Full-Text and Structural XML Indexing on B^+-Tree

Toshiyuki Shimizu[1] and Masatoshi Yoshikawa[2]

[1] Graduate School of Information Science, Nagoya University
shimizu@dl.itc.nagoya-u.ac.jp
[2] Information Technology Center, Nagoya University
yosikawa@itc.nagoya-u.ac.jp

Abstract. XML query processing is one of the most active areas of database research. Although the main focus of past research has been the processing of structural XML queries, there are growing demands for a full-text search for XML documents. In this paper, we propose XICS (XML Indices for Content and Structural search), novel indices built on a B^+-tree, for the fast processing of queries that involve structural and full-text searches of XML documents. To represent the structural information of XML trees, each node in the XML tree is labeled with an identifier. The identifier contains an integer number representing the path information from the root node. XICS consist of two types of indices, the COB-tree (COntent B^+-tree) and the STB-tree (STructure B^+-tree). The search keys of the COB-tree are a pair of text fragments in the XML document and the identifiers of the leaf nodes that contain the text, whereas the search keys of the STB-tree are the node identifiers. By using a node identifier in the search keys, we can retrieve only the entries that match the path information in the query. Our experimental results show the efficiency of XICS in query processing.

1 Introduction

The efficient processing of XPath [1] or XQuery [2] queries is an important research topic. Since the logical structure of XML is a tree, establishing a relationship between nodes such as parent-child or ancestor-descendant is essential for processing the structural part of queries. For this purpose, many proposals have been made such as structural joins, indexing, and node labeling [3–8].

In the last few years, the XML full-text search has emerged as an important new research topic [10, 11]. However, efficient processing of XML queries that contain both full-text and structural conditions has not been studied well. In this paper, we propose XICS (XML Indices for Content and Structural search), which aims at high-speed processing of both full-text and structural queries in XML documents. An important design principle of our indices is the use of a B^+-tree. Because the B^+-tree is widely used in many database systems, building indices on a B^+-tree rather than creating a new data structure from scratch is an important design choice from a practical point of view. Several indices for XML documents using a B$^+$-tree have already been proposed. For example, XISS [3] is a node index approach on a B$^+$-tree. XISS is flexible in that the basic unit to be indexed is a node; however, to process a query, the query needs to be decomposed to a node unit, and then intermediate results need to be joined. The XR-Tree [4] is another tree-structured index for XML documents. In an XR-Tree, nodes in XML documents are labeled and stored in an extended B$^+$-tree index.

These indices efficiently preserve the ancestor-descendant or parent-child relationship between nodes; however, they do not take full-text searches into consideration. Recently, an indexing approach compatible with a full-text search for XML documents that integrates structure indexes and inverted lists was proposed in [14], which uses element names or keywords as a search key of indices. In our approach, to accelerate both the structures and full-text searches of XML documents, we constructed a B$^+$-tree in which the search keys are a pair of text fragment t and the node identifier of the leaf node which contains t. The node identifiers consist of two parts: a path identifier that indicates the path from the root node and the Dewey-order among sibling nodes sharing the same path identifiers. Search keys are first sorted by text fragments; hence, the index entries that contain the same text are clustered in the index. In such a cluster, entries representing nodes that have the same structure are clustered together. We call this type of index a COB-tree (COntent B$^+$-tree). We can answer XPath queries involving both structure and contents specifications such as "//title[contains(.,'XML')]" , which needs a join operation in the case of [14], by traversing the COB-tree only once.

A COB-tree is not suitable for processing structural queries such as "//title", because entries in a COB-tree are first sorted by text. Therefore, we constructed another type of B$^+$-tree called an STB-tree (STructure B$^+$-tree). In an STB-tree, the above-mentioned node identifiers are used as search keys. An important observation about an STB-tree is that entries are not clustered by element name. This is because path identifiers do not, in general, cluster nodes having the same element names. To manage this problem, we have developed a *search-key mapping* technique in which index entries are sorted by the lexicographical order of the reverse path (the element path from the node upward to the root node) and not by the path identifier itself. Reverse paths are effective in processing XPath queries that include "//". When searching, the index is traversed by mapping the reverse path to the path identifiers. By employing the *search-key mapping* technique, entries relevant to the nodes that have the same tag name are clustered in the index; hence, a query such as "//title" can be processed efficiently.

XICS consists of a COB-tree and an STB-tree. In general, when processing an XPath query having the *contains()* function, we can filter nodes using the text conditions or the structural conditions in the query. The use of the XICS can accelerate both types of filtering. We have implemented a COB-tree and an STB-tree using GiST [12]. The experimental results show the effectiveness of XICS.

2 PSP: A Node Labeling Scheme

In this section, we explain our node labeling scheme using the XML document shown in Figure 1. Figure 1 is a tree representation of an XML document. The ovals, triangles, rhombi, and strings in the rectangles represent element nodes, attribute nodes, text nodes, and text values, respectively.

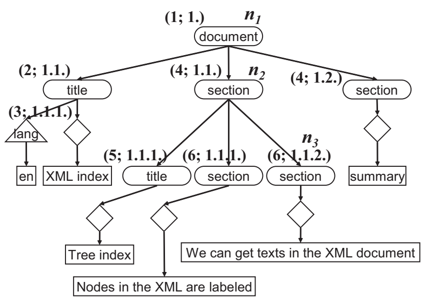

Fig. 1. Tree representation of sample XML document.

Table 1. Correspondence of path and path identifier.

Path	Path Identifier
/document	1
/document/title	2
/document/title/@lang	3
/document/section	4
/document/section/title	5
/document/section/section	6

Node labeling schemes play an important role in XML query processing, and thus many studies [3, 8] of them have been made. A widely used node identifier is a pair of *preorder* and *postorder*, which can uniquely reconstruct the topology of an XML tree. However, such node identifiers do not convey element names or path information. It is important to obtain such information easily from a node identifier in order to quickly obtain the nodes corresponding to path expressions in the query. Therefore, we have designed a node labeling scheme in which node labels contain a *path identifier.* A path identifier identifies the path from the root node to a node. Table 1 shows an instance of path identifiers assigned to the paths in the XML document in Figure 1. In general, we cannot uniquely distinguish the nodes in an XML document only by path identifier. For example, the two nodes corresponding to "/document/section" in Figure 1 have the same path identifier. Therefore, we have introduced the *Sibling Dewey Order* to preserve order information among sibling nodes. The Sibling Dewey Order of the root node is 1. The Sibling Dewey Order of a non-root node n is a concatenation of the Sibling Dewey Order of the parent of n and the sibling order of n among siblings assigned the same path identifiers.

We call a pair of a path identifier and a Sibling Dewey Order a *PSP* (*Path Sibling Pair*). Nodes are uniquely identified by a PSP. For example, with reference to the path identifiers in Table 1, each node in Figure 1 is labeled by a PSP $(x; y)$, where x denotes a path identifier, and y denotes a Sibling Dewey Order. For example, the path from the root node to node n_3 is "/document/section/section", so the path identifier is 6 Furthermore, node n_3 is the second sibling among the sibling nodes with the same element name. Therefore, the Sibling Dewey Order of the node n_3 becomes 1.1.2. because the Sibling Dewey Order of the parent node n_2 is 1.1. and the sibling order of n_3 is 2.

The nature of PSP makes it possible to identify the nodes at the instance level and to easily verify the parent-child or ancestor-descendant relationship between nodes. We can quickly obtain the path relationship between two nodes by referring to the inclusive relationship of the paths corresponding to the path identifiers. Note that the table storing the correspondence between paths and path identifiers is small enough to be kept in the main memory. Once a path relationship among nodes is verified, the instance level parent-child or ancestor-descendant relationship is verified by the subsequence matching of the Sibling Dewey Order. The PSP compactly conveys useful information for processing queries efficiently.

3 Index Construction

We propose two kinds of indices on a B$^+$-tree: A *COB-tree* (*COntent B$^+$-tree*) and an *STB-tree* (*STructure B$^+$-tree*). The search keys in a COB-tree are the pairs of the text fragment and the PSP of the node in which the text appears, which is used for processing a query that involves both the text and structure of the XML document. The search keys in an STB-tree are the PSP of all element nodes and attribute nodes, which is used for processing queries that only involve the structural information of the XML documents.

3.1 Text in COB-tree

To answer full-text searches and keep phrase information, we use the suffix texts of a text in an XML document as the text in the search key of a COB-tree. For example, the suffix of "Nodes in the XML are labeled" are as follows:

```
Nodes in the XML are labeled    XML are labeled
in the XML are labeled          are labeled
the XML are labeled             labeled
```

Pairs of each of the suffix texts and the PSP of the node that contains the suffix text make up the search keys of a COB-tree. However, keeping all phrase information in the index increases the index size. Therefore, we decided to keep only the words that were needed to distinguish the phrase from other phrases. For example, when we refer to Figure 1, for the suffix "the XML are labeled" of the text "Nodes in the XML are labeled", we keep only the first three words "the XML are" as these are enough to be distinguished from the suffix "the XML document" of the text "We can get texts in the XML document". Even if the search phrase is longer than a matching text in the index, we can narrow down the candidate nodes to only one.

3.2 Search-Key Mapping

We must pay attention to the order of the keys, since the cost can be minimized by retrieving adjacent leaf pages of a B^+-tree. If we sort the path identifiers in Table 1 simply by their value, there is almost no meaning to the order. In general, it is difficult to meaningfully assign a unique value to the path. For example, when we process the query "//title", the path identifiers corresponding to the path are expected to be clustered in the B^+-tree. However, if we sort the path identifiers simply by value, the path identifiers corresponding to the path "//title" , which are 2 and 5 in the running example, are generally dispersed in one or more leaf pages in the B^+-tree.

To overcome this problem, we propose *search-key mapping*, in which the key order is determined not by the key itself but by the value transformed when using information about the key (mapping information). Table 1 is used to retrieve the correspondence between path and path identifier in the following example. When we process a query that contains "//", such as "//title", the path identifiers 2 and 5 are expected to be clustered in the index. When such a case is considered, it is appropriate to order the path identifiers based on the reverse path of the corresponding path in the B^+-tree. That is, in this example, when we use "\" as a delimiter of the reverse path steps, we prepare mapping information such as "document\" for 1, "title\document\" for 2, "@lang\title\document\" for 3, and so on. Then, the order of the path identifiers is determined based on the lexicographical order of the corresponding reverse path. In this example, the order of the path identifier using the mapping information is $3 < 1 < 4 < 6 < 2 < 5$. Generally, the mapping information is small enough compared with XML documents, and we can retain it in the main memory. Therefore, ordering with mapping information can be done very fast.

Entries corresponding to the nodes with the same tag name are clustered in one location in the index, so we can process the XPath query containing "//" efficiently. Furthermore, since the Sibling Dewey Order is ordered by comparing the value of sibling numbers from the root node, the ordering of the key in an STB-tree is determined first by the path identifiers, using mapping information, and then by the Sibling Dewey Order when the path identifiers are equal. The ordering of the key in a COB-tree is determined first by the text and then by the same method as in an STB-tree when the texts are equal. The above approaches permit the clustering of index entries corresponding to the nodes with the same path in addition to the nodes that contain the same suffix text.

Figures 2 and 3 show an STB-tree and a COB-tree respectively, constructed for the XML document in Figure 1. In these figures, the delimiter of the text and the PSP is ",", and the delimiter of the path identifier and the Sibling Dewey Order is ";". We set $L = 3$, which is a threshold for the maximum phrase length of the text in a COB-tree to reduce the index size from a practical point of view. For simplicity, the indices in Figures 2 and 3 are constructed so that one page can contain a maximum of four entries; however, in actual indices, one page can contain over 100 entries and in this way the height of the B^+-tree is kept low.

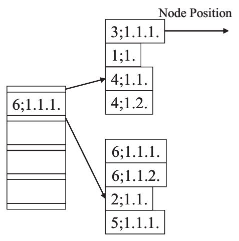

Fig. 2. STB-tree.

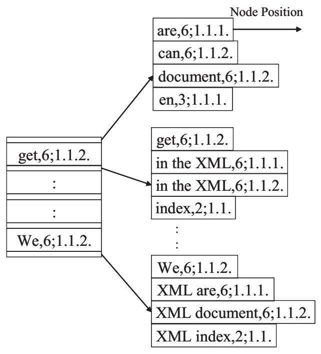

Fig. 3. Normal COB-tree.

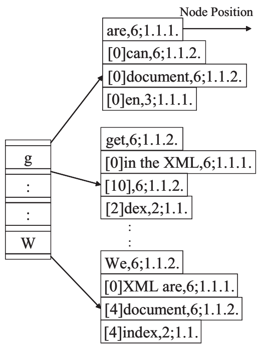

Fig. 4. Prefix-Diff COB-tree.

3.3 Prefix-Diff COB-tree

The size of a search key should be small in a B$^+$-tree. The search key of a COB-tree includes text, and if the phrase length of the text is long, the size of the search key becomes large.

To cope with this problem, we pay attention to the fact that the texts in the search keys contained within the leaf node pages of a COB-tree are ordered lexicographically, and the texts that begin with the same phrase are clustered. We compress the text in the search key by keeping only i) the length of the common prefix with the previous search key; and ii) the following character string after the common prefix. This compression can rebuild the text information in a COB-tree losslessly and reduce the size of search keys. The first search key in a leaf node page must keep the whole original text; however, the other search keys can use the above-described compression technique.

When we search in a COB-tree, the entries are retrieved by a node page block from a disk, and when we search a text in the leaf node page, texts in the search keys are rebuilt first. On the other hand, in the internal node page of a COB-tree, the search key can only be a text, or a text and a PSP pair that is enough to determine which pointer to the child node page should be followed as Prefix B-trees [15].

We call a COB-tree with the above compression a *Prefix-Diff COB-tree*. We call a COB-tree without compression a *Normal COB-tree* when we need to distinguish them. Figure 4 shows a Prefix-Diff COB-tree. The texts of the search keys in the leaf node pages in Figure 4 are compressed. For example, the compressed text "[8]document" of the search key "[8]document,10;1.1.2." indicates that the original text is the same as the text in the previous search key up to the eighth character followed by the different text "document".

4 Query Processing

XPath queries [1] can be processed by traversing XICS and retrieving entries relevant to the nodes corresponding to the query.

Those queries that consist of a path information only can be processed by traversing an STB-tree only once. We call such queries *simple path queries.* On the other hand, those queries that have a *contains()* function for the target node can be processed by traversing a COB-tree only once. We call these kinds of queries *full-text queries.* Simple path queries and full-text queries are the basic units of queries. *Composite queries* which have one or more predicates for the nodes in the path of a query are first decomposed into these basic units. We explain the query processing algorithms first for simple path queries and full-text queries and then for composite queries.

In this section, we show some examples of processing XPath queries for the XML document in Figure 1. We use Table 1 to retrieve the correspondence between the path and the path identifier included in the PSP of a node.

4.1 Simple Path Queries

A simple path query has the form $s_1l_1s_2l_2 \ldots s_kl_k$, where each s_i is "/" or "//" and l_i is a tag name. In this case, we first get the path identifiers that correspond to this path. If either s_i is "//", the multiple path identifiers are possibly retrieved. Then, we traverse the STB-tree with the path identifiers using *search-key mapping.* An example of simple path query process is as follows:

– `//title`
  The path identifiers corresponding to this path are 2 and 5. When traverse in the STB-tree and retrieve entries that have a path identifier between 2 and 5, we can retrieve entries with search keys "2;1.1." and "5;1.1.1.". The result of this query is the node positions of the "title" node included in each entry. This query can be processed efficiently because the two entries are clustered in the index by the *search-key mapping* technique.

4.2 Full-Text Queries

A full-text query has the form $s_1l_1s_2l_2 \ldots s_kl_k[contains(.,'text')]$. In this case, we traverse the COB-tree using the text in the query. The structure information of the query is also checked with the traversal. An example of full-text query processing is as follows:

– `//section/title[contains(., 'Tree')]`
  We traverse the COB-tree using the text "Tree" and the structural information "//section/title", and retrieve the corresponding entries with search key "Tree,5;1.1.1.". We can get the position of the "title" node by following the pointer of this search key.

4.3 Composite Queries

A composite query has the form $s_1l_1[Pred_1]s_2l_2[Pred_2] \ldots s_kl_k[Pred_k]$, where $Pred_i$ is either a simple path query or a full-text query. In this case, the query can be processed by first decomposing the query to "$s_1 l_1 Pred_1$", "$s_1 l_1 s_2 l_2 Pred_2$", ..., "$s_1 l_1 s_2 l_2 \ldots s_k l_k Pred_k$", and "$s_1 l_1 s_2 l_2 \ldots s_k l_k$", and then joining each result. An example of composite query processing is as follows:

– `//section[title[contains(.,'Tree')]]`
  We first decompose this query into q_1="//section/title[contains(.,'Tree')]" and q_2="//section". Then the entry with the search key "Tree,5;1.1.1." in the COB-tree is retrieved as a result of the query q_1, and the entries with search keys "4;1.1.", "4;1.2.", "6;1.1.1.", and "6;1.1.2." in the STB-tree are retrieved as a result of the query q_2. By joining these PSP labels, we know "5;1.1.1." and "4;1.1." are under a parent-child relationship, and we can get the position of the target "section" node by following the pointer of "4;1.1.".

5 Experiments

We implemented XICS and examined its effectiveness. We used GiST (Generalized Search Tree) [12] for the implementation of B$^+$-tree indices. We used the XML documents provided by the INEX Project [13].

We compared XICS with the method proposed in [14], which is compatible with full-text searches using inverted lists on tag names and keywords. We experimented with these inverted lists indexed by a B$^+$-tree. In the rest of the paper, we call the method proposed in [14] *Integration.* We applied a Prefix-Diff approach to the indices except for the STB-tree, and we set $L = 1$ as the threshold L for the maximum phrase length in the COB-tree, because *Integration* does not support phrase searches.

5.1 Index Size

In the experiment on index size, we created and used four kinds of XML document sets, changing the total size of the XML documents.

Figure 5 shows the size comparison of an STB-tree and an inverted list on tag names of *Integration*, and Figure 6 shows the comparison of a COB-tree and an inverted list on keywords. Each index size is nearly proportional to the size of the XML document set. XICS is about 1.4 times larger than *Integration.*

5.2 Query Processing Time

We examined the query processing time with XICS and *Integration* using the XPath queries in Table 2. We used the whole INEX document set (about 495 MB) in the experiment on query processing. Table 3 shows the processing time of these queries.

A join operation between the query text position and the target node position is needed in *Integration* (Q_3, Q_4, Q_5 and Q_6). Furthermore, a join operation for specifying the sibling number in a query is needed in *Integration* (Q_6). On the other hand, in the case of XICS , we need a join operation when the query is

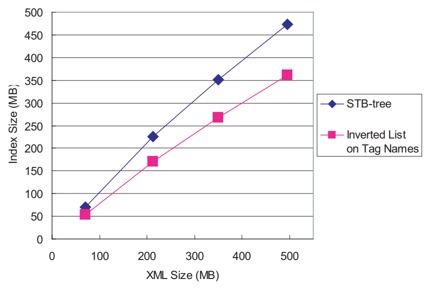

Fig. 5. Size comparison of STB-tree and inverted list on tag names.

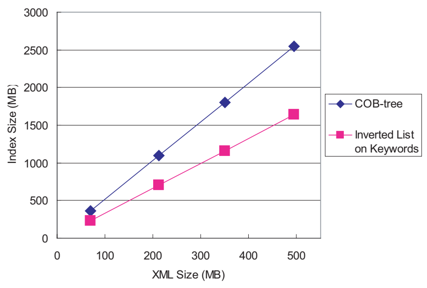

Fig. 6. Size comparison of COB-tree and inverted list on keywords.

Table 2. XPath queries for comparison with *Integration*.

	XPath
Q_1	/books/journal/title
Q_2	//sec/st
Q_3	//article/fm/abs/p[contains(., 'software')]
Q_4	//article[contains(./fm/abs/p, 'software')]
Q_5	//sec[contains(./st, 'animation')]
Q_6	//bdy/sec[1]/st[contains(., 'XML')]

Table 3. Execution time (time in milliseconds).

	XICS	Integration
Q_1	78	65
Q_2	410	1037
Q_3	79	19040
Q_4	278	21450
Q_5	332	3649
Q_6	68	46611

Table 4. Index traversal time (time in milliseconds).

	XICS	Integration
Q_1	78	65
Q_2	410	1037
Q_3	79	5137
Q_4	188	446
Q_5	321	510
Q_6	68	1658

a composite query (Q_4 and Q_5). Since join processing is not the focus of our current study, we did not use any special approach in the join operations. The join operation time depends on the join algorithm. Table 4 shows the index traversal time excluding the join operation time.

XICS achieved an execution time up to 685 times faster than *Integration* with an exception for Q_1. In the case of Q_1, the "title" nodes in the XML document set were very few, and the path information in the search keys of our indices was not so significant. However, in general, we can traverse our indices efficiently by using the path information in the search keys and only retrieve the entries that match the path information in the query. For that reason, we can restrict nodes in the join operation and reduce the whole processing time.

In XICS, we can use the Sibling Dewey Order to specify the sibling number, and we don't need any join operations for it. In general, we need costly join operations to specify the sibling number in other approaches including *Integration*.

6 Conclusions

In this paper, we proposed using XICS to accelerate the process of XPath queries. XICS is based on a B$^+$-tree and can efficiently process queries that involve structural and full-text searches of XML documents. We particularly concentrated on texts in XML documents and constructed a COB-tree using PSP that contained path information from the root node and the text fragments in the XML document. In addition, we constructed an STB-tree for processing structural queries. *Search-key mapping* enables the efficient processing of a query containing "//". We proposed a compression method in the COB-tree and built a Prefix-Diff COB-tree. We then showed the processing steps for an XPath query using XICS. The experiment results show that XICS is about 1.4 times larger than *Integration.* Paying this slight increase in the cost of the index size, XICS outperforms *Integration* up to 685 times in terms of search time.

Future works include: a more appropriate choice of PSP, a pointer in the leaf pages of the B$^+$-tree, ordering that is not based on the reverse paths in *search-key mapping*, improvement of the join operation, introduction of data statistics and query workloads, and consideration of document updates.

References

1. W3C, XPath 1.0, http://www.w3.org/TR/xpath, 1999.
2. W3C, XQuery 1.0, http://www.w3.org/TR/xquery/, 2005.
3. Q. Li and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions," In *VLDB*, pp.361–370, September 2001.
4. H. Jiang, H. Lu, W. Wang, and B. C. Ooi, "XR-Tree: Indexing XML Data for Efficient Structural Joins," In *ICDE*, pp.253–264, March 2003.
5. H. Wang and X. Meng, "On the Sequencing of Tree Structures for XML Indexing," In *ICDE*, pp.372–383, April 2005.
6. R. Goldman and J. Widom, "Dataguides: Enabling query formulation and optimization in semistrucutred databases," In *VLDB*, pp.436–445, August 1997.
7. B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon, "A Fast Index for Semistructured Data," In *VLDB*, pp.341–350, September 2001.
8. X. Wu, M. L. Lee, and W. Hsu, "A Prime Number Labeling Scheme for Dynamic Ordered XML Trees," In *ICDE*, pp.66–78, March 2004.
9. B. C. Hammerschmidt, M. Kempa, and V. Linnemann, "A selective key-oriented XML Index for the Index Selection Problem in XDBMS," In *DEXA*, 2004.
10. W3C, XQuery 1.0 and XPath 2.0 Full-Text Use Cases, http://www.w3.org/TR/xmlquery-full-text-use-cases/, April 2005.
11. S. Amer-Yahia, C. Botev, and J. Shanmugasundaram, "TeXQuery: A Full-Text Search Extension to XQuery," In *WWW*, pp.583–594, May 2004.
12. J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, "Generalized Search Trees for Database Systems," In *VLDB*, pp.562–573, September 1995.
13. INitiative for the Evaluation of XML Retrieval (INEX), http://inex.is.informatik.uni-duisburg.de/2005/.
14. R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan, "On the Integration of Structure Indexes and Inverted Lists," In *SIGMOD*, June 2004.
15. R. Bayer and K. Unterauer, "Prefix B-trees," ACM Trans. on Database Systems, vol.2, no.1, pp.11–26, March 1977.