An Efficient XML Index Structure with Bottom-Up Query Processing

Dong Min Seo, Jae Soo Yoo, and Ki Hyung Cho

Department of Computer and Communication Engineering, Chungbuk National University,
48 Gaesin-dong, Cheongju Chungbuk, Korea
{dmseo, yjs, khjoe}@chungbuk.ac.kr

Abstract. With the growing importance of XML in data exchange, much research has been done in proving flexible query mechanisms to extract data from structured XML documents. The semi-structured nature of XML data and the requirements on query flexibility pose unique challenges to database indexing methods. Recently, ViST that uses suffix tree and B^+Tree was proposed to reduce the search time of the documents. However, it can cause a lot of unnecessary computation and I/O when processing structural join queries because the numbering scheme of ViST is not optimized. In this paper, we propose a novel index structure to solve the problems of ViST. Our index structure provides the bottom-up query processing method to efficiently process structural queries. Our experiments show that the proposed index structure is efficient in processing both single-path and branching queries with various wild-cards ('*' and '//').

Keywords: XML Index, XML Query, Bottom-Up Query Processing.

1 Introduction

XML provides a flexible way to define semi-structured data with the hierarchical structure. Because of such features of XML, XML is represented by a tree structure to process XML data [1]. Several query languages, including XPath, Quilt, XML-QL, and XQuery, have been proposed for semi-structured XML data. Also, path index methods that construct a graphical index on XML data to reduce query costs have been worked [2, 3].

Recently, many research papers on the structural join methods to efficiently process the XML queries involving the ancestor-descendant relationship have been done [4, 5, 6]. The structural join methods can answer simple queries efficiently. However, queries involving branching structures usually have to be disassembled into multiple subqueries. The results of these subqueries are then combined by expensive join operations to produce final answer.

To improve existing structural join methods, ViST that combines suffix tree and B^+Tree was proposed [7]. The suffix tree with sequence matching matches structured queries against structured data to avoid many unnecessary structural join operations. B^+Tree avoids a traversal of the whole XML document tree. However, the numbering

Y. Shi et al. (Eds.): ICCS 2007, Part III, LNCS 4489, pp. 813–820, 2007.

scheme used in suffix tree is not optimized because it regards element *y* that does not have ancestor-descendant relation with element *x* in XML data as the descendent of the element *x*. For the same reason, ViST can cause a lot of unnecessary computation and I/O when processing the queries.

In this paper, we propose an efficient index structure to solve problems of ViST and a novel query processing method suitable for proposed index structure. To verify the efficiency of our index structure, we compare the performance of our index structure with that of ViST.

The rest of this paper is organized as follows. In section 2, we review related works and describe the problems of ViST. In section 3, we present our proposed indexing technique. In section 4, we present the experimental results that compare our technique with ViST. Finally, conclusions and future works are discussed in section 5.

2 Related Work

2.1 Query Processing Methods Using Suffix Tree

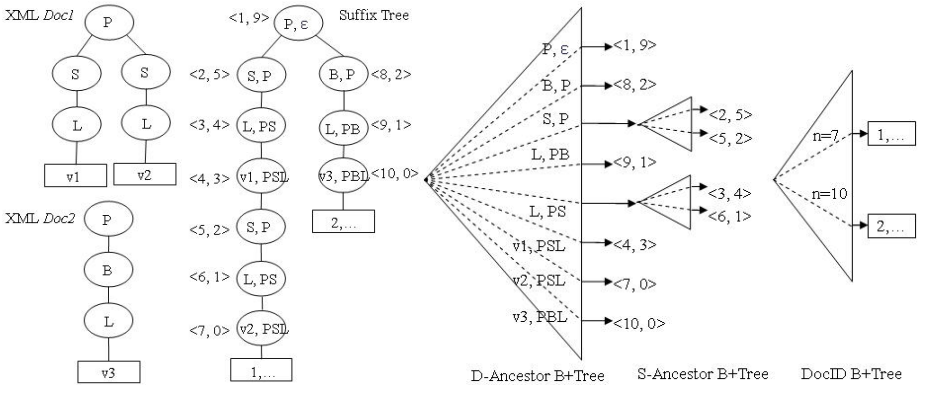
Recently, query processing methods using suffix tree were proposed [8]. This methods use a sequential representation of both XML data and XML queries. Querying XML is equivalent to finding subsequence matches. Sequence matching matches structured queries against structured data as a whole without breaking down the queries into subqueries of paths or nodes and relying on join operations to combine their results. The purpose of modeling XML queries through sequence matching is to avoid as many unnecessary join operations as possible in query processing.

After both XML data and XML queries are converted to structure-encoded sequences in the suffix tree, it is straightforward to devise the naive algorithm to perform sequence matching. However, there are several difficulties. First, searching for the satisfied nodes is extremely costly since we need to traverse a large portion of the subtree for each match. Finally, suffix tree is main memory structure that is seldom used for disk resident data, and most commercial DBMSs do not have support for such structures [7].

2.2 The Problems of ViST

ViST improves the naive algorithm by eliminating costly suffix tree traversal. It uses a virtual suffix tree to organize structure-encoded sequences to speed up the matching process. ViST labels each suffix tree node *x* by a pair $<n_x, size_x>$, where n_x and $size_x$ are the prefix traversal order of *x* and the total number of descendants of *x* in suffix tree, respectively. Figure 1 shows the index structure of ViST. ViST consists of three parts such as the D-Ancestor B^+Tree, the S-Ancestor B^+Tree and the DocID B^+Tree. To construct the B^+Trees of ViST, it first inserts all suffix tree nodes into the D-Ancestor B^+Tree using their (*symbol*, *prefix*) as keys. For all nodes *x* inserted with the same (*symbol*, *prefix*), ViST indexes them by an S-Ancestor B^+Tree, using the n_x values of their labels as keys. In addition, ViST also builds a DocId B^+Tree, which stores for each node *x* (using n_x as key), the document IDs of those XML sequences that end up at node *x* when they are inserted into the suffix tree [7]. ViST simply

determines the ancestor-descendant relationship between two elements by the D-Ancestor B$^+$Tree and the S-Ancestor B$^+$Tree. If x and y are labeled $< n_x, size_x >$ and $< n_y, size_y >$ respectively, node x is the ancestor of node y iff $n_y \in (n_x, n_x+size_x]$. As a result, ViST no longer need to search the descendent of x to find such y.



(a) The suffix tree for *Doc1* and *Doc2* (b) The structure of ViST

Fig. 1. The ViST index structure

Figure 2 shows the query processing method in ViST of Figure 1. If the query for "*/P/S/L/v1*" is requested, a structure-encoded sequence about this query is constructed. And it obtains the n_x and $size_x$ of the (P, ε) that is the first (*symbol*, *prefix*) of above structure-encoded paths by searching the D-Ancestor B$^+$Tree and the S-Ancestor B$^+$Tree of Figure 1.

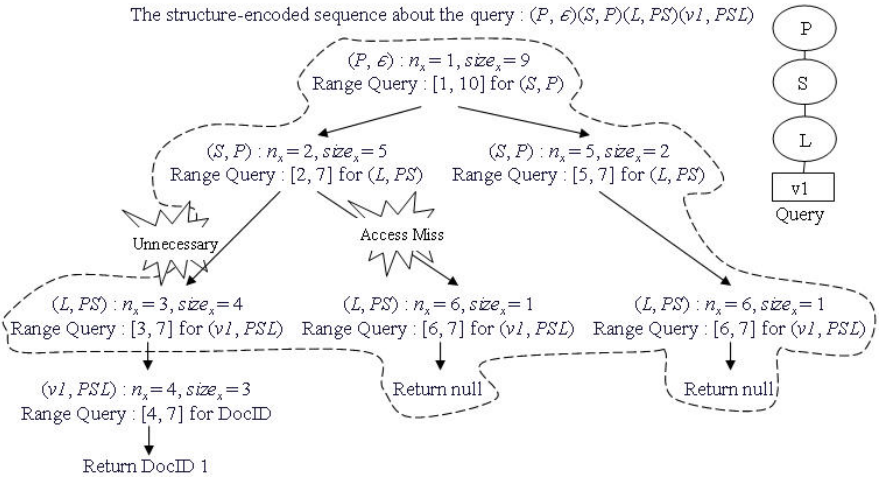


Fig. 2. The query processing in ViST

These values are used for range queries about the (*symbol*, *prefix*) of (*S*, *P*) involved in the (n_x, n_x+$size_x$) of the (*P*, ε). Then the range query about the (*symbol*, *prefix*) of the (*L*, *PS*) involved in the <n_x, n_x+$size_x$> of the (*S*, *P*) and the (*symbol*, *prefix*) of the (*v1*, *PSL*) involved in the <n_x, n_x+$size_x$> of the (L, *PS*) are executed with the set of above range query results, respectively. Finally, to obtain the documents corresponded to the query, the range query about the [n_x, n_x+$size_x$] of the last (*symbol*, *prefix*) is executed in the DocID B$^+$Tree.

As shown in Figure 2, the child node of the (*S*, *P*) with n_x=2 is only the (*L*, *PS*) with n_x=3 in the *Doc1* of Figure 1. However, when the range query about the (*symbol*, *prefix*) of the (*L*, *PS*) involved in above (*S*, *P*) is executed, the (*L*, *PS*) with n_x=6 is represented as the child of above (*S*, *P*). Moreover, because the characteristic that prefix of an element represents the path from the root element to itself is not used, the unnecessary nodes are accessed as shown in Figure 2. For example, with the *PSL* of the (*v1*, *PSL*), we estimate that the (*v1*, *PSL*) has the (*L*, *PS*) as its parent node, and the (*P*, ε) and (*S*, *P*) as its ancestor nodes. Therefore the range queries about the (*P*, ε), (*S*, *P*), and (*L*, *PS*) involving the (*v1*, *PSL*) are unnecessary. Also, because of noncontiguous subsequence matches, the branching query processing method of ViST triggers false alarms [9].

3 Our Proposed Index Structure

3.1 Our Proposed Index Structure

We use the *durable* numbering scheme [4] to solve the numbering scheme problem of ViST. Figure 3 shows our proposed index structure about the XML documents of Figure 1. The D-Ancestor B$^+$Tree and the S-Ancestor B$^+$Tree of our index structure are same as those of ViST. But our index structure inserts a pair <*order*, *size*> of the *durable* numbering scheme into the S-Ancestor B$^+$Tree.

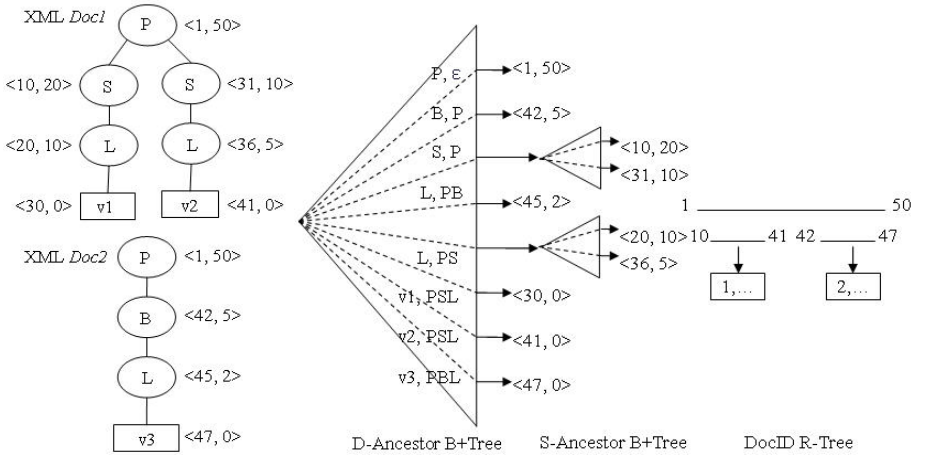
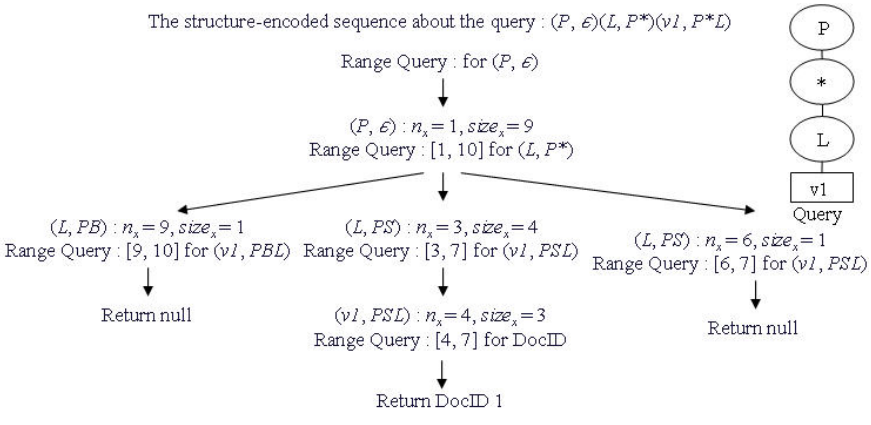


Fig. 3. Our proposed index structure

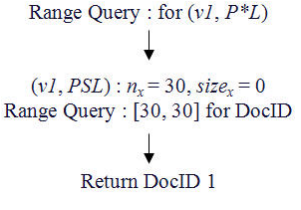
To directly find the documents involving the <n_x, $size_x$> of the last node by the range query, we use the DocID R-Tree instead of the DocID B$^+$Tree. The key of DocID R-Tree is assigned with a pair of numbers <*start*, *end*>. Element *start* is the minimum order and *end* is the maximum order in an XML document except for the root element.

3.2 Our Proposed Bottom-Up Query Processing Algorithm

Each prefix of nodes represents the path from the root node to each node. Therefore, when a query is executed, if we use the characteristic of the prefix, the query performance is significantly improved. We propose the bottom-up query processing method. Figure 4 shows the query processing methods in ViST and the proposed index structure. If the query of Figure 4(a) is executed by the top-down query processing method of ViST, ViST performs a range query to match (*L*, *P**) of the query sequence in the D-Ancestor B$^+$Tree of Figure 1 after the search for the (*P*, ε). The search then continues on the S-Ancestor B$^+$Tree with the results returned by the range query so the searches for the (*L*, *PB*) and the (*L*, *PS*) are executed. However, if the query is executed by our bottom-up processing method, our proposed index structure performs only a range query to match last element (*v1*, *P***L*) of the query sequence in the D-Ancestor B$^+$Tree of Figure 3.



(a) By the top-down query processing in ViST



(b) By the bottom-up query processing in our index structure

Fig. 4. The bottom-up wild-card query processing

As a result, in a single path query case, our method only executes the range query about last element of a query sequence. Moreover, as shown in Figure 4, our bottom-up query processing method is very efficient for wild-cards ('*' and '//') queries because node accesses for wild-cards in our bottom-up query processing method are smaller than that in ViST.

Figure 5 shows the branching query processing method in proposed index structure. First, to find (*v2*, *PSL*), our index structure performs the search in the D-Ancestor B$^+$Tree and the S-Ancestor B$^+$Tree of Figure 3. The search then continues on the D-Ancestor B$^+$Tree and the S-Ancestor B$^+$Tree to find (*L*, *PS*) involving $<n_x, size_x>$ of (*v2*, *PSL*). The search then finds (*S*, *P*) involving $<n_x, size_x>$ of (*L*, *PS*) returned by the search for (*L*, *PS*). Then our index structure performs the search to find (*v1*, *PSL*) that is involved in $<n_x, size_x>$ of (*S*, *P*) and isn't involved in $<n_x, size_x>$ of (*L*, *PS*) that is the parent node of (*v2*, *PSL*). This processing method avoids unnecessary computation and I/O of the search to find (*P*, ε) and (*L*, *PS*) that are the parent nodes of (*v1*, *PSL*). In addition, our bottom-up query processing method for the branching query has no false alarm because it processes the branching query with a pair <*order*, *size*> of the *durable* numbering scheme. In Figure 5, our bottom-up query processing executes the range queries about (*S*, *P*) and (*L*, *PS*) to avoid false alarms.

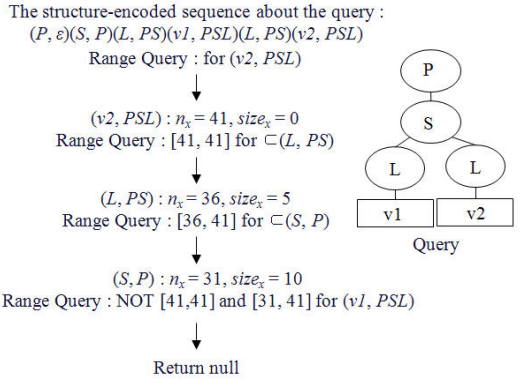


Fig. 5. The bottom-up branching query processing

4 Performance Evaluation

4.1 Experimental Setup

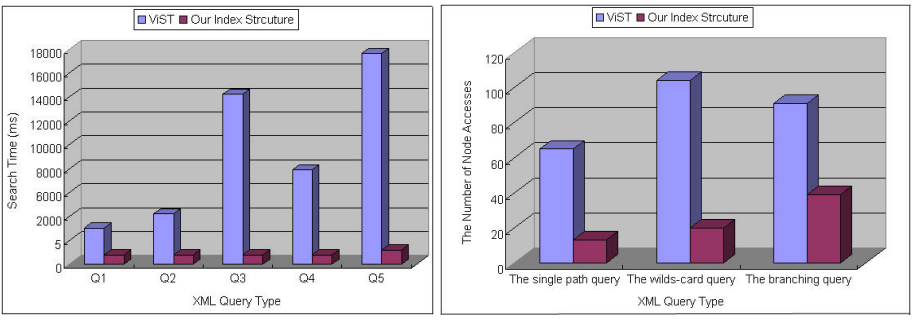
To determine the effectiveness of our index structure, we compare the performance of our index structure with that of ViST. Our results indicate that our index structure outperforms ViST. We implemented our XML indexing in C++. The implementation uses the B$^+$Tree API provided by the Berkeley DB library [10]. All the experiments are carried out on a 3GHz Pentium □ processor with 512 MB of RAM and Windows XP. We use the same set of queries for NIAGARA [11] with some slight changes on value predicates as shown in Table 1.

Table 1. Sample queries for the performance evaluation

No	Description
Q1	/W4F_DOC/Actor/Name/FirstName/Robert
Q2	/W4F_DOC/Actor/Name/*/Rebert
Q3	//Name/*/Robert
Q4	/W4F_DOC/Actor/Name[FirstName/Claudio]/LastName/Alfonsi
Q5	//*/Name[FirstName/Claudio]/LastName/Alfonsi

4.2 Performance Evaluation Results

The query processing performances of our index structure and ViST are illustrated in Figure 6.



(a) The average query processing time (b) The average number of node accesses

Fig. 6. The query processing performance about ViST and the proposed index structure

Figure 6(a) shows the average query processing time to process queries. *Q1* represents a single path query. *Q1* is evaluated with changing the depth of a query path. *Q2* represents the query with the wild-card '*' and *Q3* represents the query with wild-card '//'. *Q2* and *Q3* are evaluated with changing the appearance position and the number of wild-cards. *Q4* represents a multiple path query. *Q4* is evaluated with changing the number of the branches of queries. *Q5* represents a complex query that combines wild-cards '*' and '//'. Our index structure takes the same time in all cases because our bottom-up query processing method searches only last element of a query sequence. But, if the depth of the query path is longer, ViST takes longer time because it searches all elements of a query sequence. As a result, our index structure outperforms ViST because the proposed bottom-up query processing method avoids a lot of unnecessary computation and I/O of ViST.

Figure 6(b) shows the number of node accesses to process a query. This experiment is performed with single path queries, wild-cards queries and branching queries. Our index structure significantly outperforms ViST in all cases. In our index structure, the number of node accesses for a branching query is larger than that for a single path query and a wild-cards query. The reason is that our bottom-up query processing method for a branching query requires many range queries about elements in a query

sequence to avoid false alarms. As a result, our index structure is more efficient than ViST because our index structure has no false alarms.

5 Conclusion

In this paper, we have proposed a novel index structure for indexing XML data and processing XML queries. Our index structure provides the D-Ancestor B^+Tree and the S-Ancestor B^+Tree using the *durable* numbering scheme to efficiently determine structural relationship between any pair of element nodes and find all occurrences of structural relationships between two element sets. In addition, our index structure provides the bottom-up query processing to avoid a lot of unnecessary computation and I/O for structural join queries and has no false alarms. We have performed various experiments to evaluate the effectiveness of our index structure. Our studies show that our index structure significantly outperforms ViST for all the tested queries. In the future, we will study how to efficiently process the delimiters of the prefix schemes, to decrease the label size and to keep low label update costs.

Acknowledgement. This work was supported by the Korea Research Foundation Grant funded by the Korean Government(MOEHRD)(The Regional Research Universities Program/Chungbuk BIT Research-Oriented University Consortium) and University Fundamental Research Program supported by Ministry of Information & Communication in Republic of Korea.

References

1. T. Bray, J. Paoli, C. M. et. al., “Extensible Markup Language (XML) 1.0 3rd”, http://www.-w3.org/TR/REC-xml (2004)
2. M. Fernandez and D. Suciu, “Optimizing regular path expressions using graph schemas”, In -ICDE (1998) 14-23
3. C. W. Chung, J. K. Min, K. S. Shim, “APEX: An Adaptive Path Index for XML Data”, In S-IGMOD (2002) 121-132
4. Q. Li and B. Moon, “Indexing and querying XML data for regular path expressions”, In VL-DB (2001) 361-370
5. D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, and Y. Wu, “Structural joins: A primitive for efficient XML query pattern matching”, In ICDE (2002) 141-152
6. S, -Y, Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo, “Efficient structural joins on indexed XML documents”, In VLDB (2002) 263-274
7. H. Wang, S. Park, W. Fan, and P. S. Yu, “ViST: A Dynamic Index Method for Querying X-ML Data by Tree Structures”, In SIGMOD (2003) 110-121
8. E. M. McCreight, “A space-economical suffix tree construction algorithm”, Journal of the ACM (1976) 262-272
9. H. Wang and X. Meng, “On the Sequencing of Tree Structures for XML Indexing”, In ICD-E (2005) 372-383
10. Sleepycat Software, “http://www.sleepycat.com”, The Berkeley Database
11. The Niagara Project Group, “The Niagara Project Experimental Data”, http://www.cs.wisc.-edu/niagara/data.html (2005)