Fig. 1 shows an occurrence of primitive path-pattern $\pi_1[\pi_2]$ at locus (x, y). We note that a linear XPath expression can be viewed as a primitive path-pattern $\pi_1[\pi_2]$ with $\pi_2 = \varepsilon$, for which no x' exists and $x = y$.

DXAXEN supports a large fragment of XPath based on efficient evaluation techniques for primitive path-patterns presented in this paper, although a detailed discussion is outside the scope of this paper. Some expressions belonging to the fragment are demonstrated in Table 1, which will be briefly mentioned in Section 6. It is worth to mention here that though the examples do not include ones with other axes than child and descendant, it is easy to deal with other forward axes such as following, following-self, and following-sibling, and XPath expressions with backward axes can be transformed into equivalent ones with forward axes only, of size linear with their original size, based on the technique by Olteanu et al. [20].

Paper organization The present paper is organized as follows. Section 2 mentions some relative work. Section 3 how path-trie can be used in path matching. Section 4 then describes a method of efficient filtering based on incremental construction of path-trie. Section 5 reports experimental comparison of DXAXEN with XMLTK. Section 6 mentions an extension to more complex XPath expressions. Section 7 concludes this paper.

2 Related work

Complexity of XPath evaluation Gottlob et al. [11] pointed out that existing systems consume time exponential with respect to query size, and showed the first polynomial-time algorithms for the XPath 1.0 language. Also they presented two fragments of XPath for which linear-time algorithms exist.[1] The same authors showed in [12] that the combined complexity (i.e., both the size of the data and the query) for the *full* XPath 1.0 language is P-hard, and then identified a large and important fragment for which the combined complexity is LOGCFL-complete. XPath 1.0 supports a number of powerful modalities and it is rather expensive to process. In practice, many applications do not need the expressive power of the full language and use only a fragment of XPath. It is thus required to develop efficient algorithms for such fragments of XPath language.

Efficient XML data stream filtering The requirement is more severe in streaming XML data filtering, where a large number of XPath queries should be evaluated against XML data streams at guaranteed throughput. The problem of evaluating large collections of XPath queries on streaming XML data was first introduced in the work of Altinel and Franklin [2]. They proposed a publish-subscribe system named XFilter. Improved techniques were discussed in XTrie by Chan et al. [5], lazy DFAs by Green et al. [13] (implemented as XMLTK [4]), and YFilter by Diao et al. [8]. Basically they are multi-query evaluation techniques for the linear XPath fragment, and some of them can be extended to support a larger XPath fragment.

Deterministic/nondeterministic finite-state automata play key role in processing XPath expressions of the linear XPath fragment. XFilter, XTrie, and YFilter explicitly avoided using DFAs to guarantee their space bounds, but they had no guarantee on throughput. DFAs effectively process a large collection of XPath expressions at guaranteed throughput, but their size grows exponentially with respect to the total size of XPath expressions. (See Fig. 2.) The lazy DFA technique is a solution to the state explosion problem: States and transitions are computed from the corresponding NFA at runtime, not at compile time. The number of states in the lazy DFA was proved in [13] to be at most the size of path-trie plus one. XMLTK runs at

[1] By *linear* $O(|D| \cdot |Q|)$ running time is meant, where $|D|$ and $|Q|$ denote the size of the XML data and the queries, respectively.

high throughput based on the lazy DFA technique. Further reduction of the number of states was addressed in [21]. Although the lazy DFA has a small number of states, its memory usage is not very small. The reason follows. Each of the lazy DFA states is represented as a list of states of the corresponding NFA, and all the lists are kept for lazy construction of DFA. Space requirement for NFA state lists is the dominant in total space requirement for the lazy DFA as reported in [13]. Also, when building a new transition from a DFA state s, the list of next states is computed from the NFA state list of s and then compared with all the lists of existing DFA states in order to determine whether the corresponding DFA state is already created. This comparison task is time consuming as pointed out in [6].

Our approach In a previous work [15], we proposed a method which uses the path-trie explicitly. In the method, the path-trie is built by scanning whole XML data, outputs are added to the path-trie nodes by running NFAs for queries along the root-to-leaf paths in it, and then the XML data is processed again using the output-added path-trie as a DFA. This might seem far from stream processing since the data stream is read twice. But it is not so in a setting where the path-trie is created only once, attached to the XML packet, then sent along with the XML stream, and used by every consumer. A similar setting can be found around SIX, the stream index technique proposed in [13].

In an XML message system, messages are often binary representations of XML, rather than XML data in their native text format (see, e.g., [9]). In [18], we proposed one such binary format: XML data is converted into the path-trie and the binary one in which every start-tag is replaced with a special symbol followed by a pointer to the corresponding path-trie node and every end-tag is replaced with another special symbol. The pointers enable us to skip performing state-transitions of path-trie as DFA: We have only to refer the outputs attached to the path-trie nodes. Our implementation was named XAXEN (eXtremely-Accelerated XML filtering ENgine).

The approach taken in this paper goes along another line. We do not preprocess XML data to produce the path-trie: We maintain a DFA based on the path-trie being constructed incrementally, and by using it, we evaluate queries during single pass through an XML data, similar to XMLTK.

3 Using path-trie

3.1 Path-tries and their size

An example XML data, its XML tree representation, and its path-trie are shown in Fig. 4. Formally, an XML tree is an ordered tree such that the interior nodes are labeled with *tagnames*, and the leaves, called text nodes, are labeled

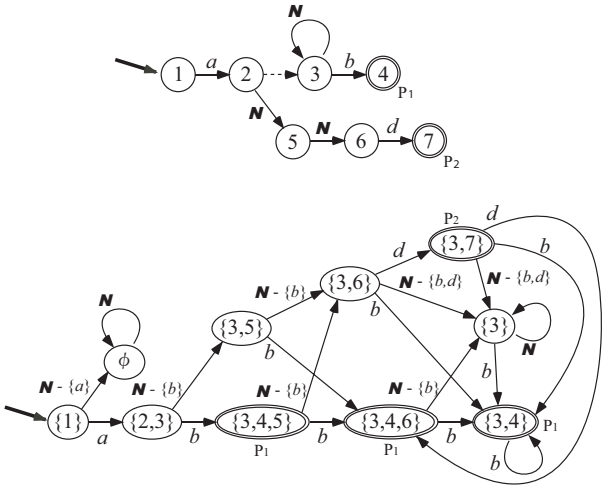


Figure 2. NFA and DFA for $\Pi = \{P_1, P_2\}$ are displayed on the upper and on the lower, respectively, where P_1 = a//b and P_2 = a/*/*/d.

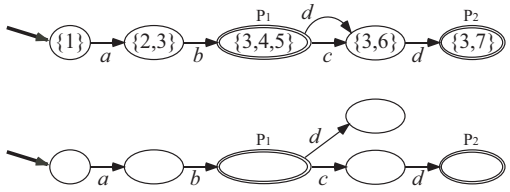


Figure 3. Lazy DFA and our DFA for the patterns of Fig. 2 are displayed on the upper and on the lower, respectively, where all possible tagname sequences are limited to $S = \{a, ab, abc, abd, abcd\}$.

with *text*.[2] Thus a path from an internal node u to another internal node v spells out a string of tagnames. Let us call it the *tagname string* of the u-to-v path. The *path-trie* of an XML tree is the trie structure that represents all the tagname strings of the XML tree. We note that path-trie is a kind of DataGuide [10], which is a graph representing the database structure, but it is simply a tree since the underlying data (i.e. XML data) has tree structure.

It is observed in [14] that most of existing XML data have DataGuide of small size, and therefore their path-trie are also small. Generally speaking, if an XML data is highly structured, then its path-trie becomes small accordingly. Table 2 gives the path-trie sizes and other information about the following XML data:

[2] 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".

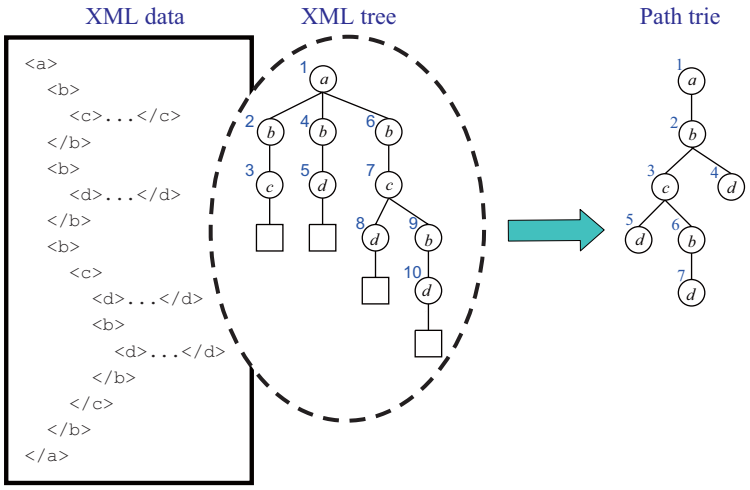


Figure 4. XML data and XML tree it represents are displayed on the left, where circles and squares mean element and text nodes, respectively. Path-trie built from the XML data is displayed on the right. Numbers adjacent to the nodes of XML tree and path-trie mean their IDs, and letters in the circles (nodes) are their tagnames.

Table 2. Path-trie size of three XML data are shown, together with the number of tag occurrences and the number of tagnames.

	# tag occurrences	# tagnames	path-trie size
(a)	198,887,920	92	160
(b)	22,215,944	41	166
(c)	4,096,360	83	549

(a) XML data of UniProtKB/Swiss-Prot [3]. 2.2GB.

(b) XML data of DBLP [16]. 377MB.

(c) XML data generated randomly by xmlgen program of XMark benchmark [23]. 116MB.

For (c), the XML tree has $4,096,360/2 = 2,048,180$ nodes (excluding text nodes), while the path-trie has only 549 nodes, which is 0.03% of the XML tree nodes. The ratios for (a) and (b) are much lower.

Next, we consider how the path-tries grow during their construction as sequentially processing an XML data. Fig. 5 shows the growth of path-trie for each of the three XML data, where we also display the increases of tag occurrences and tagnames.

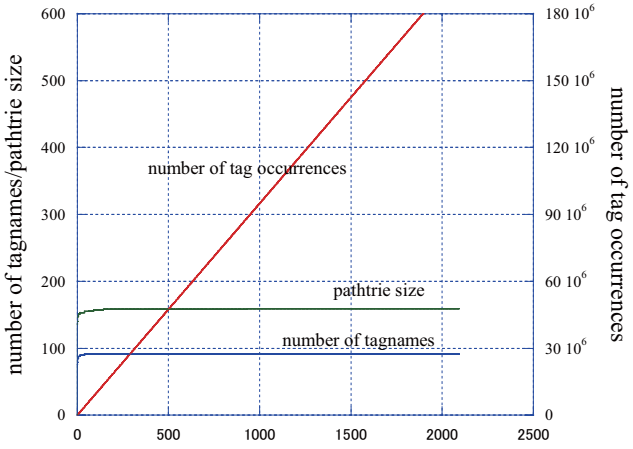
The data (a) has highly regular structure, and its path-trie has grown up at early stage. Concerning the data (b), the growth seems nearly stable except the early part and the point around 240MB at which the underlying scheme alters. For the data (c), the root has six children, which correspond to six different categories having their own schemes, and the path-trie size rapidly increases for every time the category switches. After about 57MB point, increase of tag occurrences got rapid.

3.2 Matching against path-trie, not XML tree

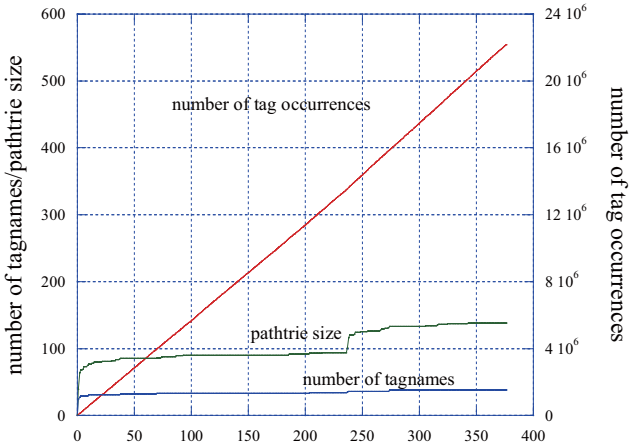
As seen in Section 3, path-tries are much smaller than their XML trees. Hence, performing path-matching against path-trie and then referring to the results stored in its nodes can be much more efficient than direct path-matching against XML tree. Below, we gives some illustrative examples.

Linear path-pattern case. Fig. 6 illustrates occurrences of the patterns P_1 = //b/c and P_2 = a/b//d within the XML data and path-trie of Fig. 4. The pattern P_1 occurs at node 3 and the pattern P_2 occurs at nodes 4, 5, 7 in the path-trie. We note that nodes 3, 7 of the XML tree correspond to node 3 of path-trie, and accordingly the pattern P_1 occurs at the nodes of the XML tree. A similar observation is obtained for the pattern P_2.

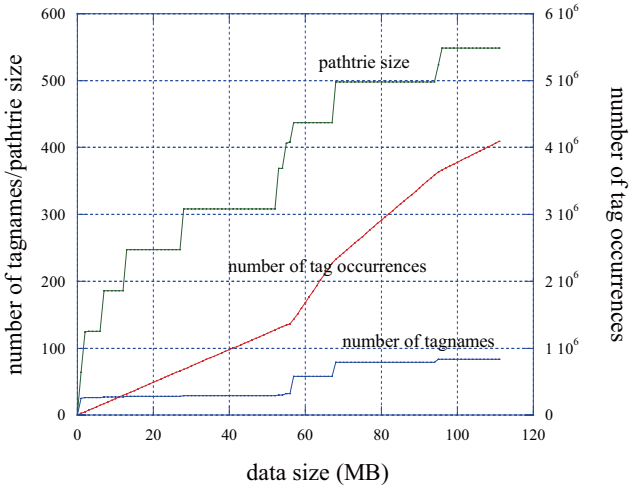
Primitive path-pattern case. Fig. 7 illustrates occurrences of the pattern P = a/b[//d] within the XML data and path-trie of Fig. 4. The pattern P occurs at loci $(2,4),(2,5),(6,7),(2,7)$ in the path-trie, and the loci can be obtained from output $\langle P,1\rangle$ of node 4, output $\langle P,2\rangle$ of node 5, and outputs $\langle P,1\rangle,\langle P,3\rangle$ of node 7, respectively. From the outputs, the loci of occurrences of P within XML



(a) UniProtKB/Swiss-Prot



(b) DBLP



(c) random data

Figure 5. Growth of path-tries when sequentially scanning XML data are illustrated.

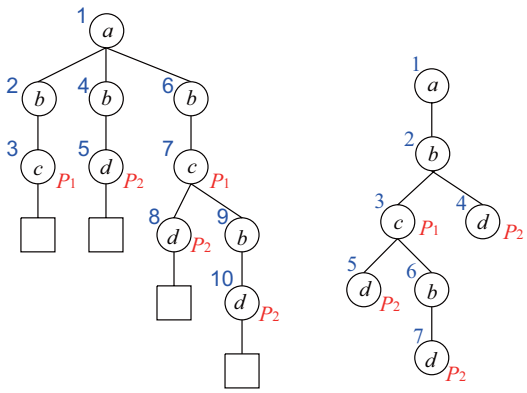


Figure 6. Occurrences of linear path-patterns P_1 = //b/c and P_2 = a/b//d in XML tree and path-trie are displayed. For instance, the occurrence of P_1 at node 3 of path-trie implies that P_1 occurs at the corresponding nodes 3 and 7 of XML tree.

tree can be obtained. For instance, since node 5 corresponds to node 4 of path-trie with output $\langle P, 1\rangle$, it has the same output. This implies that P occurs at locus $(4, 5)$ of XML tree, where 4 is the parent (the 1st ancestor) of 5.

4 Filtering with path-trie built incrementally

In this section, we describe a filtering method that sequentially scans an XML data stream and performs path-matching over incrementally constructed path-trie. We remark that the size of path-trie could be reduced by regarding tagnames not appearing in patterns as one tagname.

4.1 Construction of path-trie

We parse input XML data sequentially to detect and report occurrences of tags as events, which are to be input to incremental path-trie construction and to path-matching. We maintain a trie representing the 'current' set of tagnames: If a current tagname is present in the set, then its ID is returned; and otherwise a new ID is assigned to it and then returned.

We illustrate the algorithm by using the XML data and the path-trie of Fig. 4. We note that sequential processing of an XML data stream gives a depth-first-traversal of the corresponding XML tree. We start with an empty path-trie, and proceed as follows. Suppose we have processed the nodes 1 to 7 in the traversal of the XML tree. At this moment, the path-trie has nodes 1 to 4 since the path strings of the XML tree nodes 1 to 7 are, respectively,

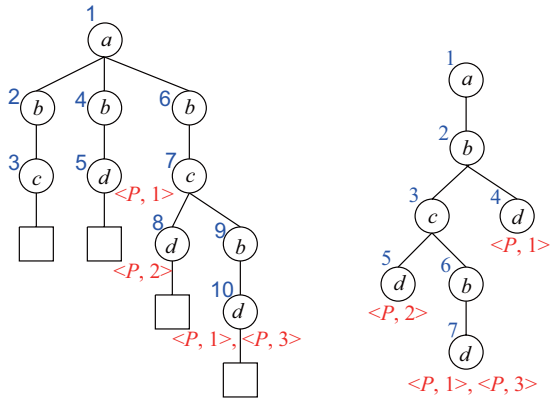


Figure 7. Occurrences of primitive path-pattern P = a//b[//d] in XML tree and path-trie are displayed, where the pairs $\langle P, d\rangle$ adjacent to nodes y represent the occurrences of P at (x, y) such that x is the d-th ancestor of y. For instance, $\langle P, 1\rangle$ at node 4 of path-trie implies that P occurs at locus $(2, 4)$ of path-trie, and at locus $(4, 5)$ of XML tree.

$a, ab, abc, ab, abd, ab, abc$, and these are represented by the path-trie nodes $1, 2, 3, 2, 4, 2, 3$, respectively. Thus, we are now at node 7 of XML tree and at node 3 of path-trie. The next node of XML tree is 8 and the corresponding tagname is d. Since the path-trie node 3 does not have a child with tagname d, a new node, numbered 5, is created as a child of 3. Thus a new path from 1 to 5 appears. We do path-matching tasks for this new path for all queries.

Repeating the above process until the entire XML data is passed through, we incrementally build the path-trie and perform path-matching against it in parallel.

4.2 Matching against growing path-trie

This subsection describes path-matching algorithms executed when a new node is added to path-trie.

4.2.1 Linear path-pattern case

Suppose we are given a set Π of linear path-patterns. Denote by $\|\Pi\|$ the total length of the patterns in Π, and let $\mathcal{N}$ be the set of tagnames appearing in patterns of Π. Two possible implementations are considered here.

DXAXEN-A: One implementation would be to build an NFA from Π in a way similar to YFilter [8]. A trie is built from Π regarding tagnames, '*', and '//' as symbols, and for every edge labeled '//', the label is replaced with ε and a loop labeled with '*' is added to its sink node to produce an NFA. The NFA construction requires $O(\|\Pi\| \cdot \log |\mathcal{N}|)$ time using $O(\|\Pi\|)$ space, where the factor $\log |\mathcal{N}|$ comes from use of the famous AVL-tree for lookup/insertion against the set of transitions from an NFA state. We let path-trie nodes v have lists of active states of NFA just after reading the tagname string of the root-to-v path. When a new path-trie node with tagname $a \in \mathcal{N}$ is created, we shall perform nondeterministic state-transitions on a to create an active-states list for it. We note that each NFA state has at most one a-transition for every $a \in \mathcal{N}$, at most one ε-transition, and at most one $*$-transition from it. Finding a-transition from it requires $O(\log |\mathcal{N}|)$ time. Creation of active-states list for a new path-trie node v requires $O(\|\Pi\| \cdot \log |\mathcal{N}|)$ time and $O(\|\Pi\|)$ space. This implementation is very similar to the lazy DFA in that lists of active states are kept.

DXAXEN-B: Another implementation would be to build an NFA for each linear path-pattern $\pi \in \Pi$, by applying the bit-parallel technique [19], in which a set of active states of NFA is stored in one integer as a bit-vector. The number of all states of the NFA is $size(\pi) + 1$, where $size(\pi)$ is defined to be the number of node tests in π. Thus, the technique can be used only for linear path-patterns of size at most the word length (typically 32 or 64 bits) minus one. Since construction of each NFA requires $O(|\pi| \cdot |\mathcal{N}|)$ time using $O(|\mathcal{N}|)$ space and we build $|\Pi|$ NFAs, totally we need $O(\|\Pi\| \cdot |\mathcal{N}|)$ time using $O(|\Pi| \cdot |\mathcal{N}|)$ space. Note that if the bit-vector of an NFA consists only of '0', the NFA is dead (never accepts). Thus, each path-trie node v has a list of pairs of an 'alive' NFA and its bit-vector just after having processed the root-to-v path. Creation of such a list for a new path-trie node requires only $O(|\Pi|)$ time and space.

4.2.2 Primitive path-pattern case

Denote by $L(\pi) \subseteq \mathcal{N}^*$ the language of linear path-pattern π. Suppose we are given a set Π of primitive path-patterns. Let $\pi_1[\pi_2]$ be any of Π. Suppose a sequence $v_1, \ldots, v_k$ of nodes is a path from the root to v_k in path-trie, and a new node v_{k+1} is created as a child of v_k. For this newly appearing path $v_1, \ldots, v_{k+1}$, we have to check whether $\pi_1\pi_2$ matches it, and in case of match, we have to find i such that π_1 and π_2, respectively match the v_1-to-v_i path and the v_{i+1}-to-v_{k+1} path. To implement this, we consider an NFA that recognizes the two languages $L(\pi_1)$ and $L(\pi_1\pi_2)$ with two distinct accepting states. Let us call it the *forward NFA* for $\pi_1[\pi_2]$. The forward NFA is easily constructed by building two NFAs with a unique initial state and a unique accepting state, that recognize $L(\pi_1)$ and $L(\pi_2)$, respectively, and then combining them with an ε-transition from the accepting state of NFA for $L(\pi_1)$ to the initial state of NFA for $L(\pi_2)$. Fig. 8 gives an example. We note that a set of forward NFAs can be merged into a single NFA, similar to the NFA discussed in DXAXEN-A. We also note that a forward NFA can be implemented by the bit-parallel tech-

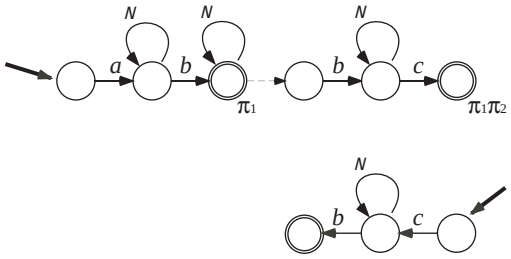


Figure 8. Forward NFA for $\pi_1[\pi_2]$ is displayed on the upper, where π_1 = a//b// and π_2 = b//c. It recognizes distinctively the two languages $L(\pi_1)$ and $L(\pi_1\pi_2)$, and is implemented as a combination of two NFAs recognizing $L(\pi_1)$ and $L(\pi_2)$, respectively. Backward NFA is displayed on the lower, which accepts the set of reversals of strings in $L(\pi_2)$.

nique, similarly in DXAXEN-B. In both implementations, the sets of active states are kept in path-trie nodes, and therefore we can find i for which π_1 match the v_1-to-v_i path. However, it should be emphasized that there is no guarantee that π_2 matches the v_{i+1}-to-v_{k+1}, even though we know $\pi_1\pi_2$ matches the v_1-to-v_{k+1} path. To confirm this, we use another NFA which recognizes the languages of consisting of the reverse strings of $L(\pi_2)$. We call such an NFA the *backward* NFA for $\pi_1[\pi_2]$. The backward NFA is run along the v_1-to-v_{k+1} path backward whenever $\pi_1\pi_2$ matches the path. We can merge a set of backward NFAs into a single one if necessary. An example of the backward NFA is displayed in Fig. 8.

In DXAXEN-A implementation, time and space complexity of constructing forward NFA and backward NFA are the same as those of constructing NFA for linear path-pattern case. Time complexity of running the forward NFA and the backward NFA for a new path-trie node to compute its output is h-multiple of the time complexity of running NFA for linear path-pattern case, where h is the height of XML tree. Space complexity is the same as the linear path-pattern case. In DXAXEN-B implementation, time and space complexities of constructing forward NFAs and backward NFAs are the same as linear path-pattern case. Time complexity of running the forward NFAs and the backward NFAs for a new path-trie node is again h-multiple of the time complexity for the linear path-pattern case. Space complexity remains unchanged.

4.3 Total time and space complexity

Let D be input XML data whose XML tree has size s and height h, and whose path-trie has size t. Let Π be input set of linear/primitive path-patterns. When the DXAXEN-A implementation is adopted, it runs in $O(|D|+s\cdot\log|\mathcal{N}|+t\cdot\|\Pi\|\cdot\log|\mathcal{N}|)$ time using $O(t\cdot\|\Pi\|)$ space. The factor $\log|\mathcal{N}|$ is dropped if the hashing technique substitutes for the AVL-tree and works well as assumed in the lazy DFA [13]. For primitive path-pattern case, the factor $t\cdot\|\Pi\|\cdot\log|\mathcal{N}|$ in the time complexity is multiplied by h.

When the DXAXEN-B implementation is adopted, it runs in $O(\|\Pi\|\cdot|\mathcal{N}|+|D|+s\cdot\log|\mathcal{N}|+t\cdot|\Pi|)$ time using $O(|\mathcal{N}|\cdot|\Pi|+t\cdot|\Pi|)$ space, under the assumption that for each pattern π in Π, $size(\pi)+1$ fits word length. For primitive path-pattern case, the factor $t\cdot|\Pi|$ in the time complexity is multiplied by h again. We note that the factor $\log N$ assumes use of the AVL-tree, and is dropped if the hashing technique works effectively.

For the lazy DFA, the time and space complexity is similar to DXAXEN-A, except that cost of searching the NFA-state lists associated with existing DFA states for a newly created list of NFA states should be considered. Since there could be $O(t)$ lists, each having size $O(\|\Pi\|)$, the time complexity is $O(t\cdot\|\Pi\|)$. Totally, the lazy DFA based method runs in $O(|D|+s\cdot\log|\mathcal{N}|+t\cdot\|\Pi\|\cdot(\log|\mathcal{N}|+t))$ time using $O(t\cdot\|\Pi\|)$ space, where the factor $\log|\mathcal{N}|$ can be dropped assuming the hashing technique.

5 Computational Experiment

We carried out a series of computational experiments against the two XML data: XML records of DBLP and the random data of XMark benchmark, stated in Section 3. Since XMLTK deals only with the linear XPath fragment, we restricted ourselves to it in the experiments. We randomly generated test sets of linear path-patterns by using pathgenerator [3] with DTDs of the two XML data sets.[4] We did not use the XML data of UniProtKB/Swiss-Prot in the experiments since no DTD is provided with it.

All experiments were executed on 2.4 GHz Intel Pentium 4, 2.0 GB RAM, running Red Hat Linux Advanced Server 2.1. Below, we report the result of performance comparison of DXAXEN with XMLTK in the three points: memory usage, processing time and throughput. Although implementations of XFilter, YFilter, XSQ are available, we excluded them in the comparison since their throughput is worse than that of XMLTK as reported in [22, 18].

Memory usage comparison Table 3 compares size of lazy DFA and reduced path-trie. The lazy DFA has smaller number of states in comparison with our DFA. The total size of NFA-state lists of lazy DFA states grows as NFA size in-

[3] http://yfilter.cs.berkeley.edu/code_release.htm

[4] We generated pattern lists of size 1, 10, ..., 100000, but there are duplications. The numbers of distinct patterns were smaller.

Table 3. Size of lazy DFA and our DFA are compared against the DBLP data and the random data.

(a) Random data whose path trie has 549 nodes.

# queries	lazy DFA		our DFA
	# states	NFA state lists	# nodes
1	4	4	16
10	23	57	122
100	103	574	534
1,000	263	12,392	542
10,000	504	167,479	542
100,000	533	1,860,257	542

(b) DBLP data whose path trie has 166 nodes.

# queries	lazy DFA		our DFA
	# states	NFA state lists	# nodes
1	4	4	12
10	13	37	49
100	68	583	156
1,000	119	6,397	156
10,000	150	80,169	156
100,000	151	791,589	156

Table 4. Memory usage comparison.

# queries	memory usage (KB)		
	DXAXEN-A	DXAXEN-B	XMLTK
1	4,692	4,680	984
10	4,736	4,708	1,044
100	5,452	4,852	1,476
1,000	8,028	5,444	5,200
10,000	14,400	7,268	38,780
100,000	35,216	17,452	369,036

creases. The total size of active-state lists stored in path-trie nodes grows similarly.

Table 4 compares the memory usage of the methods applied to the random data with linear path-patterns of size $1, 10, \ldots, 100000$. DXAXEN-B consumes $1/2$ of memory required by DXAXEN-A. DXAXEN-B consumes $1/5$ and $1/20$ of memory that XMLTK requires for 10,000 and 100,000 queries. Thus, DXAXEN is much more space-efficient than XMLTK when a large number of linear path-patterns are dealt with.

Running time comparison Table 5 compares the processing times against linear path-pattern sets of size $1, 10, \ldots, 100000$. Compared to XMLTK, DXAXEN-B is approximately 1.5 to 7 times faster and 2 to 6 times faster against the DBLP data and the random data, respectively, and the ratios become larger as the number of patterns increases.

Table 5. Running time comparison.

(a) Random data from XMark

# queries	elapsed time (sec)		
	DXAXEN-A	DXAXEN-B	XMLTK
1	1.46	1.49	2.22
10	1.50	1.49	2.58
100	1.63	1.58	3.04
1000	1.77	1.79	4.12
10,000	2.87	2.85	11.56
100,000	49.13	39.79	283.34

(b) DBLP

# queries	elapsed time (sec)		
	DXAXEN-A	DXAXEN-B	XMLTK
1	5.81	5.82	11.34
10	6.15	6.05	11.29
100	6.89	6.72	18.06
1000	7.38	7.30	21.26
10,000	15.58	14.98	59.95
100,000	175.71	195.37	1242.58

Throughput comparison Fig. 9 compares the throughputs for sets of 100 and 10,000 queries against DBLP and the random data, where the X-axis is the size of data already processed from the left. Results show that DXAXEN-A and -B outperforms XMLTK in throughput comparison.

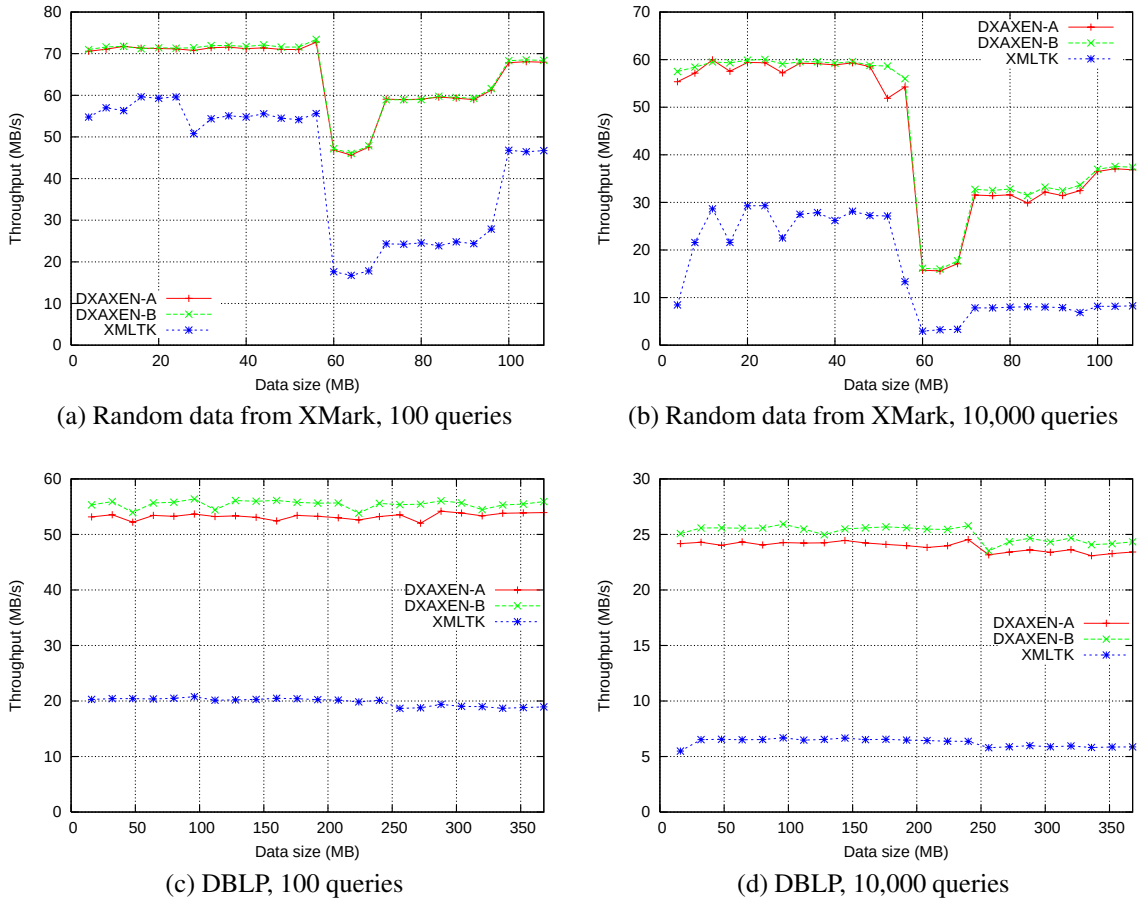
Low throughput of XMLTK at early stage implies that it is in "warm-up phase", in which many states and transitions of lazy DFA are created. This corresponds to rapid increase of path-trie size at early part we have already seen in Fig. 5. Compared with XMLTK, DXAXEN-A and -B in their warm-up phase have relatively high throughput.

For the random data (see (a) and (b)), all methods cannot have a stable throughput. Lower throughput after 57MB point correspond to rapid increase of tag occurrences we have seen in Fig. 5 (c). We note that a work of path-matching is performed only when a tag occurrence is found in all methods.

For the DBLP data (see (c) and (d)) which has a regular structure, the throughputs of the three method are stable and high. In this case, DXAXEN-A and -B showed approximately 4 times higher throughput than XMLTK does.

6 Complex XPath expressions

So far, we introduced a class of primitive path-patterns, which are regarded as primitives for more complex XPath



(a) Random data from XMark, 100 queries

(b) Random data from XMark, 10,000 queries

(c) DBLP, 100 queries

(d) DBLP, 10,000 queries

Figure 9. Throughput comparison.

expressions. Although a detailed discussion is outside the scope of this paper, we shall briefly sketch how this XPath fragment could be primitives.

Consider a series of assignments:

$$Q_1 = \pi_1^1[\pi_2^1 : expr_1]; \ldots; Q_n = \pi_1^n[\pi_2^n : expr_n]$$

where Q_i is a logical variables, and $expr_i$ is a boolean expression over keywords or one over $Q_1, \ldots, Q_n$.

Substring match: DXAXEN does not use any existing XML parser such as SAX [17]: It simultaneously performs keyword search and XML parsing, that is, Aho-Corasick' pattern matching machine (PMM for short) [1] is built and run over XML data stream, and when the beginning of tag ("<") is detected our own parser is invoked to detect a tagname string and process attribute expressions. A drastically fast XML parsing is made possible in this way. Since PMM simultaneously recognizes multipattern occurrences, we can process boolean expressions over multiple keywords.

Logical operation: Consider Q = /order//address [street or region]. Let Q_1 = /order//address[street:true] and Q_2 = /order//address[region:true]. A locus of Q can be obtained as a locus of Q_1 or Q_2. Therefore Q can be expressed as Q_3 = /order//address[ε: $Q_1 \vee Q_2$].

Nested predicates: Consider Q = //address[//region [//country and zipcode]]. Let Q_1 = //address//region[//country: true] and Q_2 = //address//region[zipcode:true]. Then Q is represented as Q_3 = //address[//region: $Q_1 \wedge Q_2$].

Predicate in any location steps: Consider Q = /order[//address]//zipcode. Let Q_1 = /order[//address: true]. Q is expressed as $Q_2 = Q_1$//zipcode[ε:true], if we allow such expressions. This expression can be evaluated by using a lazy evaluation technique.

We note that Q_i is a mapping that assigns truth-values to XML-tree nodes. We extend this to assign integer values to them, and allow $expr_i$ be an arithmetic expression, not a boolean expression. For $Q_i = \pi_1[\pi_2 : expr_i]$, the mapping Q_i assigns to an XML-tree node x the summation of the integer values obtained by evaluating at node y the arithmetic expression $expr_i$ over all the nodes y such that $\pi_1[\pi_2]$ occurs at locus (x, y). To the nodes x such that there is no y

such that $\pi_1[\pi_2]$ occurs at (x, y), the mapping Q_i assigns 0. Then, we can deal with aggregation functions as follows.

Aggregation: Consider $Q =$ /sender[count(//region)>2]. Let $Q_1 =$ /order[//address: 1]. Then Q is expressed as $Q_2 =$ /order[ε: $(Q_1 > 2)$].

Thus, we can deal with complex XPath expressions such as //b[//d]//c[//c][d]//e, based on efficient processing technique for primitive path-patterns. XPath expressions with backward axes can be dealt with after transforming equivalent ones without backward axes [20].

7 Discussion

Let $\mathcal{N}$ be a set of tagnames, and let Π be a set of linear XPath expressions. We note that the elements of Π are regular expressions over $\mathcal{N}$. DFAs recognizing the language $L(\Pi)$ could have exponential number of states.[5] The lazy DFA is one that recognizes a *superset* of $L(\Pi) \cap S$, not $L(\Pi)$, where S is the set of tagname strings of the paths from the root.

The output-added path-trie of DXAXEN can be regarded as a DFA, which recognizes exactly the language $L(\Pi) \cap S$. The graph structure of the DFA depends only on XML data, not on XPath queries (unless the reduction technique mentioned in Section 4 is applied to), and the NFA recognizing $L(\Pi)$ is used to add outputs to the DFA states. In contrast, the lazy DFA is built from NFA that recognizes $L(\Pi)$, basically with the standard subset construction. For this reason, the DFA construction time of our method is much smaller than that of the lazy DFA. The size of lazy DFA is upper-bounded by the size of path-trie representing S.

Using the lazy DFA, we have to pay a relatively high cost in computing a state and a transition a runtime, although the cost is recovered when the state and the transition are reused. The cost is mainly devoted to decrease the DFA size. It, however, approaches the path-trie size as the number of queries grows. DXAXEN requires a lower cost to compute a state and a transition.

References

[1] A. V. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. ACM*, 18(6):333–340, 1975.

[2] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB'00*, pages 53–64, 2000.

[3] R. Apweiler, A. Bairoch, C. H. Wu, W. C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, M. J. Martin, D. A. Natale, C. O'Donovan, N. Redaschi, and L.-S. L. Yeh. UniProt: the universal protein knowledgebase. 32:D115–D119, 2004.

[4] I. Avila-Campillo, T. J. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suciu. XMLTK: An XML toolkit for scalable XML processing. In *PLANX'02*, 2002.

[5] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. *VLDB Journal*, 11(4):354–379, 2002.

[6] D. Chen and R. K. Wong. Optimizing the lazy DFA approach for XML stream processing. In *Proc. 15th Australasian Database Conference (ADC2004)*, 2004.

[7] J. Clark and S. DeRose. *XML Path Language (XPath) Version 1.0*. W3C Recommendation 16 Nov. 1999. `http://www.w3.org/TR/xpath`.

[8] Y. Diao, M. Altinel, M. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM TODS*, 28(4):467–516, 2003.

[9] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, and A. Sundararajan. The BEA streaming XQuery processor. *The VLDB Journal*, 13(3):294–315, 2004.

[10] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB'97*, pages 436–445, 1997.

[11] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB'02*, pages 95–106, 2002.

[12] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *PODS'03*, 2003.

[13] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM TODS*, 29(4):752–788, 2004.

[14] L. H. and S. D. XMill: an efficent compressor for XML data. In *SIGMOD'00*, pages 153–164, 2000.

[15] A. Ishino and M. Takeda. A proposal for XQuery processor with deterministic automaton and path pruning. *DBSJ Letters*, 4(4), 2006. (in Japanese).

[16] M. Ley. DBLP computer science bibliography. `http://dblp.uni-trier.de/`.

[17] D. Megginson. SAX 2.0: Simple API for XML. `http://www.megginson.com/SAX/`.

[18] S. Mitarai, A. Ishino, and M. Takeda. Light-weight acceleration for streaming XML document filtering. In *SWOD'07*, 2007.

[19] G. Navarro and M. Raffinot. *Flexible pattern matching in strings: Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[20] D. Olteanu, H. Meuss, T. Furche, and F. Bry. Xpath: looking forward. In *XMLDM'02*, volume 2490, pages 109–127, 2002.

[21] M. Onizuka. Light-weight XPath processing of XML stream with deterministic automata. In *CIKM'03*, pages 342–349, 2003.

[22] F. Peng and S. S. Chawathe. XSQ: a streaming XPath engine. *ACM TODS*, 30(2):577–623, June 2005.

[23] 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.

[5] More precisely, DFAs have to not only recognize the language but also report which patterns in Π are found. Hence they should be Mealy machines.