An XML Indexing Structure with Relative Region Coordinate*

Dao Dinh Kha Masatoshi Yoshikawa Shunsuke Uemura
Graduate School of Information Science
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0101, Japan
{*kha-d, yosikawa, uemura*}*@is.aist-nara.ac.jp*

Abstract

For most index structures for XML data proposed so far, update is a problem because XML element's coordinates are expressed by absolute values. Due to the structural relationship among elements in XML documents, we have to re-compute these absolute values if the content of source data is updated. The reconstruction requires update of large portion of index files, which causes a serious problem especially when XML data content is frequently updated. In this paper, we propose an indexing structure scheme based on the Relative Region Coordinate that can effectively deal with the update problem. The main idea is that we express the coordinate of an XML element based on the region of its parent element. We present an algorithm to construct a tree-structured index in which related coordinates are stored together. In consequence, our indexing scheme requires update of only a small portion of index file in case of updating.

1. Introduction

XML (Extensive Markup Language) was adopted by World Wide Web Consortium and is a simplified subset of SGML that has been optimized for use on the World Wide Web. XML aims at the rendition of data by allowing users to describe the *arbitrary structure* of data using user-defined vocabulary. Since its invention, the language soon has proved its ability to be the basis for data interchange on the Internet. To exploit the power of XML, a database system for XML documents should support queries on both content and structure. That is it should keep structural information as well as content and relation between structure and content.

*This work is partially supported by the Ministry of Education, Culture, Sports, Science and Technology, Japan under grants 11480088, 12680417 and 12208032, and by CREST of JST (Japan Science and Technology).

To achieve the goal, many index structures for XML data have been proposed. Because these indexing structures use the absolute address to pinpoint where data resides, update causes a re-computation. We have to re-compute data coordinates stored in the index if an update occurs and changes the length of data in the source. If the update frequency is high, for example when XML is used for construction of public home pages, the cost of reconstruction is unbearable.

We investigate and discuss how to efficiently deal with the update problem that is significant in the first phase of document life cycle. We propose a method to represent the coordinate of content in XML documents, called *Relative Region Coordinate*. The idea is that we express the coordinate of an XML element in relation to its parent element coordinates. Using the Relative Region Coordinate to coordinate XML elements, we know only where the elements start and end inside the region of its parent elements. As a tradeoff, we need to update only a portion of index file in case of content updating. Further, we propose an approach to organize the XML data called Block Subtree that effectively exploits the advantages of Relative Region Coordinate method. Other alternatives of the method's implementation based on commercial database systems are also briefly presented.

In this paper, XML document is presented as a rooted tree with several node types such as Element, Attribute, Text, etc. [10] [11], but our discussion does not base on any specific type of node. We assign identification number to the nodes of XML tree and for simplicity let's assume that data updates occur in the leaves.

There are other works on storage of binary large data objects and structured data. In [1], [6] and [8] trees are used to organize the physical distribution of data. Objects are split at arbitrary byte positions to be stored in a sequence of pieces and the structural information within objects is not exploited. In [7], authors give a general schema to deal with the content and structure queries. They present two kinds of index: *Position-based Indexing* and *Path-based Indexing*. *Position-based Indexing* consists of information of

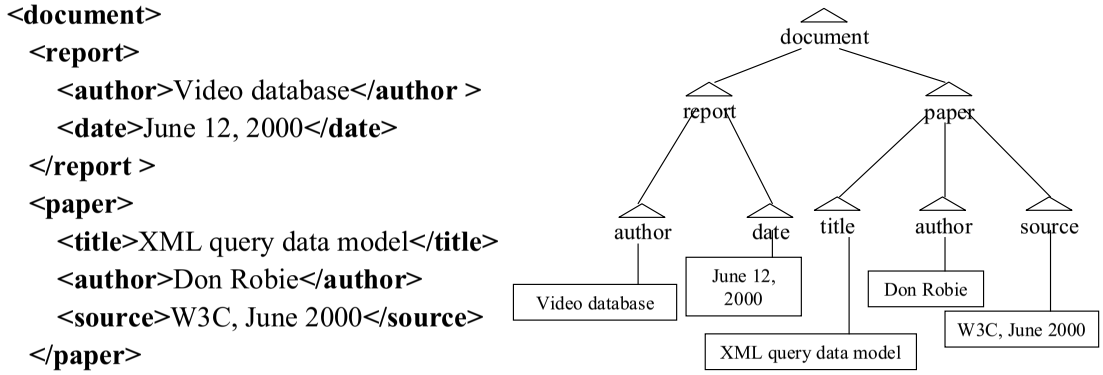

Figure 1. A simple XML document

position of words and tags within XML documents. The position is mainly represented by the absolute address of the word or tag. *Path-based Indexing* consists of encoding of all paths leading to each word. In [2], NATIX is presented as a Hybrid system that consists of a flat part with clustered grouped of nodes treated as atomic records by the lower levels using a dynamic threshold that can be adjusted. In [13], authors have proposed an indexing structure based on the Namespaces. Information of each namespace element such as the path and the position of element are recorded.

Our paper features a specific mechanism to deal with content update problem that is not adequately addressed in these works. The outline of this paper is as follows. In Section 2 we review the Absolute Region Coordinate. We present the Relative Region Coordinate in Section 3. In Section 4 we discuss the effectiveness of Relative Region Coordinate in update problem. In Section 5 we propose an index structure called *Block Subtree* suitable to Relation Region Coordinate. We briefly outline the application of Relative Region Coordinate in XML indices using Object-Relational Database in Section 6. We conclude this paper in Section 7.

2. Absolute Region Coordinate

Using region coordinate is a way to locate the content data in structured documents. Usually, we can find the content of an element in an XML document based on a pair of numbers, which point to the start and the end of the piece of text corresponding to the element. In case there are many XML documents, another number may be added to the pair to indicate which document the element belongs to. This approach uses absolute coordinate to locate elements so it is called *Absolute Region Coordinate* (ARC).

For simplicity, we consider a collection of XML documents a *big XML document* and we will omit the document number in the rest of the paper. ARC is convenient for retrieving the result of query because if we know the coordinates of the result elements we can directly access to the place where the elements reside.

Example 1. *In Figure 1, the ARC of the document's elements* **document**, **report**, **paper**, **author** *(of report)*, **date**, **title**, **author** *(of paper)*, **source** *are: [1,202], [11,84], [85,191], [19, 49], [50,75], [92,126], [127,152], and [153,183] respectively.*

Despite of the convenience, ARC has its disadvantage. When an update occurring in a leaf node causes data length of the node to be changed, absolute coordinates of the corresponding element and of all of its successive elements in the source data file must be changed too. In consequence, we usually have to compute again coordinates of many elements.

Example 2. *Suppose that we represent coordinate of elements by ARC as in Example 1, and we want to change the content of the element* **author** *of report from "Video database" to "Heijo Video database". We have to change the region coordinates of all 8 elements:* **document**, **report**, **paper**, **author**, **date**, **title**, **author** *(of paper)*, **source** *and their coordinates become: [1,208], [11,90], [91,197], [25, 55], [56,81], [98,132], [133,158], and [159,189] respectively.*

In the next section we will present another method to express region coordinate of XML elements, called *Relative Region Coordinate*. Using this method, with the same situation of the above example, we need to change region coordinates of only 5 elements.

3. Relative Region Coordinate

We present here *Relative Region Coordinate* (RRC) method, which is better than ARC in dealing with update problem. We aim at reducing the amount of disk I/O needed to update coordinate of data in index file when the source data is updated.

3.1. RRC description

To define RRC, we consider an XML tree T rooted at r. Denote n a node of the tree and $n_1, n_2, ..n_k$ are n's children.

Definition 1. (Relative Region Coordinate) *Relative Region Coordinate (RRC) of the node n of an XML tree is a pair of numbers [c_1, c_2], where c_1 (respectively c_2) is the number of bytes from the starting byte of the parent node of n to the starting byte (respectively, the end byte) of n. c_1 and c_2 are called the first and the second coordinate of n.*

In other words, RRC presents the location of a node relatively within the space of its parent. It is different from ARC, which represents the location of a node by distance to the start of the XML document, or within the space of the root node. If we compute the RRC of n's children, we have a series of pairs of numbers such as [a_1,a_2], [a_2+1,a_3],$\cdots$,[a_k+1,a_{k+1}] ($a_1 < a_2 < \cdots < a_{k+1}$). The first coordinate of the first child of a parent is not necessary to be equal to 1 because of the start tag of its parent.

Example 3. *The RRC of the document's elements* **document**, **report**, **paper**, **author** *(of report),* **date**, **title**, **author** *(of paper),* **source** *in Figure 1 are: [1,202], [11,84], [85,191], [9, 39], [40,65], [8,42], [43,68], and [69,99], respectively.*

From ARC we can easily compute the RRC of XML elements. In the other hand, ARC of a node can be computed if the RRC of all the nodes on the path leading from the root of XML tree to the considered node are available. Assume that we have to compute the ARC of the node n, the node path of n is $P \equiv n_1, n_2, \cdots, n_k = n$ and RRC of node n_i is [a_i, b_i] where i=1,2,..k then ARC of n is $[\sum_{i=1}^{k} a_i - (k-1), \sum_{i=1}^{k-1} a_i + b_k - (k-1)]$.

3.2. RRC with update problem

We call the nodes whose RRC have to be updated when the content of a leaf node n is changed RRC-updated nodes of the node n. ARC-updated nodes of a node are defined similarly. Suppose that we have to update an XML element corresponding the node n in the XML tree T. Denote $P \equiv r, n_1, n_2, ..., n_k = n$ the path from the root r of T to n.

Definition 2. (Depth of a node) *The depth of the node n is defined to be the distance between the root and n. The node depth of the root is 0. In any tree, the path between two nodes are unique, therefore the depth of a node is defined uniquely. Two nodes are called being in the same level if their node depths are equal.*

We will investigate the number of RRC-updated nodes of n. We have the following observation about the distribution of the RRC-updated nodes.

Observation 1. *In case we change the length of data in the node n with the node path P :*

- *In the same level with node n_i (i=1,2,..,k), besides n_i only the right siblings of n_i have their RRC be changed.*
- *Total number of RRC-updated for n is the sum of the numbers of RRC-updated nodes for n in each level.*

This observation is important because it hints at the way to group the information about coordinates of the XML nodes into disk blocks. We will store the nodes whose coordinates are likely to be changed at the same time into the same disk block. The idea will be realized later in Section 5.4.

Example 4. *We consider again the XML document in Figure 1. If we represent coordinate of nodes using RRC and we want to change the content of the element* **author** *of* **report** *from "Video database" to "Heijo Video database" then we have to change the region coordinates of only 5 elements:* **document**, **report**, **paper**, **author**, *and* **date** *and their coordinates become [1,208], [11,90], [91,197], [19, 55], [56,81] respectively.*

4. Theoretical evaluation of RRC effectiveness

In this section we will evaluate the effectiveness of RRC method. The criterion is the number of RRC-updated nodes. When comparing the methods ARC and RRC, we can not use the measure of disk I/O because it depends on the physical structure of the index file. However, we may follow the useful heuristic that a method with the smaller number of updated nodes is likely to be better. The justification for this heuristics is that the smaller number of updates, the fewer disk I/O needed to read and write them.

To have a naive comparison between ARC and RRC, we decide to choose a context when the XML tree T is balanced and all the nodes of the depth j have s_{j+1} children, where j=0,1,..,k-1. For each ARC and RRC we will count the total numbers of ARC-updated and RRC-updated nodes respectively when data in each leaf node of T is changed once.

Numerate the children of each parent from right to left starting at 1. Given a leaf node n, assume the node path of n

is P $\equiv$ r,n_1,n_2,$\cdots$,n_k=*n* so the node *n* shall be in correspondence with a sequence (1, i_1, i_2,..,i_k), where i_j is the order of n_j (j=1,2,..,k) among its siblings counting from right to left. The participation of number 1 in the sequence is due to every node path starts from the root node and it is unique.

Denote RUpd(*n*) the number of RRC-updated nodes of *n*, RUpd(*T*) the total number of RRC-updated nodes of all leaf nodes of *T*.

Proposition 1. *RUpd(n) = 1 + i_1 + i_2 +..+i_k = $\sum_{j=1}^{k} i_j$ + 1*

Proof. Travel from the node *n* upward to the root *r*. From Observation 1 we conclude that among the siblings of n_j, there are i_j nodes including n_j and its right siblings, that belong to RRC-updated nodes of *n*. □

Proposition 2. *RUpd(T)* = $\frac{1}{2}\prod_{j=1}^{k} s_j(\sum_{j=1}^{k} s_j + k + 2)$ (1)

Proof. From Proposition 1 we know the total number of RRC-updated nodes of all leaf nodes of *T* is $\sum_{Leaves\ set}(\sum_{j=1}^{k} i_j + 1)$. We can decompose the value into 2 components: $A = \sum_{Leaves\ set} 1 = \prod_{j=1}^{k} s_j$ and $B = \sum_{Leaves\ set}\sum_{j=1}^{k} i_j$. Consider the sum B, it is a matrix with *k* rows. Every value ranging from 1 to s_i appears in the *i*-th row totally $\prod_{j=1}^{i-1} s_j \times \prod_{j=i+1}^{k} s_j$ or $\prod_{j=1}^{k} s_j$ / s_i times. Therefore, all the elements corresponding to *i*-th row have sub-sum equal to $\prod_{j\neq i}^{k} s_j(1+2+..+s_i) = \frac{1}{2}\prod_{j=1}^{k} s_j(s_i+1)$. Sum up for *i*, we have $B = \frac{1}{2}\prod_{j=1}^{k} s_j(\sum_{j=1}^{k} s_j + k)$ or $A+B = \frac{1}{2}\prod_{j=1}^{k} s_j(\sum_{j=1}^{k} s_j + k + 2)$.□

Now we are going to count ARC-updated nodes. Denote AUpd(n) the number of ARC-updated nodes of *n*, AUpd(*T*) the total number of ARC-updated nodes of all leaf nodes of *T*. For a given leaf node *u* denote Ord(*u*) the order of *u* in the sequence of leaves starting from right to left in the set of all leaves. Because coordinates are expressed in ARC, any length-update in a leaf will result in update of coordinate of not only its parent but also of all the leaves standing *behind* it. Hence we have the following observation:

Observation 2. *In an XML tree, if n is a leaf node then AUpd(n) is equal to the sum of AUpd(***v***) and Ord(n), where* **v** *is the parent of n*

We denote P_j the value of AUpd() of the tree constructed from *T* by removing all nodes having their node paths longer than j. It is clear that P_k is AUpd(*T*).

Proposition 3. $P_k = \frac{1}{2}\prod_{j=1}^{k} s_j(\sum_{m=1}^{k}\prod_{j=1}^{m} s_j + k + 2)$

Proof. Each leaf node of P_{k-1} is the parent node of s_k children leaves of the P_k and totally there are $\prod_{j=1}^{k} s_j$ leaves in P_k. From Observation 2, we have:

$P_k = s_k P_{k-1} + \frac{1}{2}(\prod_{j=1}^{k} s_j + 1)\prod_{j=1}^{k} s_j$

Similarly applying the formula for P_{k-1} and so on, we have:

$P_k = s_k s_{k-1} P_{k-2} + s_k \frac{1}{2}(\prod_{j=1}^{k-1} s_j + 1)\prod_{j=1}^{k-1} s_j + \frac{1}{2}(\prod_{j=1}^{k} s_j + 1)\prod_{j=1}^{k} s_j$

$P_k = s_k s_{k-1} P_{k-2} + \frac{1}{2}(\prod_{j=1}^{k-1} s_j + 1)\prod_{j=1}^{k} s_j + \frac{1}{2}(\prod_{j=1}^{k} s_j + 1)\prod_{j=1}^{k} s_j$

...

$P_k = \prod_{j=1}^{k} s_j P_0 + \frac{1}{2}\prod_{j=1}^{k} s_j(\prod_{j=1}^{k} s_j + \prod_{j=1}^{k-1} s_j + ... + s_1 + k)$ (2)

Taking into account that P_0 = 1, we have:

$P_k = \frac{1}{2}\prod_{j=1}^{k} s_j(\sum_{m=1}^{k}\prod_{j=1}^{m} s_j + k + 2)$ □

Comparing (1) and (2), we have the update effectiveness ratio of RRC to ARC is:

$$\frac{\sum_{m=1}^{k}\prod_{j=1}^{m} s_j + k + 2}{\sum_{j=1}^{k} s_j + k + 2}$$

This ratio is a theoretical evaluation of the effectiveness of RRC. In practice, this ration is much greater than 1 when most of s_i are equal to or greater than 2.

5. XML indexing structure with RRC

In this section we discuss a data structure that is suitable to store XML elements expressed in RRC.

5.1. Overall structure

Consider B-tree that is widely used in RDBMS. In this data structure, pointers to data address are stored in nodes of a balanced tree. To access a specific node, we travel from the root of the tree, following the appropriate pointers until we reach the destination. It takes one disk I/O to access a child node if we know its parent.

We apply the *grid* approach for XML tree by building such a grid over an XML tree in order to access to information of nodes. The data structure relies on the following observation illustrated in Figure 2, which is concluded from Observation 1:

Observation 3. (Continuos segmentation) *For a given leaf node n, RRC-updated nodes for n are distributed in several segments, each of them contains only sibling nodes.*

The idea to use data structure similar to B-tree to manage structural information of XML data has been applied in other works such as [2]. Our method is based on the same idea but has another approach to construct the *grid* to make it specific to RRC so it can exploits the RRC advantages in the update problem.

5.2. Block Subtree description

Based on Observation 3, we desire to store a node together with its neighbors, including its right siblings in order to reduce the number of disk I/O necessary to perform updates. Given an XML tree, we will divide the node set

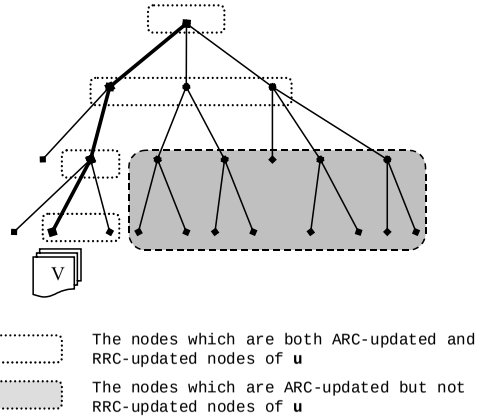

Figure 2. RRC-updated nodes distribution

into groups such as each of them fits into a disk block. Each of these groups is a Block Subtree, denoted by **bst**. The size of **bst** depends on the size of a disk block. In general, sibling nodes can be grouped in a disk block. If the number of siblings is larger, several disk blocks may be needed.

For further specification, we denote b the maximum number such that the information of b nodes together with $6b$ bytes can be stored in a single disk block. The reason why we include extra $6b$ bytes will be explained later in Section 5.4. The information of a node includes the RRC of its corresponding element and a pointer to link related blocks. Normally, a disk block consists of 8K bytes so $b \gg 1$. Also, given a tree T, we denote **nodes(*T*)** the number of nodes in T.

Definition 3. (Cover subtree) *Given a leaf node n of an XML tree T, a cover subtree denoted by* **cst** *for n is an induced subtree of T, that contains the node n.*

We notice that for any leaf node n the root of any of its **cst** belongs to the node path leading from the root of T to n. If C_1 and C_2 are two **cst**s of a node n, and the root of C_1 is ancestor of the root of C_2 then C_1 covers C_2. Based on this property, we have the next definition:

Definition 4. (Block Subtree) *Given a leaf node n of an XML tree T and an integer a>1, the Block Subtree of a nodes for n, denoted by* **bst***(a,n), is the maximal* **cst** *for n, whose number of nodes doesn't exceed a. If the subtree consists only of n then we say that we can not create the* **bst***(a,n).*

Creation of **bst**(a,*n*) is quite straightforward, as explained in Figure 3. Starting from the leaf node n we travel upward to the root and remember the last visited node. At each encountered node, check if the number of nodes in the subtree rooted at that node is less than a. If yes then go

Input: An XML tree T, a given node n of T, and a number $a > 1$
Output: The **bst**(a,*n*)
Function: CreateBst(T, a, n)

for (**c** in the node path from n to the root of T)
  if (**nodes**(**cst** rooted at **c**) $< a$ OR **c** is n) **then**
    lastnode := **c**;
    c := parent of **c**;
  else break **endif;**
endfor;

if (**nodes**(**cst** rooted at **c**) $= a$) **then**
  Return the **cst** rooted at c
endif;

if (**nodes**(**cst** rooted at c) $> a$) **then**
  if (*lastnode* is NOT n) **then**
    Return the **cst** rooted at *lastnode*
  else
    if node n's parent has no grandchild **then**
      Perform *duplication operation* (*)
      Return the **bst**(a,*n*)
    else return null **bst endif;**
  endif;
endif.

Figure 3. Construction of a *bst* for a given leaf node

further. If the number equals to a then output the current subtree. If the number exceeds a then output the subtree rooted at the last visited node.

We briefly explain the step (*) of the algorithm in Figure 3, when we can not build a **bst** because the number of children of a node exceeds a. We divide the set of children into groups of a-1 children starting from the right most child, the last group may consists of less than a-1 children. Duplicate the parent node in order to provide each of these groups with a parent node copy. The "duplicated parents" have the same ID as the ID of the original parent. Each group of children and their parent create a **bst** and these **bst**s are called *direct siblings*. The **bst** whose children set includes n is **bst**(a,*n*). The duplication does not violate Observation 1 and the set of RRC-updated nodes for a given node in fact remains unchanged.

5.3. Algorithm for BST-grid construction

The task is to divide the input XML tree into appropriate **bst**s. The algorithm, as shown in Figure 5, consists of phases and each phase consists of steps. Each step is a construction of a **bst** using the algorithm in Figure 3. A phase finishes when we can not create a new **bst** for any unmarked leaf node of the current tree. After a phase finished, all the nodes participating in created **bst**s, except the roots, are removed from the current tree and a new phase begins.

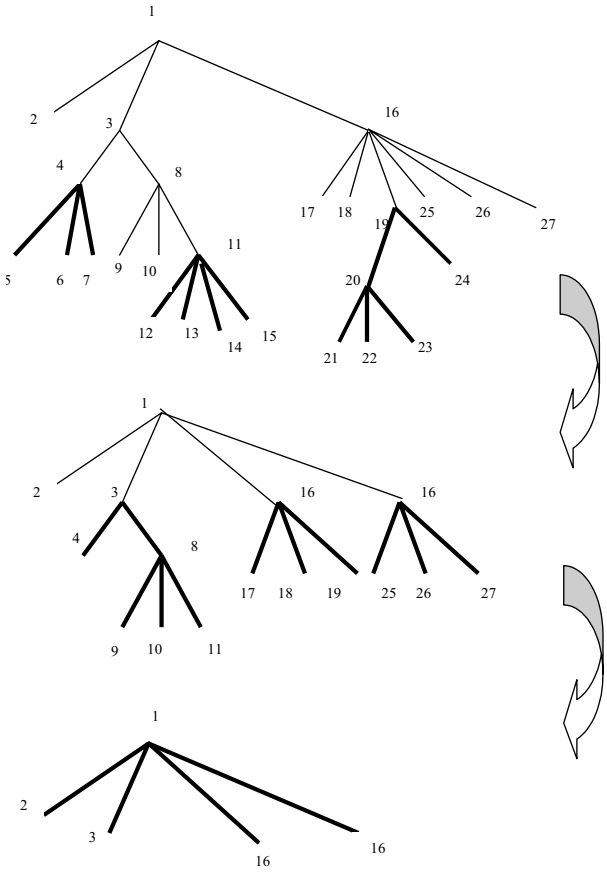

Figure 4. Construction of a BST-grid, a = 6

Because the structure of XML documents can be changed during the exploitation, we should choose the value a to create the **bst**s such that a satisfies the inequality: $2b/3 \geq a \geq b/2$.

Example 5. *Suppose we have an XML tree in the top of Figure 4 and assume that a = 6. It means each* **bst** *can have up to 6 nodes. The curve-arrows show how the* **bst**s *are built. Each induced subtree drawn by bold lines is a* **bst**. *There are 3 phases and 7 steps in the process.*

We assign a *status* to each leaf. At beginning of a given phase, a leaf's status is unsigned, a leaf's status becomes 0 if we could not create a **bst** starting from it, becomes 1 if the node is covered by a **bst**. Node status helps to determine whether the current phase terminates.

The algorithm will be terminated after a finite number of steps. It is easy to see that for any node all RRC-updated nodes will belong to successive **bst**s, upper **bst** contains the root of the lower one, and all of their roots lie in the node path of the node.

Input: An XML tree T, a number a
Output: A set of **bst** of a nodes that covers the XML tree

loop
  loop *// a phase*
    Choose the right most leaf n that is not marked in the current tree
    Run CreateBst(T, a, n) *//Create the* **bst** *for n*
    if success **then**
      Mark the leaves of the just created **bst** with 1
    else Mark n with 0 **endif;**
  until: All leaves are marked in the current tree
  Remove leaves marked with 1 and unmark leaves marked with 0
until: all nodes are removed except the root node of T

Figure 5. Construction of BST-grid of an XML tree

5.4. Block subtree realization

We have assumed that b is the number such that information of b nodes together $6b$ bytes can be stored in one disk data block. This assumption can guarantee that a **bst** with b nodes can be stored in one disk data block.

In fact, we can use b node spaces to store the information of all nodes in a **bst** tree because there are totally b nodes or less in each **bst**. Each node may have RRC, a pointer and is identified by a 4-byte integer. About the tree structure, we have to use a method to compress the information. We use the well-know compact form of representation of a tree where a parent is expressed by a list of its children separated by comma and the list is wrapped by a pair of brackets. For example (a,(b,c),(d,e,f)) represents the tree with a root, three children and two last children have their own 2 and 3 children respectively. We extend the representation in order to include also the identifier of internal nodes by adding the parent node identifier before the list of its children, such as x(a,y(b,c),z(d,e,f)). Assume a tree has i internal nodes and v leaf nodes, where $i + v$ is equals to b. The new form of representation requires only $5v - 1 + 6i$, less than $6b$, bytes to store the structural information of the tree and it means that a **bst** with less than b nodes can fit into a disk block.

We store each **bst** in a disk block and create links among the **bst**, each link points from the block containing the root node of a **bst** to the block containing the whole **bst**. We denote the realization of the **bst** grid **bst-r**. The realization of **bst** grid in Figure 4 is represented in Figure 6.

As we have noticed, for any given node all RRC-updated nodes will belong to successive **bst**s, upper **bst** contains the root of the lower one, and all of their roots lie in the node path of the given node. Because each **bst** stored in a disk block, we need load the disk blocks corresponding to these **bst**s, following the pointers from the nodes belonging to the node path of the given node.

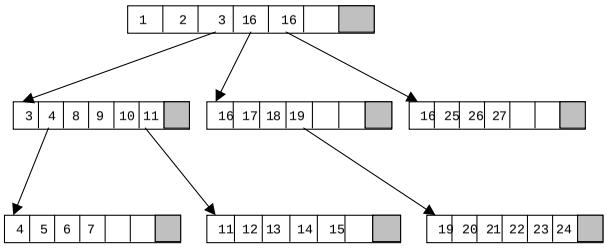

Figure 6. Data block realization

5.5. Structural update of Block subtree realization

In this subsection we will briefly describe the update mechanism for Block Sub Tree.

For Deletion, after deleting a node we consider the possibility of merging the current block with its sibling (the right sibling is higher priority) or parent. When the sum number of nodes of a child and its parent or two consecutive siblings does not excess *b* then the merge operations can be performed because a disk block can hold a subtree with *b* nodes. We omit the algorithm for node deletion here.

For Insertion, after finding the **bst-r** block where the new node will be stored, we have to consider if there is space for the new node. In other words, we have to check if the number of nodes in the **bst** stored in the block equal to *b*, the maximal numbers of nodes allowed being stored in a disk block. If it is, a splitting of the current block is necessary. The algorithm for Insertion is presented in Figure 7.

In the tracking path of these above algorithms we base on the following property: *In any block, among the nodes those belong to the node path of n, there is at most one node that has a pointer to another block.*

5.6. Query support

As we have presented in previous sections, RRC is mainly designed to deal with heavy workload of reconstruction for content update. It means that using the RRC we can maintain the coordinates of XML elements with a low cost of coordinate update.

Now we consider how the XML indexing structure with RRC supports the queries on XML data. We choose two typically common types of path expression written in form of XPath expression [12], such as $/a_1/.../a_i/*$ and $*/b$ where $a_1,..,a_i$ are elements. We will show how to use bst-r to retrieve the XML information defined by these XPath expressions. Rewrite the firts type of XPath expression as following:

$/a_1/../a_{i_1}/a_{i_1+1}/../a_{i_2}/.../a_{i_{k-1}}/a_{i_{k-1}+1}/.../a_{i_k}/*$

Input: An XML tree *T* and its **bst-r**, a node *n* and its node path, and node **m**, an integer *i*
Output: The **bst-r** of *T* after insertion of **m** as the i^{th} child of *n* in *T*

Load the root block of the **bst-r** and set it the current block
loop
  if node *n* belongs to the current **bst-r** block
    if there is a space for the new node **m**
      Insert the new node **m** into the i^{th} position of the current block.
      Update information about the subtree stored in the block
    else
      Split the current block and use the algorithm creating **bst** in Figure 3 to build **bst** grid on the subtree after insertion
    endif;
    Return;
  else
    In the current block, find the node belonging to the node path of *n* and having a pointer to another block
    This block becomes the current block
  endif;
until;

Figure 7. Insertion of a node into a bst

and suppose that by reading the root of **bst-r** we can load into the main memory $a_1,..,a_{i_1}$. Checking the pointers corresponding to $a_1,..,a_{i_1}$ and follow the pointers pointing to the next element in the XPath expression, a_{i_1+1}, to load the blocks, each of those contains some continuos nodes $a_{i_1+1},..,a_{i_2}$. The argument of this process is similar to one in Subsection 5.5. Repeat the process until the element a_{i_k} is loaded into main memory. Any downward path from a_{i_k} contains the result of the query.

Queries of the second type of XPath expression can be performed using an *invert indexing structure* expressing the relationship between elements and **bst**. Using UID [4], for each XML element **b** we list the **bst** containing the element. Together, they form an element of the invert index.

5.7. Case improvements

There are some improvements that can be applied in specific cases. The technical details can be found in [3]:

Higher organization of Block Subtree. When the XML collection is huge, we need to further develop the index schema by considering the **bst-r** as the tree source and build **grid**s for the tree.

Two datum-points for RRC. When the number of children of a parent excesses the threshold *b*, we divide the set of children into two subsets of approximately equal cardinalities. The coordinates of children in these subsets are computed based on the first and second coordinate of the parent node respectively.

Spare space number. Using DTD to anticipate the needed space that makes **bst-r** structure relatively **stable** in term of reducing number of its structural updates when the structure of the source data changes.

6. Storing XML data in relational databases using RRC

There are several approaches to use commercial databases system for storing and indexing XML data [5] [9]. We briefly present here an XML storage system with RRC based on the system introduced in [9].

Based on Object-Relational database, XML data is decomposed into four relations **Element**, **Attribute**, **Text**, **Path**. Each word has its absolute position within document as *integer number*. Each tag is represented by a pair of a word position and position of tag in the current sequence of tags, starting from the word. An element coordinate is a pair of start tag and end tag.

We redefine the coordinate of a word, from "the word position in whole document" to "the word position in parent element". The relative coordinate of a tag becomes defined by the relative position (in the parent region) of the nearest left word and by the order of the tag in the sequence of tag starting from the word. Similarly with the argument in Section 3, by this way we can reduce the scope of content update. It is worthy to remark that by expressing coordinate in RRC, the relationship of absolute positions of words can be expressed through the relative position. In order to take the advantages of built-in indexing techniques of the database system, we extend the coordinate of an element into <path-info, start-tag, end-tag>. By choose appropriate path information, R*-tree on the field can be constructed.

7. Conclusion

Reconstruction of index file due to a partial update is a problem that XML database applications inevitably have to face. Absolute Region Coordinate is convenient to express location of XML element but it can not help in solving this problem. In this work we have presented a method, called Relative Region Coordinate, to express XML element coordinate based on the region of parent element, therefore we can reduce the workload of the reconstruction. We also have proposed a data structure that exploits the advantages of RRC. The index schema features that siblings of a parent are grouped in one or several **bst**. Each **bst** can be stored in a disk block, therefore we need a few disk I/O to update coordinate of a group of siblings.

RRC of the elements can be easily computed based on ARC. Another way is to compute RRC in bottom up manner based on the DOM of the given document. The second approach has been successfully implemented on the collection of plays of Shakespeare using the DOM parser from the Xerces.1.0.3 software foundation.

Our future work will focus on how to adapt the index schema to specific XML query languages and evaluating the effectiveness of the method. Due to the complex structure of XML documents, an index that is efficient for both update and retrieval may not available. One of alternatives is building two separate indices such that one is suitable when update is frequent, the other is better at query processing. In this case, a transformation mechanism between the indexing structures is need to be developed.

References

[1] A. Biliris. An efficient database storage structure for large dynamic objects. *Proc. of ICDE, Arizona, USA*, pages 301–308, 1992.

[2] C.C Kanne, Guido Moerkotte. Efficient storage of xml data. *Proc. of ICDE, California, USA*, page 198, 2000.

[3] D.D.Kha, M.Yoshikawa, S.Uemura. An xml indexing structure with relative region coordinate. *Technical report. Graduate School of Information Science, Nara Institute of Science and Technology*, 2000.

[4] H.Jang, Y.Kim, D.Shin. An effective mechanism for index update in structured documents. *Proc. of CIKM, Kansas, USA*, pages 383–390, 1999.

[5] J.Shanmugasundaram, K.Tufte, G.He, C.Zhang, D.DeWitt, J.Naughton. Relational databases for querying xml documents: Limitations and opportunities. *Proc. of VLDB, Edinburgh, Scotland*, pages 302–314, 1999.

[6] Michael J. Carey, David J. DeWitt, Joel E. Richardson, Eugene J. ShekitaMichael J. Carey, David J. DeWitt, Joel E. Richardson, Eugene J. Shekita. Object and file management in the exodus extensible database system. *Proc. of VLDB, Kyoto, Japan*, pages 91–100, 1986.

[7] R.Sacks-Davis, T.Dao, J.A. Thom, J.Zobel. Indexing Documents for Queries on Structure, Content and Attributes. *Proc. of International Symposium on Digital Media Information Base (DMIB), Nara, Japan*, pages 236–245, 1997.

[8] Tobin J. Lehman, Bruce G. Lindsay. The starburst long field manager. *Proc. of VLDB, Amsterdam, Netherlands*, pages 375–383, 1989.

[9] T.Shimura, M.Yoshikawa, S.Uemura. Storage and retrieval of xml documents using object-relational databases. *Proc. of DEXA, Florence, Italy. Lecture Notes in Computer Science*, 1677:206–217, 1999.

[10] World Wide Consortium. Extensible markup language (xml) 1.0. *http://www.w3.org/TR/REC-xml*, 2000.

[11] World Wide Consortium. Xml information set. *http://www.w3.org/TR/xml-infoset*, 2000.

[12] World Wide Consortium. XML Path language (XPath) Version 1.0. *http://www.w3.org/TR/xpath*, 2000.

[13] Y.Yamamoto, M.Yoshikawa, S.Uemura. On indices for xml documents with namespaces. *Markup Technologies, Philadelphia, USA, GCA*, pages 235–243, 1999.