A Light-Weight XML Query Processor for a Large Number of Structural and Textual Patterns

Takeda, Masayuki
Department of Informatics, Kyushu University

Ishino, Akira
Office for Information of University Evaluation Kyushu University

Mitarai, Shuichi
Department of Informatics, Kyushu University

出版情報：DOI Technical Report. 226, 2006-07. Department of Informatics, Kyushu University
バージョン：
権利関係：

A Light-Weight XML Query Processor for a Large Number of Structural and Textual Patterns

Masayuki Takeda
Department of Informatics
Kyushu University
6-10-1 Hakozaki, Fukuoka 812-8581, Japan
takeda@i.kyushu-u.ac.jp

Akira Ishino
Office for Information of University Evaluation
Kyushu University
6-10-1 Hakozaki, Fukuoka 812-8581, Japan
ishino.uoc@mbox.nc.kyushu-u.ac.jp

Shuichi Mitarai
Department of Informatics
Kyushu University
6-10-1 Hakozaki, Fukuoka 812-8581, Japan
mitarai@i.kyushu-u.ac.jp

Abstract

A light-weight XML query processor is presented which deals with structural and textual pattern queries. Unlike the existing index-based methods, it builds no indices. Unlike the existing stream-oriented methods, it preprocesses the input XML file to yield a path trie and a binary XML file, which are processed in path-pattern matching stage and in text-node searching stage, respectively. Our method is space efficient, and most suited for the situation that the input XML file is given in advance and not very frequently updated, and a large number of queries should be processed at a time. The performance is successfully reported in comparisons with existing XML query processors.

1 Introduction

Establishing basic techniques for processing queries written in XML query languages such as XPath and XQuery is of great importance from both theoretical and practical viewpoints. At the early stage of the XML query processing research, the main stream was the DOM-oriented approach, where a tree structure of the input XML document, called the DOM tree, is built and resides in main memory. Since the DOM tree is huge in size and consumes much time to build, the approach cannot be applied to XML files of moderate size, say 50-200MB.

The current XML query processing technologies can be divided into two groups. One is the index-based technique that concentrates on indexing the relationships between two XML-tree nodes such as the parent-child and the ancestor-descendant relations, basing on numbering schemes for elements, attributes, and structures and/or on the relational-database techniques. (See e.g. [35, 22, 16].) The other is the stream-oriented approach, where the input XML document is given as a data stream. (See e.g. [2, 5, 13, 7, 29, 18, 23, 28, 15, 12, 20, 10, 19, 27].) One important advantage of stream-oriented methods is space efficiency. The memory requirement depends only on the query size, not on the XML document size.

On the other hand, most of the past researches have focused on processing of structural XML queries and have paid little attention to a full-text search. However efficient searching in text nodes has recently emerged as an important research topic [34, 3, 33].

This paper presents a light-weight XML query processing method for structural and textual patterns. We concentrate on the case where the input XML document is given in advance and not very frequently updated, but a large number of queries should be processed at a time. Typical examples can often be found in Web search services, in which the update is done a few times a day, and the bulk of requests from clients arrive at peak time of one day. Simultaneously processing multiple queries is one good way to respond to a considerable amount of requests concentrated at peak times. In this paper, we try to develop a space-economical method which is suitable for the case.

Similar to the stream-oriented methods, we create no indices. But we are allowed to preprocess them to accelerate query processing since the input XML document is relatively 'static' in our setting. Our preprocessing method is

Table 1. Sizes of the path tries for DBLP [21] and for a randomly generated XML document using xmlgen [31] are demonstrated. Path tries are sufficiently smaller than the original XML documents in practice.

	XML document			path trie
	size (MB)	# tags	# nodes	# nodes
DBLP	324	35	7,947,321	136
random	111	83	2,048,180	549

quite simple. We read once the input XML file to construct a trie representing all the label strings of the paths from the root, which is called the *path trie*. We note that in practice the path trie is sufficiently small and fits main memory for moderate XML files. (See Table 1.) During the construction of path trie, we also build a *binary XML file* from the input XML document by replacing every occurrence of the start tags with a special code followed by a pointer indicating the corresponding node in the path trie and every occurrence of the end tags with another special code.

After the preprocessing, the algorithm processes a large number of queries by exploiting the path trie and the binary XML file. The query processing has two stages: path-pattern matching and text-node searching. In the path-pattern matching stage, we build NFAs from the path patterns, make them run along the paths of the path trie in a depth-first manner, and add to the path-trie nodes information about the loci at which path patterns occur. In the text-node searching stage, the algorithm reads once the binary XML file from the left to the right virtually traversing the corresponding XML tree in a depth-first manner, searches for keywords in text nodes by using Aho-Corasick's pattern matching machine [1] (PMM in short), and then reports all occurrences of query patterns by combining the outputs from PMM with the information added to the path-trie nodes.

It should be mentioned that, if the path-trie nodes have the lists of the corresponding regions in XML document, then it can serve as an index (see e.g. the strong DataGuides [11]). On the contrary, the path-trie nodes are referred to by the integers preceded by special code implying start tags during a full scan of the binary XML file in our method.

To evaluate the performance of our algorithm, we compare the performance of our algorithm with the following XML query processors:

XMLTK. A light-weight XML stream processor which can process a large number of path expressions at high throughput. For path-pattern matching, the "LazyDFA" technique [4] is adopted.

Tamino (ver. 4.1.4.1). A commercial product of native XML database system developed by Software AG. It supports a large part of XQuery.

NeoCore XMS (ver. 3.1). Another commercial product of native XML database system developed by Xpriori, basing on its patented Digital Pattern Processing (DPP) technology. It covers a large part of XQuery.

The experimental results show that our algorithm outperforms or is competitive with them.

The organization of this paper is as follows. Section 2 presents a formal statement of the problem to be addressed. Section 3 illustrates the preprocessing part of our algorithm. Section 4 and Section 5 are, respectively, devoted to describe the path pattern matching stage and the text node searching stage. Section 6 discusses how to extend our method to cope with more complex queries. Section 7 reports successful results in experimental comparisons of our algorithm with XMLTK, Tamino, and NeoCore XMS. Finally, Section 8 gives conclusions and future works.

2 Problem definition

Let Σ be a finite set of characters, and denote Σ^* (resp. Σ^+) be the set of strings (resp. nonempty strings) over Σ. 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 Σ^*.[1]

A *simple path pattern* is a sequence consisting of tag names and *'s, separated by "/" or "//", where "/" and "//" correspond to the parent-child and the ancestor-descendant relationships, respectively. A simple path pattern π is said to *match* a path in an XML tree if π 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 π_1 and π_2, often 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 π_1 and π_2, respectively, match the path from the root to x and the path from x' to y, where x' is the node preceded by x on the path from the root to y.

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

[1] 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".

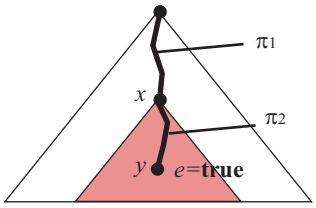

Figure 1. An illustration of occurrence of XPattern $\pi_1[\pi_2 : e]$.

An *XPattern* is a pair of a path pattern $\pi_1[\pi_2]$ and a Boolean expression e over keywords $w_1, \ldots, w_m$ in Σ^+, 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 $\pi_1[\pi_2 : \mathbf{true}]$.

Now, we give a formal definition of the problem.

Definition 1 (XML Document Retrieval)
Given. *An XML document* T.
Query. *A set of XPatterns* $P_1, \ldots, P_\ell$.
Answer. *Pairs of a node* x *of the XML tree for* T *and a bit vector of size* ℓ *representing the patterns* P_i *that occur at* x.

3 Preprocessing

In the preprocessing part, we read once the given XML file from the left to the right to build the *path trie*, a trie representing the set of label strings of the paths from the root. At the same time we also modify the XML file as follows. We replace every occurrence of the start tags with a special byte code followed by an integer indicating the corresponding node in the path trie, and every occurrence of the end tags with another special byte code. We refer to the resulting binary file as the *binary XML file*. It should be noted that the start tag occurrences are replaced with the corresponding path-trie node IDs, not with the tag IDs. Fig. 2 illustrates the preprocessing part.

Searching a tag name in the list of tag names already processed (and assigning its ID if it is new) can be done in $O(1)$ time using standard hashing technique. The preprocessing part thus takes $O(n \log |\mathcal{N}| + |T|)$ time, where n is the number of occurrences of start tags in input XML document T, $\mathcal{N}$ is the set of tag names occurring in T, and $|T|$ is the length of T.

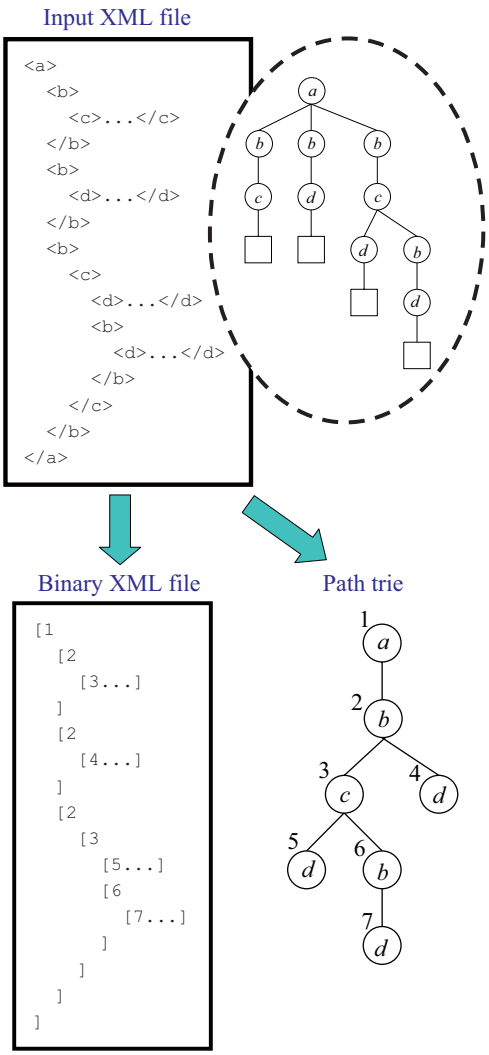

Figure 2. Preprocessing part of the algorithm is illustrated. An XML file with its tree representation is displayed on the upper, where the squares represent the text nodes. The binary XML file and the path trie created from the XML file are shown on the lower-left and on the lower-right, respectively. The numbers adjacent to the nodes of the path trie mean 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.

4 Path pattern matching stage

We want to find all loci (x, y) in the input XML tree at which given path patterns $\pi_1[\pi_2]$ occur. Let v be the label

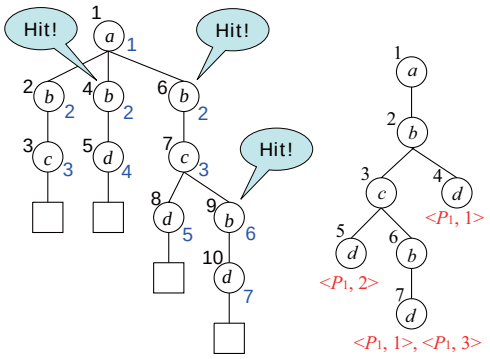

Figure 3. On the left, the XML tree of Fig. 2 is displayed again. The path pattern P_1 = a//b[//d] occurs at loci $(x, y) = (4, 5), (6, 8), (6, 10), (9, 10)$. The loci can be obtained from the information added to the path trie shown on the right. For instance, the output $\langle P_1, 2\rangle$ of the path-trie node 5 implies that the occurrence of P_1 at locus (x, y) such that $y = 8$ is associated with the path-trie node 5 and $x = 6$ is the 2nd ancestor of y.

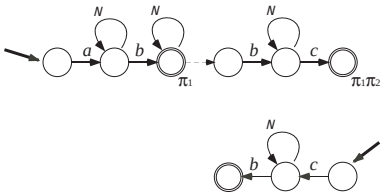

Figure 4. The NFAs for the pattern $\pi_1[\pi_2]$ are displayed, where π_1 = a//b// and π_2 = b//c. The forward NFA is displayed on the upper, composed of the NFA M_f^1 for π_1 (on the upper-left) and the NFA M_f^2 for π_2 (on the upper-right). The backward NFA M_b is displayed on the lower, which accepts the set of reversals of the strings the pattern π_2 matches.

string of the path from the root to y, and let d be the length of the path from x to y. Our problem is then reduced to the problem of finding all possible such pairs (v, d). Since the candidates for v are represented as the nodes of the path trie, the problem can be stated over the path trie.

Definition 2 (Occurrence Locus Listing in Path Trie)
Given. *A trie T representing a finite set of strings over $\mathcal{N}$.*
Query. *Path patterns $P_1, \ldots, P_\ell$.*
Compute. *All pairs (v, d) of a node v of T and an integer $d \geq 0$ such that P_i occurs at locus (u, v) of T such that u is the d-th ancestor of v, for each $i = 1, \ldots, \ell$.*

Fig. 3 demonstrates the path trie of Fig. 2 with information about the pairs (v, d) for the path pattern P_1 = a//b[//d], where the path-trie node v has an output $\langle P_1, d\rangle$ for each (v, d) of the pairs.

Consider the following problem: *Given a string t over $\mathcal{N}$ and a path pattern $P = \pi_1[\pi_2]$, to find all the pairs (i, j) such that $1 \leq i \leq j \leq |t|$, and π_1 and π_2 match $t[1..i]$ and $t[i + 1..j]$, respectively.* Since a quadratic number of pairs (i, j) could be the answer, at least a quadratic time is required. One solution would be to build two NFAs M_f^1 and M_f^2 for π_1 and π_2, respectively. The size of M_f^i is $c(\pi_i) + 1$, where $c(\pi_i)$ denotes the number of tag names and *'s appearing in π_i. We make M_f^1 run on t to find the positions i at which occurrences of π_1 end. Whenever the i is found, we make M_f^2 run to find the positions j such that π_2 matches $t[i + 1..j]$. By using the bit-parallel techniques [26], the method requires $O\big(|t| \cdot \lceil (c(\pi_1) + 1)/w\rceil + |t|^2 \cdot \lceil (c(\pi_2) + 1)/w\rceil\big)$ time using $O\big(|\mathcal{N}| \cdot (\lceil (c(\pi_1) + 1)/w\rceil + \lceil (c(\pi_2) + 1)/w\rceil)\big)$ space after $O\big((|\pi_1| + |\pi_2|) \cdot |\mathcal{N}|\big)$ time preprocessing, where w is the computer word length. The algorithm runs $O(|t|^2)$ time when the NFA sizes fit the computer word length w. In practice the condition is often satisfied since w is 32 or 64 in current architectures.

Our practically faster solution is as follows. We combine the NFAs M_f^1 and M_f^2 into one NFA called the *forward NFA*, by adding an ε-transition from the unique final state of M_f^1 to the unique initial state of M_f^2. The size of the forward NFA is $c(\pi_1) + c(\pi_2) + 1$. We note that the forward NFA has two final states to detect distinctively occurrences of π_1 and $\pi_1\pi_2$. We also build the *backward NFA* M_b, the NFA accepting the set of reversal of the strings π_2 matches. The size of M_b is $c(\pi_2)+1$. The construction of the forward and the backward NFAs requres $O\big((|\pi_1|+|\pi_2|)\cdot|\mathcal{N}|\big)$ time. Make the forward NFA run on t, with storing into a one-dimensional array of size $|t| + 1$ the sequence of bit vectors representing the active states. Whenever the pattern $\pi_1\pi_2$ is found at position j, we make the backward NFA run on t in the reverse direction in order to find the start positions $i + 1$ of occurrences π_2 ending at j. If the backward NFA finds a match of π_2 and the corresponding bit vector stored in the array contain the final state of M_f^1, one of the start positions is detected. We note that at least one start position is necessarily found. This method also takes $O(|t|^2)$ time, but runs faster than the above solution in practice.

It should be emphasized that the pattern matching with the backward NFA cannot be omitted. In general, for any strings u, v, π_2 does not necessarily match v even if $\pi_1\pi_2$ matches uv and π_1 matches u. Consider the case illustrated in Fig. 5.

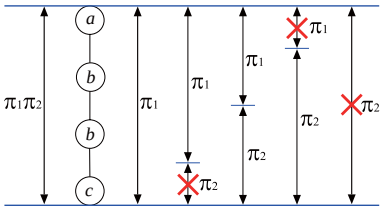

Figure 5. Let π_1 = a//b//, π_2 = b//c, and t = a/b/b/c. $\pi_1\pi_2$ = a//b//b//c matches t and π_1 matches the prefixes a/b/b/c**,** a/b/b**, and** a/b **of t. Of the corresponding suffixes,** b/c **is matched by π_2 but** c **and ε are not matched by π_2.**

To solve the problem of Definition 2, we perform a depth-first traversal of the path trie for each of the path patterns $P_1, \ldots, P_\ell$. When going down in the path trie, we simulate the nondeterministic state transitions of the forward NFA in parallel for each node label, with storing a series of bit-vectors each representing the active states. If the pattern $\pi_1\pi_2$ is found, then make the backward NFA run along the path string of the current node back to the root in order to detect the occurrence loci.

Processing time (excluding NFA construction) per one path pattern is proportional to the summation of $|path(v)|$ over all the nodes v in path trie T, where $path(v)$ denotes the path from the root to v. Then the path pattern matching stage totally requires $O(|\mathcal{N}| \cdot \sum_{i=1}^{\ell} |P_i| + \ell \cdot h \cdot n)$ time, where h and n denote the height of T and the number of nodes in T, respectively, when the pattern lengths fit the computer word length.

5 Text-node searching stage

5.1 Index- versus PMM-based methods

The contents of text nodes can be taken as documents. Then the problem to be considered here is stated as follows.

Definition 3 (Boolean-Model Document Retrieval)
Given. *Documents $d_1, \ldots, d_k$ in Σ^+.*
Query. *Boolean expressions $e_1, \ldots, e_n$ with variables $w_1, \ldots, w_m$ in Σ^+.*
Answer. *The lists $S_1, \ldots, S_n$, where S_i consists of the integers j such that d_j satisfies e_i.*

Since we are allowed to preprocess the contents of text nodes, it is natural to consider using text indices. The most popular indices are the *inverted lists*. For each word in vocabulary (the set of words in text data), the list of all its occurrence positions is stored. More sophisticated index structures are *suffix trees* and *suffix arrays* (see e.g. [6]), which locate all occurrences of *any* substrings (not necessarily words) of text data. They are superior to the inverted lists for searching phrases or more complex queries, or in applications where the concept of word is of no use, such as computational biology.

One solution would be to (1) build a text index for the documents, (2) find the set of documents in which keyword w_i occurs, for each $i = 1, \ldots, m$, and then (3) perform the set operations required for the Boolean expressions $e_1, \ldots, e_n$.

Let us consider the following sub-problem: *Given documents $d_1, \ldots, d_k$ in Σ^+ and a keyword w in Σ^+, to answer the list L of integers j such that w occurs in d_j.* Muthukrishnan in [25] presented an optimal algorithm which runs in $O(|w| + |L|)$ time, after $O(N)$ time and space preprocessing, where $N = |d_1| + \cdots + |d_k|$. This algorithm uses the generalized suffix tree (GST) [17] for the set of documents and additional two integer arrays of size $N + k$. The space requirement of GST is at least $10N$ bytes and the drawback of the algorithm is thus space inefficiency. Replacing the GST with simple binary searches over the suffix array built for the concatenation of documents d_i each followed by a special character $\$_i$ decreases the space requirements without loss of practical search speed (although $O(|w| \log(N + k) + |L|)$ time is needed theoretically), but the suffix array itself requires $4(N + k)$ bytes in addition to $N + k$ bytes for the text data. The *Self-indices* are compressed index structures including text data. Instances of the self-indices are the compressed suffix arrays [14, 30] and the FM-index [8, 9], which are smaller than the text size. The search speed is, however, much slower than suffix arrays.

The index-based methods discussed here proceed pattern-by-pattern, and therefore a considerable amount of extra working space is needed for set operations. The approach taken in this paper is to build from the keywords $w_1, \ldots, w_m$ a PMM and to make it run on the binary XML file. One advantage is space efficiency. Another important advantage is document-by-document processing based on simultaneous search of multiple keywords $w_1, \ldots, w_m$. Thanks to the advantage, no set operations are needed and Boolean operations for every document are enough.

The main drawback of our approach is search time inefficiency. But the search time per one query is considered to be relatively small, since we process a large number of queries at a time in our setting. In fact our preliminary experiments showed that the PMM-based approach is in some cases only 3–5 times slower than the (uncompresed) suffix array based approach in solving the Boolean-Model Document Retrieval problem with 500-1,000 keywords.

5.2 Algorithm

Let $W = \{w_1, \ldots, w_m\}$ be the set of keywords occurring in the Boolean expressions e of the XPatterns $P_1, \ldots, P_\ell$. 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 the current node of depth d. 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**.

  If there is an integer q in $\{1, \ldots, \ell\}$ such that $Q[depth][q]$ is **true**, then outputs the node $(start, \mathit{offset})$ and the bit vector representing $Q[depth][1..\ell]$. 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 $Occ[depth][1..m]$ appropriately.

6 Extensions

6.1 Permitting references to other queries

Let the input queries in the XPatterns format be:

$$\begin{cases} P_1 &= \pi_1^1[\pi_2^1 : e^1] \\ &\vdots \\ P_\ell &= \pi_1^\ell[\pi_2^\ell : e^\ell] \end{cases}$$

The expressions e^i are Boolean expressions over string patterns $w_1, \ldots, w_m$. Here we extend e^i to be Boolean expressions over variables $w_1, \ldots, w_m$ and $P_1, \ldots, P_{i-1}$. 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 the two dimensional array $Q[depth][1..i-1]$ for the current value of $depth$ in the algorithm stated in Section 5.2. This extension enables us to introduce the logical connectives and the nested predicates as follows.

Logical connectives. Consider the XPath $\pi[\pi_1 \wedge \pi_2]$, for instance. Let $P_1 = \pi[\pi_1 : \mathbf{true}]$ and $P_2 = \pi[\pi_2 : \mathbf{true}]$, and let $P_3 = \pi[\varepsilon : P_1 \wedge P_2]$. The XPath $\pi[\pi_1 \wedge \pi_2]$ occurs if and only if $Q[depth][3] = \mathbf{true}$ after the truth-value evaluation in the modified algorithm mentioned above.

Nested predicates. Consider the XPath $\pi_1[\pi_2[\pi_3 \wedge \pi_4]]$. 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 $\pi_1[\pi_2[\pi_3 \wedge \pi_4]]$ occurs if and only if the value $Q[depth][3]$ 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 $Q[depth][6]$ can be **true** even when there is no occurrence of the XPath $\pi_1[\pi_2[\pi_3 \wedge \pi_4]]$.

6.2 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. Inequations 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 $\pi_1[count(\pi_2) > 1]$. We replace the statement

> If e is **true**, then set $Q[depth - d][q]$ to **true**

of the algorithm stated in Section 5.2 with

> Increment $Q[depth - d][q]$ by e.

Let $P_1 = \pi_1[\pi_2 : 1]$ and $P_2 = \pi_1[\varepsilon : (P_1 > 1)]$. Then the XPath $\pi_1[count(\pi_2) > 1]$ occurs if and only if $Q[depth][2] > 0$.

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

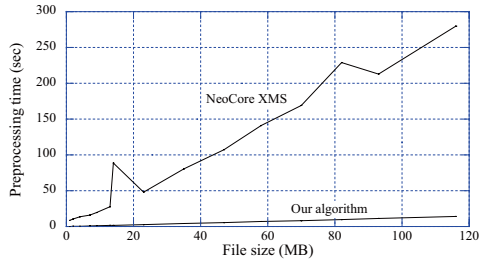

Figure 6. Preprocessing times of our algorithm and NeoCore XMS are displayed.

7 Experimental results

We ran a series of experiments to evaluate the performances of our algorithm, compared with XMLTK, Tamino and NeoCore XMS. We chose the Windows 2003 server edition of NeoCore XMS as it is said to be most finely tuned. We tested its performances on a PC with a 3.6GHz Xeon processor and 3.5GB RAM running Windows 2003 server. For comparisons, the performances of our algorithm were tested on the same PC running RedHat Enterprise Linux AS v3.0 for EM64T. On the other hand, comparisons of our algorithm with XMLTK and with Tamino were carried out on a PC with a 2.4GHz Intel Pentium 4 processor and 2.0GB RAM running RedHat Linux Advanced Server 2.1. The input XML document files used were randomly generated by xmlgen [31].

7.1 Processing time

First, we measured the preprocessing times (elapsed time) of our algorithm. The results are shown in Fig. 6, where the preprocessing times for NeoCore XMS are also shown for a comparison. The preprocessing of our algorithm thus requires small amount of time.

Second, we measured the path-pattern matching times against the simple path pattern and path pattern classes. The simple path patterns used here were randomly generated by a pathgenerator from "http://yfilter.cs.berkeley.edu/code_release.htm", where the maximum depth of the simple path patterns was set to 10, and the probabilities $prob(//)$ and $prob(*)$ of occurring "//" and "*" were equally set to 0.01, 0.05, and 0.10. The path patterns were built from the simple path patterns. The path trie was built from an XML file of size 111MB and had 549 nodes. Table 2 shows the path-pattern matching times averaged over 100,000 patterns generated for each combination of a pattern class and a probability value. The times consumed in the path pattern matching stage are thus quite small, and stable independently of the probability value.

Table 2. Path-pattern matching times (i.e. times for building and running NFAs) against simple path pattern and path pattern classes are shown, where the probability is varied $prob(//) = prob(*) = 0.01, 0.05$, and 0.10.

	CPU time (μsec)		
	build	run	total
simple path pattern (0.01)	2.62	17.49	20.11
simple path pattern (0.05)	2.59	17.53	20.12
simple path pattern (0.10)	2.63	17.60	20.23
path pattern (0.01)	2.64	20.99	23.63
path pattern (0.05)	2.61	21.22	23.83
path pattern (0.10)	2.68	21.40	24.08

Third, we measured the search times (times for processing binary XML file, excluding the preprocessing times and the path-pattern matching times) of our algorithm against patterns in the following four classes.

(1) Simple path patterns with $prob(//) = prob(*) = 0.05$.

(2) Path patterns built from (1).

(3) XPatterns, which were built by associating each of the path patterns of (2) with the logical OR of five words randomly selected from the highly frequent 100 words occurring in the text nodes.

(4) Path patterns with aggregation, which were built by associating each of the path patterns of (2) with the $count$ function.

For each the classes, we created four pattern sets of size 1, 10, 100, and 1000. We generated a series of XML files of size up to 116MB, and the sizes of corresponding binary XML files are about 90% of their original ones.

Fig. 7 compares the search times against the pattern set of size 10 for each of the classes (1)–(4). The differences in search times due to the pattern classes are quite small except for (3). (Recall that each pattern in (3) has five keywords to be searched for.) Table 3 compares the search times against four pattern sets of size 1,10,100, and 1000 for each of the four classes, where the input XML file is fixed to the one of size 116MB. It is observed from the table that, for each pattern class, even if the number of queries increases 10 times, the search time does not become so: It is even smaller than 10 times the original search time. The difference between the search times of (2) and (3) depends upon the number of occurrences of keywords associated with queries. We note that the number of distinct keywords does not increase accordingly as the queries increase. For example, the pattern sets of size 100 and 1000 in (3), respectively, carry 500 keywords and 5000 keywords but there are at most 100 distinct

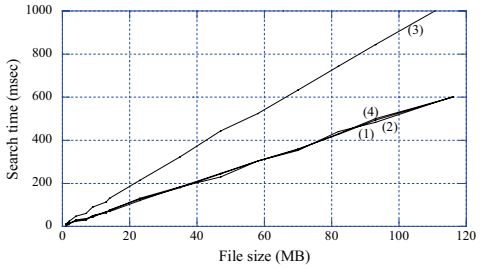

Figure 7. Search times of our algorithm for all pattern classes (1)–(4) are displayed against XML file of varying size. The pattern sets used are of size 10.

Table 3. Search times of our algorithm for all classes (1)–(4) are displayed for varying number of patterns. The XML file used is of size 116MB.

	elapsed time (sec)			
	(1)	(2)	(3)	(4)
#queries = 1	0.560	0.553	1.000	0.663
#queries = 10	0.603	0.600	1.047	0.600
#queries = 100	1.113	1.083	1.550	1.360
#queries = 1000	5.913	5.823	6.203	8.203

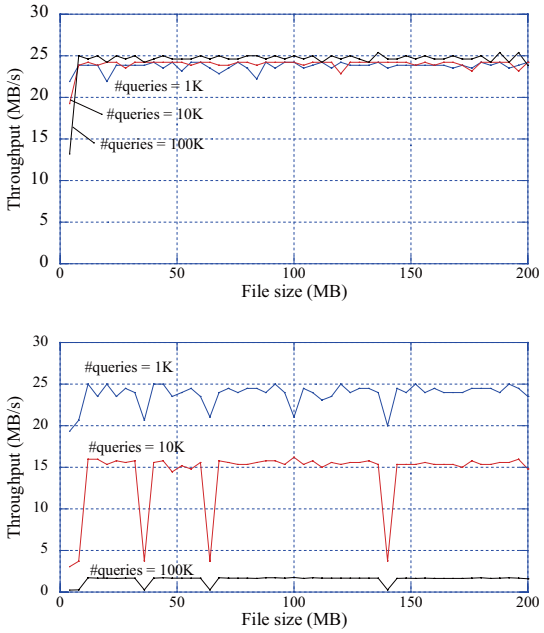

Figure 8. Throughputs of our algorithm and XMLTK are displayed on the upper and on the lower, respectively. For a fair comparison, a modified version of our algorithm was used which emits only the maximal regions matched by some pattern as XMLTK does.

keywords. For this reason, the search time for the pattern set of size 1000 in (3) is relatively small compared to that for the pattern set of size 1000 in (4).

7.2 Throughput comparison

We compared the throughputs of our algorithm and XMLTK against a huge number of queries. For a fair comparison, we used a modified version of our algorithm that emits only the maximal regions matched by some pattern as XMLTK does. We note that XMLTK basically does not preprocess input XML file and performs the path-pattern matching during the scan of the input XML file, whereas our algorithm does the path-pattern matching on the path trie before scanning the binary XML file. In what follows we see the effect of preprocessing of our algorithm.

Since XMLTK does not support path patterns nor more complex patterns, the experiment was performed against simple path patterns. We used the sets of simple path patterns with $prob(//) = prob(*) = 0.05$ mentioned above.

The throughputs of our algorithm and XMLTK are compared in Fig. 8. We note that the throughputs of our algorithm are translated values in terms of the original XML file size. Since XMLTK bases on "the LazyDFA technique" [12], a warm-up phase is seen at the beginning of the processing. The throughput of XMLTK became stable after the warm-up phase. It is observed that the throughputs of XMLTK get worse according to the increase of the number of queries, whereas those of our algorithm are stable, not depending on the number of queries.

7.3 Versus commercial products

We extended the implementation of our algorithm to cope with XQuery. Then we compared the performances of our algorithm, Tamino, and NeoCore XMS in processing queries provided by XMark benchmark [31], a tool kit for evaluating the retrieval performance of XML stores and query processors. This benchmark includes a suite of 20 benchmark queries written in the XQuery language. The current implementation of our algorithm does not support a full set of XQuery, for example, the join and sort operations have not been implemented yet. It can process only the nine queries: Q1, Q6, Q7, Q13–Q17, and Q20. See "http://www.ins.cwi.nl/projects/xmark/Assets/xmlquery.txt".

The query processing times of our algorithm and Tamino against the nine queries are displayed in Fig. 9. We see that the processing times of our algorithm are linear with

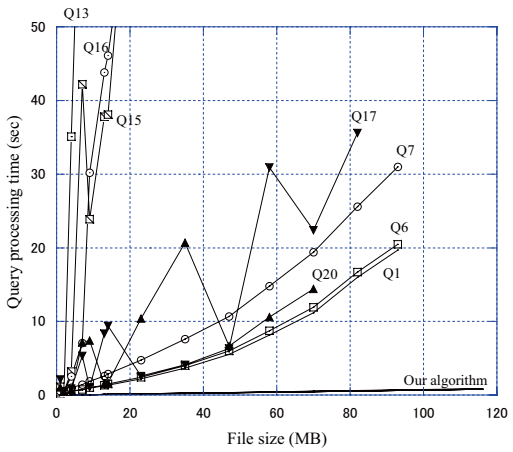

Figure 9. Query processing times (elapsed time) of Tamino and our algorithm against XMark benchmark queries are displayed. The times of our algorithm are piled up into a line near the X-axis, whereas those of Tamino are scattered depending both on the queries and on the file size.

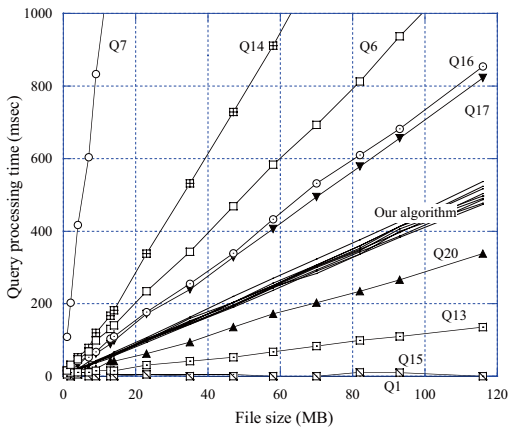

Figure 10. Query processing times (elapsed time) of NeoCore XMS and our algorithm against XMark benchmark queries are displayed. The performances of NeoCore XMS vary depending upon the queries, whereas those of our algorithm are stable. For five of the nine queries, our algorithm outperforms NeoCore XMS.

respect to the length of input XML data. They are stable not depending on queries and piled up into a single line near the X-axis. On the contrary, those of Tamino are scattered depending on queries and on file size.

The query processing times of NeoCore XMS and our algorithm against the nine queries are displayed in Fig. 10. Our algorithm is thus competitive to NeoCore XMS.

Next, we compared the query processing times of our algorithm with those of NeoCore XMS against multiple queries. Since NeoCore XMS cannot process multiple queries at once, we measured the sum of times for repeated searches by NeoCore XMS. The results are shown in Table 4. Concerning this comparison, our algorithm drastically beats out NeoCore XMS.

Table 4. Query processing times of our algorithm and NeoCore XMS against sets with size 100 of randomly generated simple path patterns and path patterns. The values are the processing times per one query. The XML file used is of size 116MB.

	elapsed time (msec)	
	our algo.	NeoCore XMS
simple path pattern (0.01)	8.18	205.11
simple path pattern (0.05)	8.31	924.32
simple path pattern (0.10)	8.61	1668.89
path pattern (0.01)	11.73	5064.00
path pattern (0.05)	15.88	6164.60
path pattern (0.10)	14.57	6546.19

8 Conclusion

We have presented a light-weight XML query processor, which is based on the path-pattern matching against the path trie, and on the text-node searching using Aho-Corasick's pattern matching technique. A series of experiments proved that our algorithm outperforms or is competitive with existing XML query processors such as XMLTK, Tamino, and NeoCore XMS.

In the experiments we dealt with XML file of size up to about 100MB. To cope with a huge amount of XML data, we adopt a simple distributed parallel computation technique: We use one director server and n search servers. The director divides the XML data into n pieces, and distributes them to the search servers. The search servers preprocess their distributed XML data to create path tries and binary XML files. Given queries the search servers respectively process them and then send the answers to the director. A similar strategy using 128 CPUs has been successfully adopted to a bio-database system developed by Fujitsu Ltd. and Japan's National Institute of Genetics. We thus concentrate on improving time and space efficiencies of query processing by a respective search server against relatively small XML data since n can be large.

Two techniques for accelerating our algorithm exist. One is the stream index (SIX for short) [12], which is a list of pairs of positions at which a start tag and the corresponding end tag occur, respectively. The size of SIX is not large relatively to the size of the XML document. The SIX is expected to speed up our algorithm by skipping several subtrees in binary XML files. The other technique is speeding-up by text compression [32, 24] developed by our research group, which reduces the running time of Aho-Corasick's pattern matching machine at nearly the same rate as the compression ratio.

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. In *VLDB'00*, pages 53–64, 2000.

[3] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. TeXQuery: A full-text search extension to XQuery. In *WWW'04*, pages 583–594, 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] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2002.

[7] Y. Diao, P. Fisher, M. J. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *ICDE'02*, pages 341–342, 2002.

[8] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS'00*, pages 390–398, 2000.

[9] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *SODA'01*, pages 269–278, 2001.

[10] 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.

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

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

[13] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *ICDT'03*, pages 173–189, 2003.

[14] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *STOC'00*, pages 397–406, 2000.

[15] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *SIGMOD'03*, pages 431–442, 2003.

[16] P. J. Harding, Q. Li, and B. Moon. XISS/R: XML indexing and storage system using RDBMS. In *VLDB'03*, pages 1073–1076, 2003.

[17] L. C. K. Hui. Color set size problem with application to string matching. In *Combinatorial Pattern Matching*, volume 644 of *LNCS*, pages 230–243, 1992.

[18] Z. G. Ives, A. Y. Halevy, and D. S. Weld. An XML query engine for network-bound data. *The VLDB Journal*, 11(4):380–402, 2002.

[19] V. Josifovski, M. Fontoura, and A. Barta. Querying XML streams. *The VLDB Journal*, 14:197–210, 2005.

[20] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An optimizing XQuery processor for streaming XML data. In *VLDB'04*, pages 1309–1312, 2004.

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

[22] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB'01*, pages 361–370, 2001.

[23] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In *VLDB'02*, 2002.

[24] M. Takeda, et al. Speeding up string pattern matching by text compression: The dawn of a new era. *Trans. Information Processing Society of Japan*, 42(3):370–384, 2001. Special issue for IPSJ 40th anniversary award papers.

[25] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA'02*, pages 657–666, 2002.

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

[27] D. Olteanu, T. Furche, and F. Bry. An efficient single-pass query evaluator for XML data streaams. In *SAC'04*, pages 627–631, 2004.

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

[29] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *SIGMOD'03*, pages 431–442, 2003.

[30] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *ISAAC'00*, LNCS 1969, pages 410–421, 2000.

[31] 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.

[32] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. In *CIAC'00*, LNCS 1767, pages 306–315, 2000.

[33] T. Shimizu and M. Yoshikawa. Full-text and structural XML indexing on B^+ tree. In *DEXA'05*, pages 451–460, 2005.

[34] W3C. XQuery 1.0 and XPath 2.0 full-text use cases. http://www.w3.org/TR/xmlquery-full-text-use-cases.

[35] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM Trans. on Internet Technology*, 1(1):110–141, 2001.