Available online at www.sciencedirect.com

ScienceDirect

The Journal of Systems and Software 81 (2008) 56–70

The Journal of Systems and Software

www.elsevier.com/locate/jss

Dynamic interval-based labeling scheme for efficient XML query and update processing

Jung-Hee Yun [a,*], Chin-Wan Chung [b]

[a] *ITA & Standard Team, IT Performance Evaluation Division, National Information-society Agency, NIA Bldg. 77, Mugyo-dong, Jung-gu, Seoul 100-775, Republic of Korea*

[b] *Division of Computer Science, Department of Electrical Engineering & Computer Science, Korea Advanced Institute of Science and Technology, 373-1, Guseong-dong, Yuseong-gu, Daejeon 305-701, Republic of Korea*

Received 21 August 2006; received in revised form 30 April 2007; accepted 22 May 2007
Available online 7 June 2007

Abstract

XML data can be represented by a tree or graph structure and XML query processing requires the information of structural relationships among nodes. The basic structural relationships are parent–child and ancestor–descendant, and finding all occurrences of these basic structural relationships in an XML data is clearly a core operation in XML query processing. Several node labeling schemes have been suggested to support the determination of ancestor–descendant or parent–child structural relationships simply by comparing the labels of nodes. However, the previous node labeling schemes have some disadvantages, such as a large number of nodes that need to be relabeled in the case of an insertion of XML data, huge space requirements for node labels, and inefficient processing of structural joins. In this paper, we propose the nested tree structure that eliminates the disadvantages and takes advantage of the previous node labeling schemes. The nested tree structure makes it possible to use the dynamic interval-based labeling scheme, which supports XML data updates with almost no node relabeling as well as efficient structural join processing. Experimental results show that our approach is efficient in handling updates with the interval-based labeling scheme and also significantly improves the performance of the structural join processing compared with recent methods.
© 2007 Elsevier Inc. All rights reserved.

Keywords: XML query; XML update; XML labeling scheme

1. Introduction

As XML is widely being used for data exchange, there have been many researches about the problem of storing, managing and querying XML data efficiently. Generally XML data is modeled by a tree or graph structure, where nodes represent elements, attributes and text data, and the parent–child relationship between two nodes is represented by an edge.

Several query languages such as XPath (Clark and DeRose, 1999) and XQuery (Chamberlin et al., 2001) have been proposed to process XML data, and handling tree pattern matching, especially the ancestor–descendant or parent–child structural relationship, is important to execute queries efficiently. In order to improve the processing time for deciding these structural relationships, several node labeling and index schemes have been suggested (Li and Moon, 2001; Zhang et al., 2001; Wu et al., 2004; Tatarinov et al., 2002; Kaplan et al., 2002; Catania et al., 2005; Chen et al., 2004; Yoshikawa and Amagasa, 2001). Most of them model an XML document as a node-labeled tree and every node is given a unique identifier (label) based on its location in the document or its order in the XML data tree traversal. These labels can be used to determine whether ancestor–descendant or parent–child structural relationship between two nodes exists.

* Corresponding author. Tel.: +82 2 21310428.
E-mail addresses: yunjh@nia.or.kr (J.-H. Yun), chungcw@cs.kaist.ac.kr (C.-W. Chung).

In the case of the interval-based labeling scheme (Li and Moon, 2001; Zhang et al., 2001) that assigns start and end position numbers as the label to each node, the label of each node can be determined by the sequential assignment of positive integer numbers during the depth first traverse of an XML data tree. By using this labeling scheme, it is possible to decide the ancestor–descendant or parent–child structural relationship between two nodes very simply. Also the feature of interval-based labeling, in which the interval (between the start position and the end position) of an ancestor node includes the interval of a descendant node, makes it feasible to process structural join efficiently. Several algorithms and index structures (Srivastava et al., 2002; Chien et al., 2002; Jiang et al., 2003) using this feature for efficient structural join processing have been studied. However they cannot handle dynamic updates efficiently because of the sequential numbering. To solve this problem, additional space is reserved for future data insertions (Li and Moon, 2001), but after several data insertions the space required to hold inserted data has exceeded the reserved space and in the worst case the relabeling of the whole data tree is needed.

In the prefix labeling scheme (Tatarinov et al., 2002; Kaplan et al., 2002), the nodes in XML data tree are labeled such that the ancestor–descendant structural relationship between two nodes is determined by whether one label is the prefix of the other. If the order of nodes does not need to be stored, data insertions do not affect the labels of existing nodes. But if the order of nodes must be stored, insertions cause changes in the labels. Also the large size of labels and delimiters incurs high storage overhead, and when the fan-out of the XML tree is large, the size of labels could be large. Moreover, structural join processing using a prefix labeling scheme is less efficient than those using an interval-based labeling scheme because the prefix comparison is slower than the simple integer comparison.

The prime number labeling scheme (Wu et al., 2004) is based on the property of prime numbers. This scheme assigns a unique prime number to each node as the self-label and the label of each node is the product of its parent node's label and its own self-label. In this labeling scheme, the determination of the ancestor–descendant structural relationship between any two nodes depends on whether the label of a descendant-candidate node is divisible by the label of an ancestor-candidate node. When a new node is inserted, it is easy to assign a prime number that has not been assigned before as the self-label for any node. However the space size for the node label is huge because the label of each node is the product of self-labels from the root to each node. Also several algorithms and index structures (Srivastava et al., 2002; Chien et al., 2002; Jiang et al., 2003) for efficient structural join processing cannot be applied under this labeling scheme which does not have the property of the interval, and $m \times n$ scans are necessary for the structural join when the ancestor-candidate node list has m nodes and the descendant-candidate node list has n nodes. Therefore this scheme cannot support the efficient XML query processing.

In (ONeil, 2004), a new labeling scheme called ORDPATH, a dynamic variant of the Dewey order, is provided. In ORDPATH, the label of each node is determined by the Dewey order scheme except that it reserves even and negative integers for later insertions into an existing tree. Also it stores the label of each node as the compressed binary representation and the ancestor–descendant or parent–child structural relationship between two nodes is determined by the substring comparison. Because of the reserved even and negative integers, almost no node relabeling occurred for new data insertions. But the length of the binary representation of the label is very long and becomes longer by frequent data insertions.

Recently (Catania et al., 2005) proposes a new XML update approach, the lazy XML update, which deals with both XML updates and structural join processing in an efficient way, based on the use of segments. The segment is a set of elements that must be inserted into or deleted from the XML database and this approach takes a segment as the unit of updates. Each segment is labeled by the global position, local position and length. The segment containment relationship can be determined by the labels of two segments. In this approach, the traditional structural join algorithm is improved into the segment-based extended algorithm, which improves query performance. However by the XML data insertion, the global position and the length of each segment must be relabeled, and the total number of segments is limited because all segments must be in memory. Also this approach is not applicable to the real time update processing of XML data. In order to manage the segments, the update log, the SB-tree and tag-list, must be maintained additionally.

In this paper, we propose an effective node labeling scheme that solves the weak points of the previous schemes by supporting efficient update and query processing. XML document consists of elements and we can consider an element as a unit of XML data update. These elements are expressed by subtrees in XML data tree, and the XML data insertion and deletion can be treated by the combination of subtree insertions and subtree deletions respectively.

The interval-based labeling scheme is weak in data updates because of the interval property of node labels. Although additional space is reserved for the future data insertions, node relabeling is inevitable in the case of a large data insertion or many data insertions. This is because the reserved space size is not sufficient for the data insertions, so it is solvable if we can process the insertion of a large XML data with small space. Motivated by this observation, we propose a nested tree structure for the dynamic interval-based labeling scheme. The inserted subtree is labeled as one leaf node and the nodes in subtree are labeled by new numbering. Then the ancestor–descendant or parent–child structural relationship between one node in the subtree and another node not in the subtree is determined by comparing the label of the subtree and the label of the latter node. The ancestor–descendant or parent–child structural relationship between two nodes in the subtree can be obtained by the new labels marked in the scope of the subtree. If the data insertion occurs in the previous inserted subtree, a new subtree is formed in the inserted subtree. As the data insertions like this occurs continually, the structure of the whole tree is nested by subtrees, and we call it the Nested Tree structure.

In the proposed Nested Tree structure, the numbering scheme is basically the same as the traditional interval-based numbering scheme except that an integer list is used to represent the start position and the end position instead of an integer. If the integer comparison operation is changed to the integer list comparison operation, the traditional structural join algorithm based on the interval-based numbering scheme such as Tree-Merge-Anc and Stack-Tree-Desc (Srivastava et al., 2002) can be applied with no change.

The contributions of this paper are the following:

- We propose new structures, the Nested Tree structure and Nested Inverted List for the dynamic interval-based node labeling and efficient structural join processing. Results of extensive experiments show that our approach is more efficient than representative existing approaches for update as well as structural join processing. Especially, our approach is on the average 6.2 times and up to 52.1 times faster than the most efficient representative approach for structural join processing while on the average 10% faster for update processing.
- We present an XML data insert and delete processing method with almost no node relabeling using the advantage of the Nested Tree structure.
- We show the traditional structural join algorithms can be applied to our model just with the change of the comparison operation between two labels. Also we develop an enhanced structural join algorithm using the proposed Nested Inverted List.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 proposes the Nested Tree structure, while Section 4 discusses the XML data update and delete algorithms. We describe the method of query processing using the Nested Tree structure in Section 5. Section 6 reports experimental results and then Section 7 concludes the paper.

2. Related work

XML data is generally modeled by a node-labeled tree structure, where nodes represent elements, attributes and text data, and the label of each node stands for the name of an element, attribute, or the text data. To efficiently process complex XML queries which can be represented by the query languages, such as XPath (Clark and DeRose, 1999) or XQuery (Chamberlin et al., 2001), it is important to quickly determine the ancestor–descendant or parent–child structural relationship between any pair of tree nodes and to efficiently find all occurrences of these structural relationships.

To support efficient XML query processing, several node labeling schemes in the XML data tree were proposed. And then several algorithms and index structures using these schemes for XML query processing have been studies.

In the interval-based labeling scheme, such as (Li and Moon, 2001), the label of any node in a XML data tree is represented as the tuple (`DocID`, `StartPos`, `EndPos`, `LevelNum`), where (i) `DocID` is the identifier of the document; (ii) `StartPos` and `EndPos` can be generated by counting the number of words from the beginning of the document with identifier `DocID` to the start of the element and end of the element, respectively, or by the sequential assignment of positive integers during the depth first traversal of XML data tree; and (iii) `LevelNum` is the depth of the element in the document.

The ancestor–descendant or parent–child structural relationship between two tree nodes N_1 and N_2 whose positions are recorded in this fashion, such as (D_1, S_1, E_1, L_1) and (D_2, S_2, E_2, L_2), can be determined easily: (i) *ancestor–descendant*: N_2 is a descendant of N_1 iff $D_1 = D_2$, $S_1 < S_2$ and $E_2 < E_1$; (ii) *parent–child*: N_2 is a child of N_1 iff $D_1 = D_2$, $S_1 < S_2$, $E_2 < E_1$ and $L_1 + 1 = L_2$.

There have been many studies about the structural join algorithms using the interval-based labeling scheme. (Srivastava et al., 2002) proposed two families of structural join algorithms for matching parent–child and ancestor–descendant relationships efficiently: Tree-Merge and Stack-Tree. The Stack-Tree-Desc is the most efficient algorithm among four algorithms in (Srivastava et al., 2002). It assumes that each element list of the inverted list is stored ordered on `StartPos` and a stack mechanism is used to maintain elements that will be used later in the join. This leads to the optimal join performance, which is proportional to the input and output size.

(Chien et al., 2002) proposed the stack-based structural join algorithm that uses the B+-tree index built on the `StartPos`. They enhance the stack-based structural join algorithm by avoiding the comparisons of some of the elements that do not participate in the join.

(Jiang et al., 2003) proposed the index for efficient structural joins, XR-Tree. It is the XML Region Tree, which is a dynamic external memory index structure. This index can be used for an efficient new structural join algorithm that can evaluate the ancestor–descendant structural relationship between two XR-tree indexed element sets by skipping ancestors and descendants that do not participate in the join.

These algorithms and index structures were made by using the interval-based labeling scheme. Therefore the labeling schemes that produce a single label for each node such as the prefix labeling and the prime number labeling scheme (Tatarinov et al., 2002; Kaplan et al., 2002; Wu et al., 2004) cannot be applied to these algorithms and index structures.

In the prefix labeling scheme (Tatarinov et al., 2002), the nodes in XML data tree are labeled such that the ancestor–descendant structural relationship between two nodes is determined by whether one label is the prefix of the other. If the order of nodes does not need to be stored, data insertions do not affect the labels of existing nodes. But if the order of nodes must be stored, insertions cause changes in the labels. Also the large size of labels and delimiters incurs high storage overhead, and when the fan-out of the XML tree is large, the size of labels could be large. Moreover, structural join processing using a prefix labeling scheme is less efficient than those using an interval-based labeling scheme because the prefix comparison is slower than the simple integer comparison.

The prime number labeling scheme (Wu et al., 2004) is based on the property of prime numbers. This scheme assigns a unique prime number to each node as the self-label and the label of each node is the product of its parent node's label and its own self-label. In this labeling scheme, the determination of the ancestor–descendant structural relationship between any two nodes depends on whether the label of a descendant-candidate node is divisible by the label of an ancestor-candidate node. When a new node is inserted, it is easy to assign a prime number that has not been assigned before as the self-label for any node. However the space size for the node label is huge because the label of each node is the product of self-labels from the root to each node. Also several algorithms and index structures (Srivastava et al., 2002; Chien et al., 2002; Jiang et al., 2003) for efficient structural join processing cannot be applied under this labeling scheme which does not have the property of the interval, and $m \times n$ scans are necessary for the structural join when the ancestor-candidate node list has m nodes and the descendant-candidate node list has n nodes. Therefore this scheme cannot support the efficient XML query processing.

In (ONeil, 2004), a new labeling scheme called ORDPATH, a dynamic variant of the Dewey order, is provided. In ORDPATH, the label of each node is determined by the Dewey order scheme except that it reserves even and negative integers for later insertions into an existing tree. Also it stores the label of each node as the compressed binary representation and the ancestor–descendant or parent–child structural relationship between two nodes is determined by the substring comparison. Because of the reserved even and negative integers, almost no node relabeling occurred for new data insertions. But the length of the binary representation of the label is very long and becomes longer by frequent data insertions.

(Silberstein et al., 2005) proposes new data structures, W-BOX and B-BOX, that efficiently maintain order-based labeling for dynamic tree-structured XML data. The proposed data structures handle arbitrary update patterns while consuming minimal amount of storage. The two structures provide tradeoff between update and lookup costs. W-BOX uses weight-balanced B-trees to reduce the relabeling overhead, obtaining a logarithmic amortized update cost and constant worst-case lookup cost, whereas B-BOX further reduces update costs, resulting in a constant amortized update time and logarithmic worst-case lookup cost, by avoiding the storage of labels. While the reported theoretical and experimental results are good, no experimental results for queries are reported.

Recently (Catania et al., 2005) proposes a new XML update approach, the lazy XML update, which deals with both XML updates and structural join processing in an efficient way, based on the use of segments. The segment is a set of elements that must be inserted into or deleted from the XML database and this approach takes a segment as the unit of updates. Each segment is labeled by the global position, local position and length. The segment containment relationship can be determined by the labels of two segments. In this approach, the traditional structural join algorithm is improved into the segment-based extended algorithm, which improves query performance. However by the XML data insertion, the global position and the length of each segment must be relabeled, and the total number of segments is limited because all segments must be in memory. Also this approach is not applicable to the real time update processing of XML data. In order to manage the segments, the update log, the SB-tree and tag-list, must be maintained additionally.

(Lu et al., 2005) proposes a new labeling scheme, an extended Dewey labeling scheme, in which all the elements names along the path from the root to the element can be derived from the label of an element alone. Based on extended Dewey they design a novel holistic twig join algorithm, TJFast. To answer a twig query, TJFast only needs to access the labels of the leaf query nodes. This reduces disk access and supports an efficient evaluation of queries with wildcards in branching nodes. However, if there is not the DTD of the XML data, to construct the finite state transducer for the extended Dewey labels the scan of the whole XML data is necessary. Also this scheme is not suitable for the update processing because the labels of whole nodes and the finite state transducer must be reconstructed after data insertions.

3. Nested tree structure

The interval-based labeling scheme cannot support the dynamic update of XML data efficiently because it takes the sequential numbers as the labels of nodes and there is the interval property between the start and end positions for each node. When a new node is inserted, re-labelings of the existing nodes are indispensable. In order to solve this problem, it is possible to leave additional space between any two nodes for future XML data insertions. However, the size of the space cannot be decided simply because update patterns are not fixed generally.

While the inserted XML data can be regarded as one XML element, namely a small tree representing the inserted XML data, the insertion of this tree can be treated as the insertion of one node into the original XML tree. If the number of nodes in that small tree is 100, the space of more than 200 in the inserted position is needed for inserting this data in the original interval-based labeling scheme. However, if the inserted data is regarded as one XML element or one small tree, one space is sufficient for insert processing.

Based on this observation of XML data insertion, we propose the Nested Tree structure for the dynamic interval-based labeling scheme. For example, in Fig. 1, if a new section is inserted into the second book, represented by the dashed box, the space of between 72 and 75 is not enough to label all the nodes of the new section. But if we treat the new section data as one tree node, the number 74 (or 73) can be the label of the section subtree. In such a case the subtree is defined as Nested Tree. The labels of the section, title and New nodes in the Nested Tree can be marked by new numbering regardless of the numbering of the original tree, and the number 74 represents the position of the Nested Tree in the original XML data tree.

We now define some terminologies and then prove a theorem that is necessary to define the rule for the determination of ancestor–descendant or parent–child structural relationships between tree nodes which are in the Nested Tree structure.

Definition 1. A Nested Tree is a subtree which has an interval-based number as a node of the containing tree and its own interval-based numbering as a tree.

Definition 2. A k-Nested Tree ($k = 1, 2, \ldots$) is defined as follows:

(1) 1-Nested Tree is a Nested Tree of XML data tree which is not included by any other Nested Trees.
(2) m-Nested Tree, T_m, is a Nested Tree that is included by $(m-1)$-Nested Tree, T_{m-1}, $(m = 2, 3, \ldots)$ and there is not any other Nested Tree that includes T_m and is included by T_{m-1}.

Definition 3. The Nested Tree of node N is the Nested Tree that includes the node N. The last Nested Tree of node N is the n-Nested Tree where there are n Nested Trees of node N.

Definition 4. The startList of any tree node N is the list, $s_1, \ldots, s_n; s_{n+1}$, where the last Nested Tree T of N is an n-Nested Tree, where s_i is the label of the i-Nested Tree of the node $N (i = 1, 2, \ldots, n)$ and s_{n+1} is the start position of N in the n-Nested Tree T. The endList of node N is defined in the same way of the previous definition of startList of N except that the start position is substituted by the end position of N.

To apply the Nested Tree structure to the interval-based labeling scheme, the label of each node can be represented as the 4-tuple (`DocID`, `sList`, `eList`, `Level`), where (i) `DocID` is the identifier of the document; (ii) `sList` and `eList` is the startList and endList of the node, respectively; and (iii) `Level` is the depth of the node in the data tree. We call the inverted list that is composed of the lists of the above 4-tuples representation, Extended Inverted List.

For example, in Fig. 1, the inserted section node is represented as 4-tuple, $(1, 74; 1, 74; 12, 3)$, assuming that the `DocID` of every node in the tree is 1. The label of the first book node is (1, 7, 33, 2). Fig. 2 shows 4-tuple representations of tree nodes except DocID and Level.

Lemma 1. *For the labels of two nodes N_1 and N_2, $(D_1, s_1; s_2; \ldots; s_m, e_1; e_2; \ldots; e_m, L_1)$ for N_1 and $(D_2, t_1; t_2; \ldots; t_n, f_1; f_2; \ldots; f_n, L_2)$ for N_2, if $s_i = t_i$, for each i, $1 \leqslant i \leqslant k-1$ $(k \leqslant min(m,n))$, then $s_i = e_i$ and $t_i = f_i$, for each i, $1 \leqslant i \leqslant k-1$.*

Proof. The Lemma is provided by the Definition 4. □

Theorem 1. *For the labels of two nodes N_1 and N_2, $(D_1, s_1; s_2; \ldots; s_m, e_1; e_2; \ldots; e_m, L_1)$ for N_1 and $(D_2, t_1; t_2; \ldots; t_n, f_1; f_2; \ldots; f_n, L_2)$ for N_2, if $s_i = t_i$ for each i, $1 \leqslant i \leqslant k-1 (k \leqslant min(m,n))$, and $s_k \neq t_k$, then N_1 is the ancestor of N_2 if and only if $s_k < t_k$ and $f_k < e_k$.*

Proof. By Lemma 1, $s_i = e_i = t_i = f_i$, for each i, $1 \leqslant i \leqslant k-1$. Therefore N_1 and N_2 are included in the same $(k-1)$-Nested Tree.

(Case 1) If $m = n = k$, in other words the last Nested Tree of N_1 and N_2 is $(k-1)$-Nested Tree, by the definition

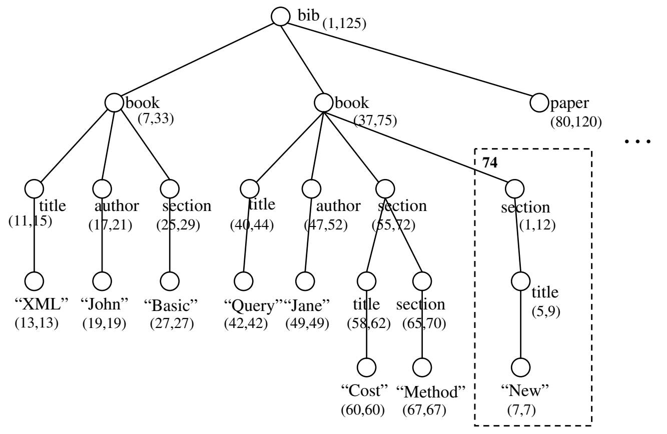

Fig. 1. An example of data inserting.

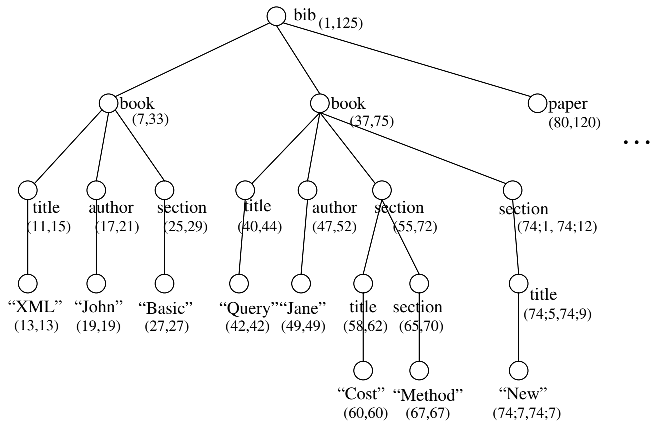

Fig. 2. An example of nested tree structure.

of startList and endList, s_k and e_k are the start position and end position of N_1 in the $(k-1)$-Nested Tree, and t_k and f_k are the start position and end position of N_2 in the same $(k-1)$-Nested Tree. Therefore N_1 is the ancestor of N_2 if and only if $s_k < t_k$ and $f_k < e_k$ from the property of the interval-based numbering.

(Case 2) If $min(m,n) > k$, N_1 and N_2 are included in k-Nested Trees, and the k-Nested Tree of N_1 is different from that of N_2. Therefore, there is not an ancestor–descendant relationship between N_1 and N_2. Since, $s_k = e_k$ and $t_k = f_k$ in this case, $s_k < t_k$ and $f_k < e_k$ is false. Consequently, the theorem holds for this case.

(Case 3) If $min(m,n) = k$(suppose $m = k$), in other words the last Nested Tree of N_1 is $(k-1)$-Nested Tree and that of N_2 is not, then $t_k = f_k$ and t_k is the label of the k-Nested Tree of N_2. s_k and e_k are the start position and end position of N_1 in the $(k-1)$-Nested Tree. Therefore, $s_k < t_k < e_k$ if and only if the k-Nested Tree of N_2 is a descendant of N_1, in other words, N_2 is a descendant of N_1. □

Fig. 3 illustrates these three cases.

According to Theorem 1, ancestor–descendant or parent–child structural relationships between tree nodes whose positions are represented in the 4-tuple (`DocID`, `sList`, `eList`, `Level`) can be determined by the following method: For a tree node N_1 whose position in the XML data is encoded as $(D_1, s_1; s_2; \ldots; s_m, e_1; e_2; \ldots; e_m, L_1)$ and N_2 as $(D_2, t_1; t_2; \ldots; t_n, f_1; f_2; \ldots; f_n, L_2)$, suppose that $s_i = t_i$, for each i, $1 \leqslant i \leqslant k-1 (k \leqslant min(m,n))$ and $s_k \neq t_k$. (i) *ancestor–descendant*: N_1 is an ancestor of N_2 iff $D_1 = D_2$, $s_k < t_k$ and $f_k < e_k$; (ii) *parent–child*: N_1 is the parent of N_2 iff $D_1 = D_2$, $s_k < t_k$, $f_k < e_k$ and $L_1 + 1 = L_2$.

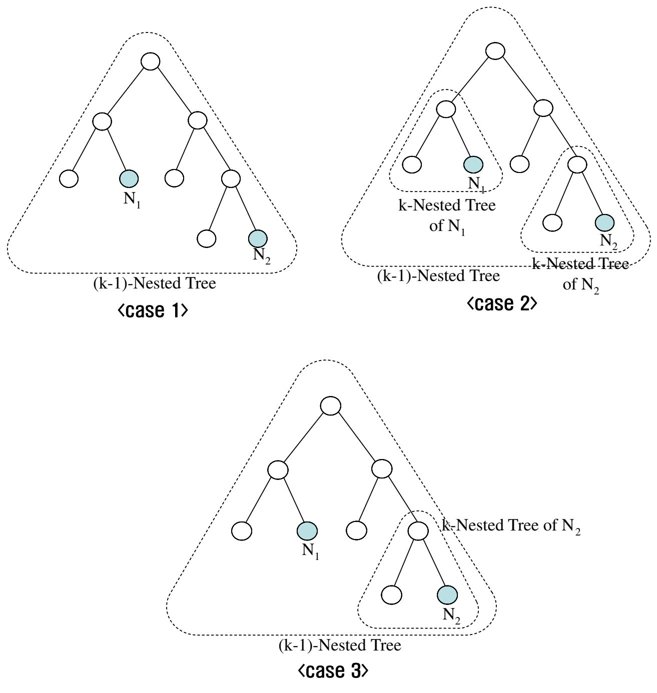

Fig. 3. Examples for the cases of theorem.

For example, in Fig. 2, the book node with label (37, 75) is an ancestor of the title node with label (74;5, 74;9), and the section node with label (74;1, 74;12) is the parent of the title node with label (74;5, 74;9).

4. Update processing

In this paper, we treat the XML data insertion as a subtree insertion into the original XML data tree and the deletion as a subtree deletion from the data tree because all patterns of XML data updates can be handled by the combination of subtree insertions and deletions. In this section, we propose the XML data insert and delete algorithms based on the Nested Tree structure.

4.1. Insert processing

The XML data insertion can be processed by adding a subtree into the original XML data tree. In the case of the interval-based labeling, if there is enough space at the position of the insertion, it is possible to label nodes in the inserted subtree with integer numbers in the range of the space. However, it is difficult to predict the pattern of data insertions and the space becomes smaller gradually after several insertions of new data. Accordingly it is necessary to solve such cases as the size of the space is zero or smaller than the size of inserted data. We propose an insert algorithm based on the Nested Tree structure to handle the above cases.

The space is the range of integers that are possible to be used as new labels for the inserted data and the size of the space is the number of integers in the range. We call the size of the space `SpaceSize` and the size of the inserted data `InsertSize`. The insert processing is classified as follows:

- `SpaceSize` > `InsertSize`: use the integers in the range of the space as labels for the inserted subtree,
- 0 < `SpaceSize` ⩽ `InsertSize`: treat the inserted subtree as a new Nested Tree and label the Nested Tree with an integer in the range of the space,
- `SpaceSize` = 0: combine the inserted subtree with the subtree rooted by the parent of the inserted subtree, treat the combined subtree as one Nested Tree and label the Nested Tree with an integer in the space.

Fig. 4 shows the example of these cases. The first case does not need a new method to process data insertions because the SpaceSize is enough to label the nodes of the new inserted subtree. In the second case, the size of the inserted subtree is larger than the size of the space. But if we treat the new inserted subtree as one Nested Tree, only one integer is needed for the label of the new Nested Tree. Accordingly if the size of the space is one or more, the relabeling for the nodes in the original data tree is not necessary for the new data insertion. The third case is the worst case. Because there is no space at the position for the data

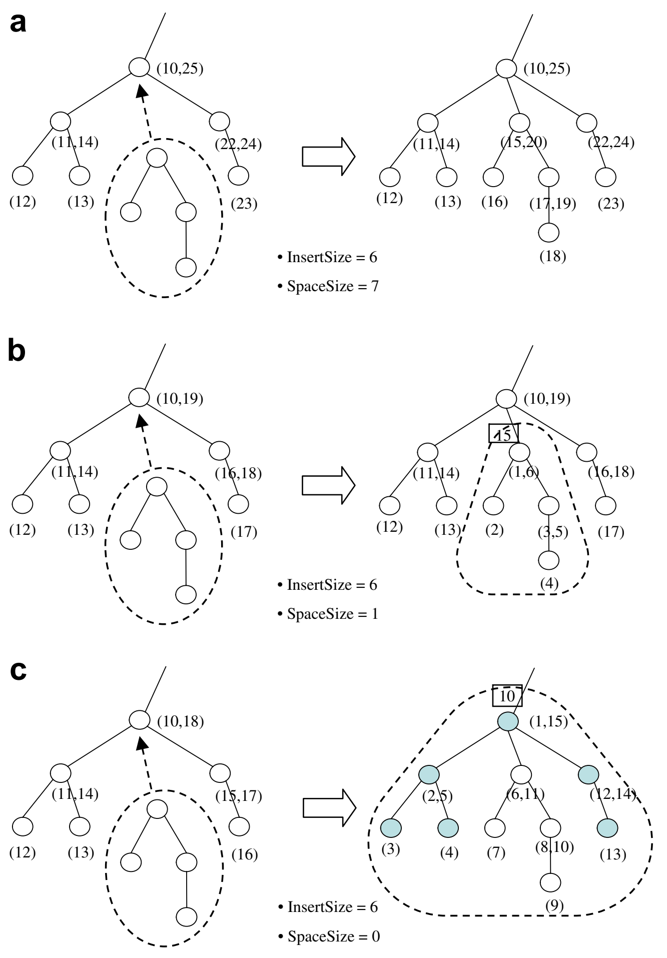

Fig. 4. Cases of insert processing.

```
procedure insert(ST, position)
//ST is the inserted subtree,
//position is the (start position, end position) pair
of the node under which the subtree is inserted
begin
1.   foreach node n of ST do
2.      Initialize the startList and endList of n to be
        the startList and endList of the current Nested Tree

3.   SpaceSize ← getSpaceSize(position)
4.   InsertSize ← getSubtreeSize(ST)
5.   $< l_1, l_2, ..., l_{SpaceSize} >$ ← getNewLabel(position)

6.   if SpaceSize > InsertSize then
7.      Label the nodes in ST by interval-based labeling scheme starting from $l_2$
8.      Attach the start and end positions to the startList and endList
9.      of the nodes in ST

10.  else if $0 <$ SpaceSize $\leq$ InsertSize then
11.     k ← $l_{\lceil \frac{SpaceSize}{2} \rceil}$ // the label of the new Nested Tree
12.     foreach node n in ST do
13.        Attach k to the startList and endList of n
14.     Label the nodes in ST by a new numbering
15.     Attach the start and end positions to the startList and endList
        of the nodes in ST

16.  else //SpaceSize = 0
17.     PST ← subtree rooted by the node that ST will be attached to
18.     foreach node n in PST do
19.        Remove the last start and end position
           from the startList and endList of n
20.     NST ← PST combined with ST
21.     NSpaceSize ← getSpaceSize(position of root of PST)
22.     $< l_1, l_2, ..., l_{NSpaceSize} >$ ← getNewLabel(position of root of PST)
23.     k ← $l_{\lceil \frac{NSpaceSize}{2} \rceil}$ // the label of NST
24.     foreach node n in NST do
25.        Attach k to the startList and endList of n
26.     Label the nodes in NST by a new numbering
27.     Attach the start and end positions to the startList and endList
        of the nodes in NST
end
```

Fig. 5. Insert procedure.

insertion, we cannot treat the new inserted subtree as one Nested Tree. However, in this case, we can extend the scope of the new Nested Tree such that the Nested Tree includes the subtree rooted by the parent of the inserted subtree. In this case, it is required to relabel some nodes in the original data tree.

Fig. 5 shows the Insert Algorithm. In line 1–2, initialize the startList and endList of each node of the inserted subtree. In line 3, get the size of the empty space and in line 4, get the size of the inserted subtree. The first case of insert processing is in line 6–9, the second case is in line 10–15 and the third case is in line 16–27. In the first case, SpaceSize is larger than InsertSize, the procedure uses the integers in the range of the space as labels for the inserted subtree. In the second case, a new Nested Tree is created by the inserted subtree and the third case makes a new Nested Tree by combining the inserted subtree with the parent subtree.

When the SpaceSize is larger than the InsertSize, we can choose one from two options, embedding the inserted tree using integers in space or creating a new Nested Tree. The two options provide a tradeoff between label size and space size. The first option leaves less space for later insertions and the second one makes labels longer. When data insertions occur frequently, the second option is better and eliminates node relabelings of later data insertions. But the query processing time can be higher because of the length of labels. As the Nested Tree insertions occur, the lengths of the startList and endList of nodes increase and the number of integer comparisons for processing a structural join increases also. However as shown in experimental results, the nested depth of Nested Trees does not affect the performance of the structural join significantly and also we can release Nested Trees in delete processing or during the XML database maintenance time. On the other side, when later queries access the inserted data frequently, the first one is better.

When the insertion of subtree occurs inside of the Nested Tree which is made from the previous subtree insertion and SpaceSize is less than InsertSize but more than one, the worst case for the label size happens. In this case, for each insertion, a new Nested Tree is created. Therefore the lengths of the startList and endList of the inserted node are increased by one for each insertion. The label size in the worst case is proportional to the number of insertions.

4.2. Delete processing

The XML data deletion can be processed by the deletion of a subtree from the original XML data tree. In the case of the interval-based labeling, the data deletion does not require any processing except the subtree deletion because the rule of the interval-based labeling is not broken after deleting some nodes. However, the more subtree insertions occur, the more Nested Trees are created. The more Nested Trees are created, the longer the lengths of the startList and endList of nodes are. As the lengths of the startList and endList of the nodes increase, the number of integer comparisons for the processing of a structural join increases also. So it is necessary to release Nested Trees in the process of subtree deletion as much as possible. In this subsection, we propose a delete algorithm which releases Nested Trees during data deletions.

The way to release a Nested Tree by using the space made by subtree deletions is classified by two cases. The first case is to release the last Nested Tree in which the deleted subtree is included and the second case is to release following-sibling or preceding-sibling Nested Trees of the deleted subtree recursively. If the deleted subtree is not included by any Nested Tree, the former is not applicable. In Fig. 6, the `PositionSize` is the size of the space in which the Nested Tree is included, and the `RemainTreeSize` is the size of the Nested Tree, both after delete processing. As the cases in Fig. 6, in order to release a Nested Tree, `PositionSize` must be larger than `RemainTreeSize`.

Fig. 7 is the delete procedure for these processes. The delete procedure is divided into two cases. The first is that the deleted subtree is included in a Nested Tree and the second is not. Line 1–8 is the procedure for the first case and line 9–25 is for the second case. In the first case, when PositionSize is larger than RemainTreeSize, line 5 to 8 can be applied and the Nested Tree that includes the deleted subtree is released by the procedure. When the deleted subtree is not included any Nested Tree, the line 10–25 can be applied.

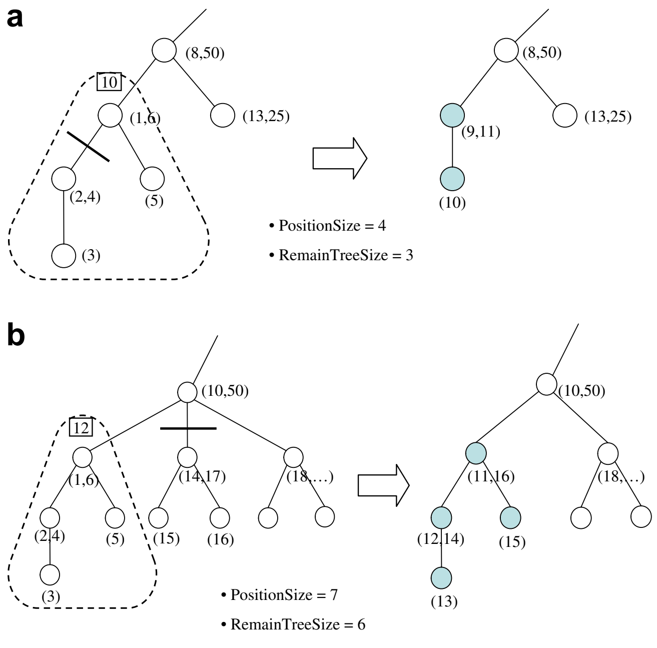

Fig. 6. Cases of delete processing.

```
procedure delete(ST)
//ST is the subtree to be deleted
begin
1.  if ST is included in any k-Nested Tree then
2.     T ← k-Nested Tree in which ST is included
3.     PositionSize ← getPositionSize(T, ST)
4.     RemainTreeSize ← getRemainTreeSize(T, ST)

5.     if PositionSize > RemainTreeSize then
6.        Delete the nodes in ST
7.        Release T
8.        Relabel the nodes remaining in T

9.  else //ST is not included in any Nested Tree
10.    if the following-sibling(or preceding-sibling)
          of ST is Nested Tree then
11.       NST ← following-sibling
             (or preceding-sibling) of ST
12.       PositionSize ← getPositionSize(NST + ST, ST)
13.       RemainTreeSize ← getSize(NST)
14.       if PositionSize > RemainTreeSize then
15.          while following-sibling(or preceding-sibling)
                of NST is Nested Tree(NNST) and the size
                of NNST is smaller than (PositionSize-RemainSize) then
16.             NST ← NST + NNST
17.             PositionSize ← getPositionSize(NST+ST, ST)
18.             RemainSize ← getSize(NST)
19.          Delete the nodes in ST
20.          Release NST
21.          Relabel the nodes in NST
22.       else
23.          Delete the nodes in ST
24.    else
25.          Delete the nodes in ST
end
```

Fig. 7. Delete procedure.

Because recursively releasing the last Nested Tree is not possible, our algorithm applies the recursive release procedure to only the second case, releasing sibling Nested Trees. If releasing the $(n-1)$-Nested Tree of the deleted subtree could be possible, where the last Nested Tree is n-Nested Tree, it would have been released before the releasing last nested Tree. After releasing the last Nested Tree (n-Nested Tree), the number of nodes in the $(n-1)$-Nested Tree is bigger than that before the release. On the other hand, recursively releasing sibling Nested Tree is possible if enough space is added after deleting subtree.

The delete procedure prefers releasing the last nested tree to releasing siblings because releasing the last nested tree releases more nodes compared with releasing siblings. The last nested tree containing the deleted subtree includes any sibling nested trees of the deleted subtree.

The performance of this delete procedure is not better than that of the simple deletion procedure. However because the performance of the processing of the future queries is influenced by whether the Nested Tree is released or not, this delete procedure is very important.

4.3. Analysis of update processing

It is the number of relabeled nodes that determines the performance of the update procedure. In the case of the insert procedure, node relabeling is required when there is no space at the position for the data insertion. The number of relabeled nodes is the size of the subtree that is rooted by the parent of the inserted subtree. In the case of delete procedure, node relabeling is necessary when the nearest sibling of a deleted subtree is a Nested Tree and it is possible to release the Nested Tree by using the space deleted. The number of relabeled nodes is the size of the Nested Tree.

The worst case of the insert procedure is that the position for the data insertion is under the root of the original data tree and there is no space for the insertion. In this case, it is required to relabel the whole nodes of the original data tree. If d is the depth of the data tree and f is the maximal fan-out of the data tree, the number of relabeled nodes is $\sum_{i=0}^{d} f^i$ in the worst case.

If for each node in the data tree the probability to be chosen as the parent node of an insertion is equal, the average number of relabeled nodes for a data insertion is $O(d)$.

Theorem 2. *Suppose that for each node in the data tree the probability to be chosen as the parent node of insertion is equal. If the depth of the data tree is d and the maximal fan-out is f with $f > 1$, the average number of relabeled nodes for a subtree insertion is $O(d)$.*

Proof. Let N be the average number of nodes to be relabeled. Then N is calculated by the following equation.

$$N = \frac{\sum \#\text{ of nodes in each subtree}}{\#\text{ of nodes in the tree}}$$

Then

$$N = \frac{\sum_{i=0}^{d}(i+1)\cdot f^i}{\sum_{i=0}^{d} f^i} = \frac{(d+1)\cdot f^{d+1}}{f^{d+1}-1} - \frac{1}{f-1} < \frac{(d+1)\cdot f^{d+1}}{f^{d+1}-1}$$

Because $\frac{f^{d+1}}{f^{d+1}-1} < 2$, N is $O(d)$. □

Considering the fact that XML data trees with huge numbers of nodes have relatively small numbers of depths, from Theorem 2, insert processing based on the Nested Tree is efficient.

5. Query processing

Under the proposed Nested Tree structure, the existing structural join algorithms based on the interval-based labeling scheme can be used to determine structural relationships between two elements, if the integer comparison is changed to the integer list comparison. Therefore we can adopt the advantages of the existing structural join algorithms under the Nested Tree structure. Also by using the features of the Nested Tree structure we can extend the inverted list to the Nested Inverted List, and the structural join algorithm using the Nested Inverted List improves the performance of the structural join processing. In this section, we present the query processing using existing algorithms and the query processing using Nested Inverted List.

5.1. Query processing using traditional algorithms

The structural join processing using the existing algorithms (i.e. Stack-Tree-Desc (Srivastava et al., 2002), Anc-Desc-B+ (Chien et al., 2002) or XR-tree (Jiang et al., 2003)) can provide the result in time proportional to the input and output size. The proposed labeling method using the Nested Tree structure is an enhanced interval-based labeling scheme that overcomes the weakness of the existing interval-based labeling scheme in update processing. Therefore, we can use all the existing algorithms to process the structural joins in the Nested Tree structure.

The position of a node in the Nested Tree structure is represented by the startList and endList, defined in Section 3.2. The startList and endList are lists of integers. If the integer comparison operation is changed to the integer list comparison operation given in Fig. 8, the existing structural join algorithms can be used under the Nested Tree structure.

5.2. Query processing using nested inverted list

The structural join processing using the existing algorithms described in the above section must access all the nodes in the result of a structural join. For example, in Fig. 9a, to get the result of structural join $A//B$, the node a_2 in the element A list must be compared with nodes b_1, b_2, b_3 and b_4 in the element B list. The result consists of four pairs (a_2,b_1), (a_2,b_2), (a_2,b_3) and (a_2,b_4). However the nodes b_1, b_2, b_3 and b_4 are represented by the label of the Nested Tree, that is 6, and the label of node a_2 includes the label 6. Therefore we can make the four result pairs with only one comparison of node a_2 with node b in Fig. 9b.

In order to use this feature of the Nested Tree structure we define the Nested Inverted List to represent the list of nodes. Fig. 9 illustrates the structure of the Nested Inverted List. The Nested Inverted List is the list of Nodes, in which the Node is either the pair of start and end or the label of a Nested Tree and the link to a child list. If there is no Nested Tree in XML data, the Nested Inverted List is the same as the traditional inverted list. But as the Nested Trees are nested deeply, the Nested Inverted List is nested deeply also. The Nested Inverted List can reduce the space to store the inverted list because the repeated labels of Nested Trees are stored by one label and link to a child list.

```
procedure int comparison($a_1; a_2; ...; a_m$, $b_1; b_2; ...; b_n$)
//$a_i$ and $b_i$ are integers
//$a_1; a_2; ...; a_m$ and $b_1; b_2; ...; b_n$ are lists of integers
begin
1.  $i \leftarrow 1$
2.  while($a_i = b_i$ and $i < m$ and $i < n$)
3.      $i \leftarrow i + 1$
4.  if($i \geq m$ or $i \geq n$) then return 0
5.  if($a_i < b_i$) then return -1
6.  else if($a_i > b_i$) then return 1
end
```

Fig. 8. Integer list comparison operation.

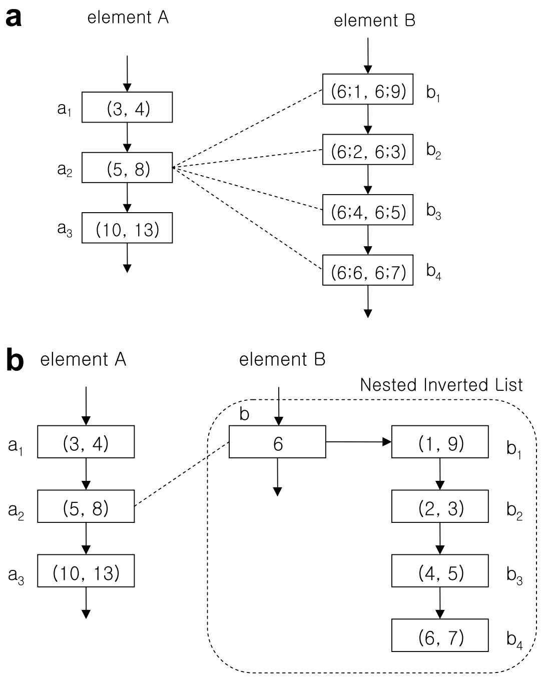

Fig. 9. Structural join.

The structural join processing based on the Nested Inverted List structure can also use the existing structural join algorithms (Srivastava et al., 2002; Chien et al., 2002; Jiang et al., 2003). Fig. 10 shows the structural join algorithm, Nested-Stack-Tree-Desc, using the Nested Inverted List and the Stack-Tree-Desc (Srivastava et al., 2002) algorithm. The procedure is similar to the existing algorithm Stack-Tree-Desc, except that it includes the case that the label of AList is the same as that of BList. In such case, the Nested-Stack-Tree-Desc is called recursively and the parameters are substituted by child lists. The line 10 to 11 shows this case.

```
procedure Nested-Stack-Tree-Desc(AList, DList)
begin
1.   a ← AList.firstNode
2.   b ← DList.firstNode
3.   output ← null
4.   while the input lists are not empty
       or the stack is not empty do
5.     if a.startPos > stack.top.endPos
         and d.startPos > stack.top.endPos then
6.       stack.pop()
7.     else if a.startPos < d.startPos then
8.       stack.push(a)
9.       Let a ← a.nextNode
10.    else if a.startPos = d.startPos then
11.      Nested-Stack-Tree-Desc(a.child, d.child)
12.    else
13.      foreach node a1 in stack do
14.        append (a1, d) to output
15.      d ← d.nextNode
end
```

Fig. 10. Nested query procedure.

5.3. Query processing using query workload

A Nested Tree can be created after an XML data insertion. However, if we know the query workload, we can make Nested Trees during the initial load in order to improve the efficiency of query processing. For example, for the query $A//B$, when C is an ancestor of B and a descendent of A, and there are several B elements below an element C, we can make the Nested Tree rooted by C and several B elements can be treated by one child list in the Nested Inverted List and then the performance of the query $A//B$ can be improved. In this case we call C the middle-element of the query $A//B$. Of course, there can be many such C's in the XML data tree.

We can consider several node labeling methods based on the Nested Tree structure by analyzing a given query workload. The simplest way is to make Nested Trees which are rooted by middle-elements of a query in the workload. Ultimately there are Nested Trees as many as middle-elements of queries in query workload. However, if there are thousands of queries, this way is not good for an efficient query processing, and it is better to make Nested Trees based on frequently used queries.

Next we should consider whether there is an ancestor–descendant structural relationship between two different middle-elements. In such a case, if we create Nested Trees each of which is rooted by a middle-element, the depth of Nested Trees will increase. Accordingly the length of startList and endList will become longer, which makes the query processing inefficient. So it is necessary to make the Nested Trees without increasing their depths.

We propose a method for node labeling using given query workload. The method is based on the set of frequently used queries. Table 1 shows the set of frequently used queries and the middle-element for each query. The Ratio shows the relative frequency of the query and the sum of all ratios is 100.

Table 2 is made from the middle-elements in Table 1 removing the duplicated middle-elements. The Ratio in Table 2 is the sum of all ratios of the duplicated middle-elements in Table 1.

We can partition the middle-elements in Table 2 such that there is an ancestor–descendant structural relationship between two elements in the same partition. For each partition, we assign a weight to each element in the partition such that the sum of weights is 100.

The performance of each query is determined by the weight of the corresponding middle-element. The bigger

Table 1
Frequently used queries

Query	Ratio (%)	Middle-element
$a_1//d_1$	t_1	c_1
$a_2//d_2$	t_2	c_2
…	…	…
$a_n//d_n$	t_n	c_n

Table 2
Ratio of middle-element

Middle-element	Ratio (%)
s_1	p_1
s_2	p_2
...	...
s_k	p_k

the weight of the middle-element is, the better the performance of the query is. We consider two methods to assign weights to middle-elements in the same partition: (1) Assign the same weight to each middle-element and (2) Assign the weight to each middle-element proportional to the ratio of the middle-element in Table 2.

Let the partition be represented by $(e_{11}, e_{12}, \ldots, e_{1l_1})$, $(e_{21}, e_{22}, \ldots, e_{2l_2}), \ldots, (e_{m1}, e_{m2}, \ldots, e_{ml_m})$ with $l_1 + l_2 + \cdots + l_m = k$. Then the weight of each element is calculated as follows when p_{ij} is the p_k of e_{ij}:

$$\text{Weight of} e_{ij} = \begin{cases} 100 \times 1/l_i & \text{for case1} \\ 100 \times p_{ij}/(p_{i1} + p_{i2} + \ldots + p_{il_i}) & \text{for case2} \end{cases}$$

After the weights of elements in the partition are determined by the above method, we can label nodes based on the given weights. For example, element A and B are in the same partition and the weights are 80 and 20, respectively. Then A and B are in the same path and we make the Nested Trees rooted by A for 80% of the paths and rooted by B for 20%.

6. Experimental results

Our experiments were carried out on an Intel Pentium 1.7Ghz with 1GB memory, running Windows XP. All procedures are implemented in Java. All experiments were repeated 10 times independently and the average processing time was calculated disregarding the maximum and minimum values.

The experiments described in this section use three sets of test data. The three data sets are Shakespeare (Bosak), XMark (XMark, 2001) and Nasa (nasa). The XMark data set contains information about auctions and it is a synthetic benchmark data set generated by the XML Generator from XMark. The Shakespeare data set represents Shakespeare's plays in XML format. The Nasa data set is generated from the legacy flat file format of astronomical data. Characteristics of these data sets are summarized in Table 3.

Table 3
Experimental data

	Shakespeare	XMark	Nasa
Size	7.7 MB	115.7 MB	25.2 MB
Nodes	179,619	1,666,315	476,646
Depth	7	12	8

6.1. Update processing

In order to analyze the efficiency of the update processing based on the Nested Tree structure, we measure the performance of XML data insertions and deletions based on the traditional interval-based labeling, prime labeling, lazy approach, ORDPATH and our approach, the Nested Tree structure, that we call Interval, Prime, Lazy, ORDPATH and Nested, respectively.

We use the XMark data with various sizes as the original data, and assume that there is space more than one but less than the number of nodes in the inserted data. We measure the processing time of data insertions as we increase the size of the original data. There are 382 nodes in the inserted data. The experimental result is shown in Fig. 11. We can see that the data insertion time of Interval increases proportionally as the size of the inserted XML data increases. In the Interval approach, the relabeling of nodes in the original data tree is inevitable when new data is inserted, and the number of nodes to be relabeled increases as the size of the original data increases. In the Prime, Lazy, ORDPATH and Nested approach, the relabeling of nodes in the original data tree is not necessary and only the labeling of nodes in the inserted data is needed. Therefore the size of the original data cannot affect the time of data insertions. However in the Prime approach the label of a node is determined by the product of the self-label and the label of the parent node, so the time of data insertion exceeds these of Lazy, ORDPATH and Nested approaches. In the Lazy approach, the relabeling of inserted data is not necessary, but labels of segments are updated in processing XML data insertions. Therefore the time of data insertions of Lazy is similar to that of Nested, but the average time for Lazy is 251 ms while that for Nested is 216 ms. In the ORDPATH approach, the dewey-similar labels are converted to binary representations. Therefore the average time of data insertions of ORDPATH is 245 ms that is worse than the Nested approach. In the Nested approach, a new label is assigned to the inserted Nested Tree, and then the label of each node in the inserted Nested Tree is assigned by the sequential assignment of new integers during the depth first traversal

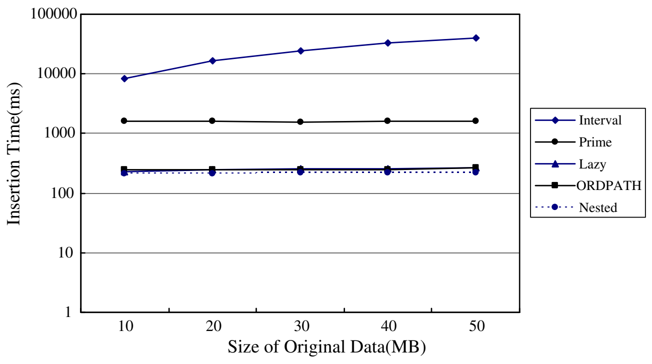

Fig. 11. Insert processing as increasing the size of original data.

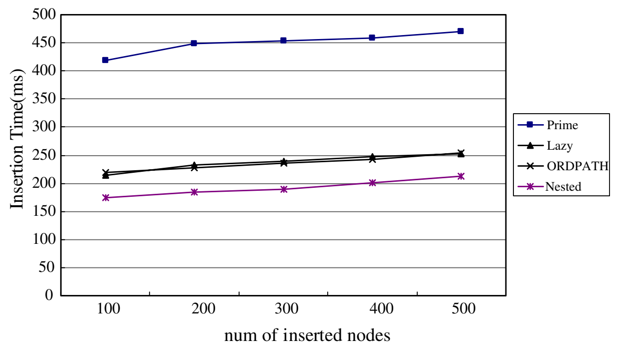

Fig. 12. Insert processing as increasing the number of inserted nodes.

of the inserted data. Therefore in the Nested approach, the data insertion is processed by a simple integer assignment to each node, so the performance is the best.

In order to see the effect of the size of inserted data in insert processing, we measure the processing time of data insertions as we increase the number of inserted nodes. The size of original data is 20 MB. The experimental result is shown in Fig. 12. The result of Interval is about 16,000 ms, which cannot be included in the figure. We can see that the data insertion times of Interval, Prime, Lazy, ORDPATH and Nested increase proportionally as the number of inserted nodes increases. In this case, our approach has also best performance.

To see the performance of XML data deletions, we used 20 MB XMark data as the original data and measure the processing time of data deletions as increasing the number of deleted nodes. In the case of Nested, we used the delete algorithm including Nested Tree releasing processing. The experimental result is shown in Fig. 13. The result of Nested is somewhat slower than others, but in the case of no releasing, the deletion time of Nested is close to Interval (see Fig. 14).

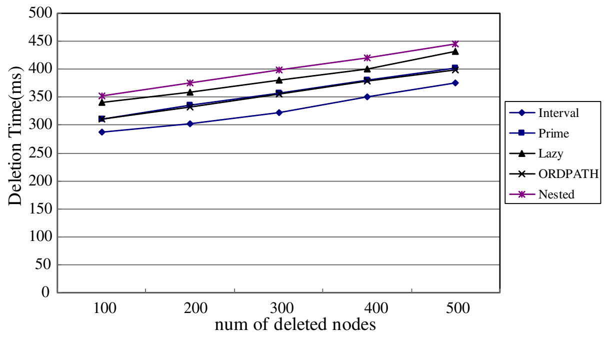

Fig. 13. Delete processing as increasing the number of deleted nodes.

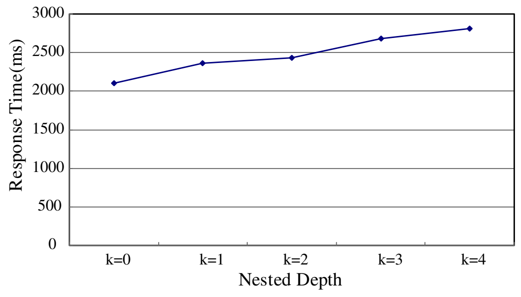

Fig. 14. Structural join processing with increasing nested depth.

6.2. Structural join processing

Query Sets. We use several structural join patterns against the three data sets to analyze the query performance. All the structural joins use ancestor–descendant relationships because the parent–child relationships can be simply determined by whether the level of a child is one larger than that of a parent. Table 4 shows the information about experimental queries. For the Shakespeare data set, we use seven queries from Q1 to Q7 used in (Wu et al., 2004), and for the XMark data set, we use seven queries from Q8 to Q14 a part of which is used in (Catania et al., 2005). For the Nasa data set, we use five queries from Q15 to Q19. In Table 4, the Results shows the number of pairs in query results for 7.7 MB Shakespeare data, 115.7 MB XMark data and 25.2 MB Nasa data.

Performance. As described in Section 5, to process structural joins under the Nested Tree structure, we can consider two cases. The first case is to extend the traditional inverted list into the Extended Inverted List that represents the start and end positions of a node as integer lists. Because this approach takes the same method as the existing algorithms, it is not able to improve the query performance. But it supports the update processing efficiently and query processing with reasonable performance. It is the depth of the nested tree containing the nodes in the lists (this depth will be called the nested depth of a node subsequently), which are the list of A and the list of B under the structural join $A//B$, that dominates the performance of

Table 4
Query set

Data set	Query	XPath expression	Results
Shakespeare	Q1	play//act	185
	Q2	speech//line	107,991
	Q3	play//persona	969
	Q4	act//speech	30,937
	Q5	act//line	107,405
	Q6	play//speech	31,014
	Q7	play//line	107,791
XMark	Q8	person//phone	12679
	Q9	profile//interest	37,689
	Q10	person//watch	50,269
	Q11	person//interest	37,689
	Q12	regions//incategory	82,151
	Q13	category//listitem	1267
	Q14	open-auctions//bidder	59,486
Nasa	Q15	dataset//reference	2435
	Q16	dataset//author	9766
	Q17	tableHead//name	60,663
	Q18	dataset//field	60,663
	Q19	dataset//name	71,688

structural join processing. The Fig. 12 shows the time to process Q8 for the XMark data set as we increase the nested depth k of nodes in the lists of person and phone. $k = 0$ means that there is no data updates and so there is not Nested Tree in the data tree. This performance is the same as that of the traditional structural join processing. For $k = 1$, the nested depth of each node in the lists is one, that means each node is included in one Nested Tree, and the Nested Trees are different. For $k = 2$, the nested depth of each node is two and for $k = 3$, three. In the case of $k = 4$, the nested depth of each node in the list of person is three while that in the list of phone is four because the depth of person is three while the depth of phone is four in the data tree. It is natural that the processing time increases as k increases. However the increasing rate is small enough not to degrade the performance of structural join processing.

The second case is to construct the Nested Inverted List that represents the position of each node as a nested list described in Section 5.2 and this case avoids unnecessary integer comparison operations.

To analyze the effect of the Nested Inverted List, we measure the performance of structural join processing for XMark data with 115.7 MB. Initially we load 50% of the data and then perform the insert processing of the rest of the data. The result of structural join processing in this approach is shown in Fig. 15. The performance of our approach for from Q10 to Q14 is better than those of other approaches. Those queries are to find the ancestor–descendant structural relationships, so they can take advantage of the Nested Inverted List. However, for the parent–child structural relationship, like Q8 and Q9, the performance of the ORDPATH is better than our approach.

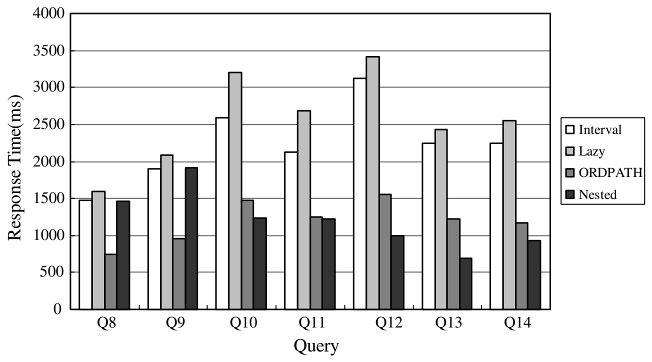

Fig. 15. Structural join processing for the XMark data (115.7 MB).

Secondly we measure the performance of structural join processing using the approach described in Section 5.3. During the initial data load, we label tree nodes as we make Nested Trees using the query workload. We assume that the queries given in Table 4 are frequently used queries. First we perform the structural join processing using the case 1 in Section 5.3, which assigns the same ratio to each middle-element in the same partition.

Figs. 15–18 show the results of structural join processing for Shakespeare, XMark and Nasa data. For the Shakespeare data with 2.8 MB, we compare our approach with

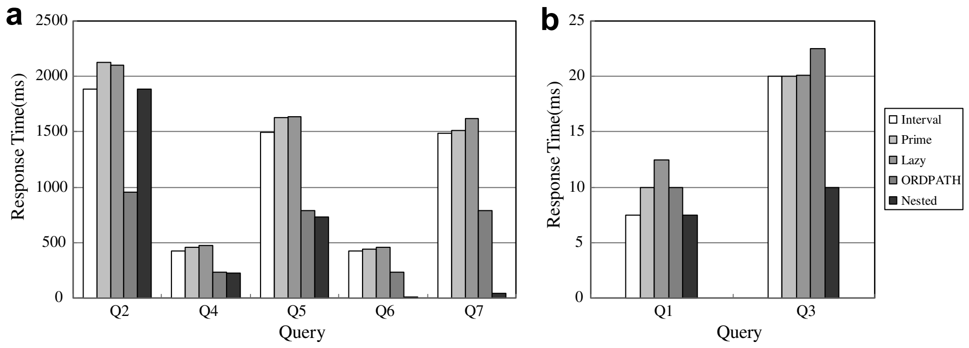

Fig. 16. Structural join processing for the Shakespeare data (2.8 MB).

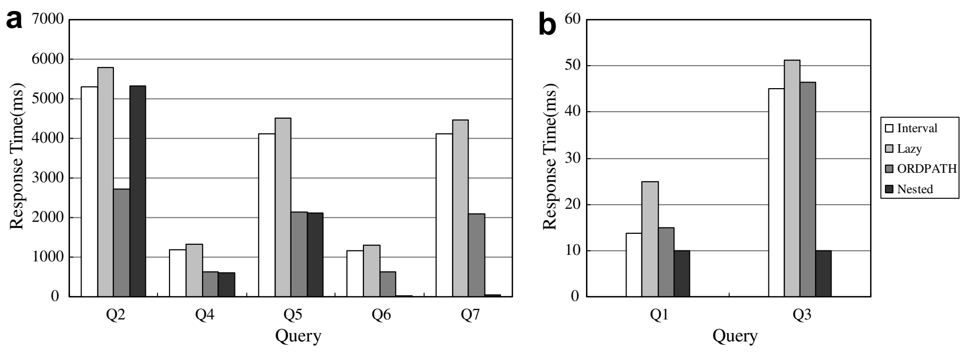

Fig. 17. Structural join processing for the Shakespeare data (7.7 MB).

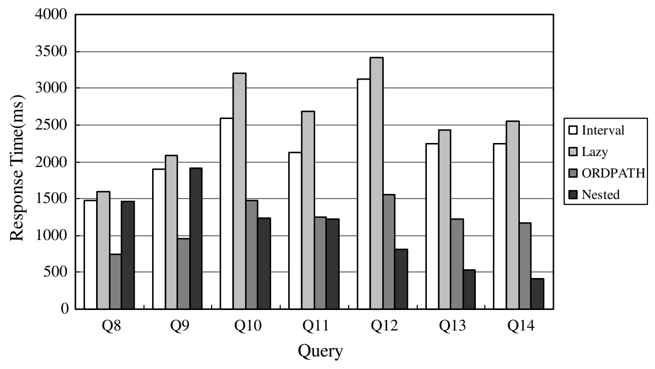

Fig. 18. Structural join processing for the XMark data (115.7 MB).

four approaches, Interval, Prime, Lazy and ORDPATH. The size of Shakespeare data must be less than 2.8 MB in order for Prime to process structural joins given in Table 4. For comparing with the Lazy approach, we use the three data sets and construct 50 segments for the Lazy approach. For the Shakespeare data with 7.7 MB, the XMark data with 115.7 MB and the Nasa data with 25.2 MB, we compare our approach with the Interval, Lazy and ORDPATH approach because the data set is too large for Prime to process (see Fig. 19).

Mostly the performance of our approach, the Nested approach, is better than those of other approaches. However, for Q2, Q8, Q9 and Q15, the performance of the Nested is similar to that of Interval, better than those of the Prime and Lazy, but worse than that of the ORDPATH. These queries have the parent–child relationship, so they cannot take advantage of the Nested Inverted List.

Particularly the performance of the Shakespeare data set shows huge improvements. In the Shakespeare data set, some elements with the same name are repeated under one parent, and this feature improves the performance of structural join processing.

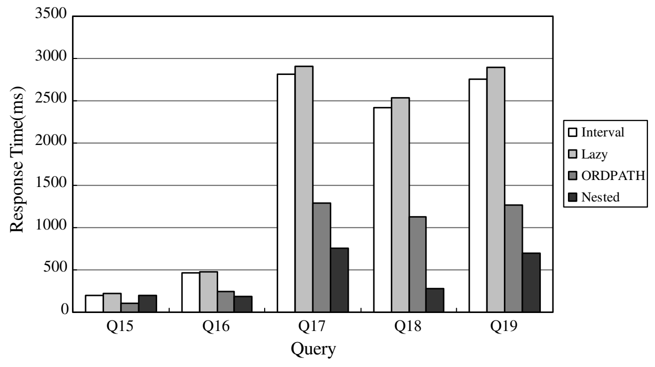

Fig. 19. Structural join processing for the Nasa data (25.2 MB).

Table 5 shows the performance comparisons. The average number is generated by dividing the average processing time of another approach by that of our approach. In update processing, our approach is 113.0 times faster than Interval on the average and 182.7 times faster maximally. Other numbers can be similarly interpreted.

7. Conclusions

In this paper we propose the Nested Tree structure to provide efficient XML update processing and query processing. The Nested Tree structure solves the weak points and takes advantages of previous node labeling schemes. It can support the dynamic interval-based labeling scheme, which supports XML data updates efficiently. We present XML data insert and delete processing with almost no node relabeling using the proposed Nested Tree structure. Also we show the traditional structural join algorithm can be applied to our model just with the change of the comparison operation between two labels, and we develop an enhanced structural join algorithm using the proposed Nested Inverted List. In our experiments, the performance of XML data insert processing of our approach is better than those of Prime, Lazy and ORDPATH. Also the performance of structural join processing can be upgraded by our proposed structure. Experimental results show that our approach is significantly better than the Lazy and ORDPATH, which are the recent approaches, for query processing while slightly better for update processing.

Acknowledgement

This research was supported by the Ministry of Information and Communication, Korea, under the College Information Technology Research Center Support Program, grant number IITA-2006-C1090-0603-0031.

References

Bosak, J. Shakespeare, <http://www.ibiblio.org/xml/examples/shakespeare/>.
Catania, B., Ooi, B.C., Wang, W., Wang, X., 2005. Lazy xml updates: laziness as a virtue of update and structural join efficiency. In: Proceedings of the ACM SIGMOD 2005.
Chamberlin, D. et al., 2001. XQuery 1.0: An XML query language, W3C Working Draft.

Table 5
Performance comparison

	Interval		Prime		Lazy		ORDPATH	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
Update	113.0	182.7	7.3	7.5	1.2	1.2	1.1	1.2
Join	11.4	102.1	12.9	44.1	12.6	111.1	6.2	52.1

Chen, Y., Davidson, S.B., Zheng, Y., 2004. BLAS: An efficient XPath processing system. In: Proceedings of the ACM SIGMOD 2004, pp. 47–58.

Chien, S., Vagena, Z., Zhang, D., Tsotras, V.J., Zaniolo, C., 2002. Efficient structural joins on indexed XML documents. In: Proceedings of the VLDB 2002, pp. 263–274.

Clark, J., DeRose, S., 1999. XML path language(XPath) Version 1.0, W3C Recommendation.

GSFC/NASA XML Project, Nasa, http://www.cs.washington.edu/research/xmldatasets/.

Jiang, H., Lu, H., Wang W., Ooi, B.C., 2003. XR-Tree: Indexing XML data for efficient structural joins. In Proceedings of the ICDE 2003, pp. 253–263.

Kaplan, H., Milo, T., Shabo, R., 2002. A comparison of labeling schemes for ancestor queries. In: Proceedings of the SODA 2002.

Li, Q., Moon, B., 2001. Indexing and querying XML data for regular path expressions. In: Proceedings of the VLDB 2001, pp. 361–370.

Lu, J., Ling, T.W., Chan, C., Chen, T., 2005. From region encoding to extended dewey: on efficient processing of XML twig pattern matching. In: Proceedings of the VLDB 2005, pp. 193–204.

ONeil, P.E. et al., 2004. ORDPATHs: insert-friendly XML node labels. In: Proceedings of the ACM SIGMOD 2004, pp. 903–908.

Silberstein, A., He, H., Yi, K., Yang, J., 2005. BOXes: efficient maintenance of order-based labeling for dynamic XML Data. In: Proceedings of the ICDE 2005.

Srivastava, D., Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Wu, Y., 2002. Structural joins: a primitive for efficient XML query pattern matching. In: Proceedings of the ICDE 2002, pp. 141–152.

Tatarinov, I., Viglas, S.D., Beyer, K., Shanmugasundaram, J., Shekita, E., Zhang, C., 2002. Storing and querying ordered XML using a relational database system. In: Proceedings of the ACM SIGMOD 2002, pp. 204–215.

Wu, X., Lee, M.L., Hsu, W., 2004. A prime number labeling scheme for dynamic ordered XML trees. In: Proceedings of the ICDE 2004, pp. 66–78.

XMark: The XML-benchmark Project, http://monetdb.cwi.nl/xml/index.html, April 2001.

Yoshikawa, M., Amagasa, T., 2001. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. ACM Transactions on Internet Technology 1 (1), 110–141.

Zhang, C., Naughton, J., Dewitt, D., Luo, Q., Lohman, G., 2001. On supporting containment queries in relational database management systems. In: Proceedings of the ACM SIGMOD 2001, pp. 425–436.