IPSJ Transactions on Databases Vol. 3 No. 1 1–17 (Mar. 2010)

Regular Paper

A Labeling Scheme for Dynamic XML Trees Based on History-offset Encoding

Bei Li,[†1] Katsuya Kawaguchi,[†2] Tatsuo Tsuji[†1] and Ken Higuchi[†1]

This paper presents a novel labeling scheme for dynamic XML trees. The scheme employs *history-offset* encoding method for multidimensional datasets and takes advantage of this method by embedding an XML tree into a multidimensional extendible array. Even if structural updates are made on the XML tree, no relabeling of nodes is required under the support of extra data structure for preserving the document order. The most significant advantage of our scheme over other existing labeling schemes is that the storage cost for generated labels is very small irrespective of the order and the position of node insertions; in most of our competing schemes, the generated label size would become very large if the insertions occur around the same position. After describing our labeling scheme, label size, total label storage cost and node access performance are examined compared with other sophisticated schemes, such as ORDPATH, QED, DLN and Prime Numbering, and proves that our scheme outperforms these schemes in some criteria.

1. Introduction

Recently, XML [1] is observed as an efficient way to represent the semi-structured data, and efficient storing and querying XML data have gained more attention. In order to handle structure of XML data efficiently, it is important to provide a labeling scheme for XML nodes that can well capture the underlying tree structure. Various labeling schemes, such as range based labeling scheme [2], prefix scheme [3),4)], prime number labeling scheme [5] and other several approaches [6] have been proposed and analyzed. In these labeling schemes, for a given label of an XML node, its axis node such as parent, child or sibling can be determined based on the given label value.

†1 Graduate School of Engineering, University of Fukui
†2 Toshiba Solutions Corporation

Reference 8) takes an approach which maps an XML tree to a complete k-ary tree, and some improved schemes of this approach can be found in Refs. 9), 10). In these schemes, a query on an axis such as *parent* or *sibling* can be quickly answered through simple arithmetic operation on label values. However, they are mainly for static XML trees; when new nodes are dynamically inserted, almost all of the node label values are necessary to be recomputed to preserve document order. Moreover, they are not efficient in consumption of label value space and the space can be quickly saturated by the dynamic node insertions.

There exist some sophisticated labeling schemes with their encoding methods which support dynamic structural updates such as node insertions and deletions. These schemes include ORDPATH [11], QED [12] and DLN [13] etc. The advantage of such schemes is that they can preserve document order without relabeling any existing nodes even if dynamic structural updates are made. But the important disadvantage of them is that their label size and encoded label value size can increase rapidly when concentrated insertions of new nodes occur around the same position in the XML tree.

This paper proposes a new labeling scheme based on *history-offset* encoding method for multidimensional datasets presented in Refs. 14), 15). The scheme takes advantage of the history-offset encoding method by embedding an XML tree into a multidimensional dynamically extendible array. It can preserve the advantage of the k-ary tree scheme while resolving or alleviating its drawbacks. Namely, in our new labeling scheme, like in Refs. 8)–10), a query on an axis can be quickly answered through simple arithmetic operation on label values, and the scheme also preserves the advantage of the flexible extendibility of the history-offset encoding method, which is appropriate for handling dynamic XML trees. Moreover, the most significant advantage of our scheme over ORDPATH, QED and DLN labeling schemes is that the storage cost for label values is small irrespective of the node insertion order and places.

On the other hand, the important drawback of our scheme is that separate data structures are necessary to preserve the document order among siblings for dynamic node insertions or deletions, and the size of these data structures is fairly large. But, practically, this drawback can be compensated by the direct accessibility to the sibling nodes. Note that this drawback is not shared in the

labeling schemes satisfying the above specified requirement, but the length of labels tends to be long in order to include the document order information in the label value itself depending on the node insertion order and places.

In general, the purpose of labeling XML tree nodes is to use the node labels for retrieving nodes, so that the label of a node is to carry the position information of the node in the XML tree. It is required that the ordering between arbitrary two elements in an XML document corresponding to their nodes can be directly known by using only such position information kept in the labels.

In our history-offset labeling scheme presented in this paper, similar to the prime number labeling scheme presented in Ref. 5), is not possible to determine the document order only by label values themselves, and some extra information or data structures are necessary. So the scheme is not classified into a labeling scheme in the strict sense specified above, but in the following we say the scheme as a labeling scheme and an encoded value by the scheme as a label or label value under such restriction. Both of our scheme and the prime number labeling scheme employ a table in order to determine the document order among nodes.

After describing our labeling scheme, label size, total label storage cost and node access performance are examined compared with DLN, ORDPATH, QED, then proves that our scheme outperforms these schemes in some criteria. Then, the prime number labeling scheme is also evaluated and compared with our scheme. The rest of the paper is organized as follows. Section 2 introduces the background of our new labeling scheme and Section 3 presents the new labeling scheme. In Section 4, we provide the way to compute label value of an axis node of a context node. After discussing related work in Section 5, we give experimental results of our scheme compared with other competing schemes in Section 6. The last section concludes the paper.

2. History-offset Encoding Method

In this section, we briefly describe *history-offset* method [14),15)] for encoding multidimensional datasets as the basis of our labeling scheme. Here we discuss on implementation of a relational table, which is a typical example of a multidimensional dataset.

2.1 Extendible Array

In the conventional implementation of relational tables, each tuple is placed on secondary storage one by one in the input order. This arrangement suffers from some shortcomings.

(1) The same column values in different tuples will have to be stored many times and hence the volume of the database increases rapidly. Such a situation is typical in the columns of categorized value like “blood group”, “sex” etc.

(2) In the retrieval process of tuples of a specified column value, unless some indexes are prepared, it is necessary to load tuples in the table sequentially in main memory and check the column value. In consequence, the whole table should be fetched into memory. Therefore retrieval time tends to be long.

The implementation using multidimensional arrays can be used to overcome problem (2) above and can perform retrieval not in the sequential manner. Each column of a relational table is mapped to a distinct dimension of the corresponding array. Nevertheless, such an implementation causes further problems:

(3) Conventional schemes for storing arrays do not support dynamic extension of an array and hence addition of a new column value is impossible if the size of the dimension overflows.

(4) In ordinary situation, implemented arrays are very sparse.

The concept of *extendible array* we will employ is based upon the index array model presented in Ref. 16). An n dimensional extendible array A has a history counter h and three kinds of auxiliary table for each extendible dimension $i (i = 1, \ldots, n)$. See **Fig. 1**. These tables are history table H_i, address table L_i, and coefficient table C_i. The history tables memorize extension history. If the size of A is $[s_1, s_2, \ldots, s_n]$ and the extended dimension is i, for an extension of A along dimension i, contiguous memory area that forms an $n-1$ dimensional subarray S of size $[s_1, s_2, \ldots, s_{i-1}, s_{i+1}, \ldots, s_{n-1}, s_n]$ is dynamically allocated. Then the current history counter value is incremented by one, and it is memorized on H_i, also the first address of S is held on L_i. Since h increases monotonously, H_i is an ordered set of history values. Note that an extended subarray is one to one corresponding with its history value, so the subarray is uniquely identified by its

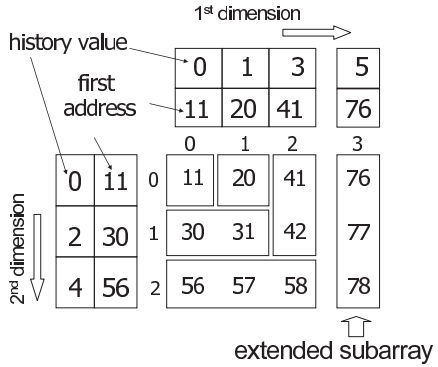

Fig. 1 An extendible array.

history value.

As is well known, element $(i_1, i_2, \ldots, i_{n-1})$ in an n-1 dimensional fixed size array of size $[s_1, s_2, \ldots, s_{n-1}]$ is allocated on memory using addressing function like:

$$f(i_1, \ldots, i_{n-1}) = s_2 s_3 \ldots s_{n-1} i_1 + s_3 s_4 \ldots s_{n-1} i_2 + \ldots + s_{n-1} i_{n-2} + i_{n-1} \quad (1)$$

We call $< s_2 s_3 \ldots s_{n-1}, s_3 s_4 \ldots s_{n-1}, \ldots, s_{n-1} >$ as a coefficient vector. Such a coefficient vector is computed at array extension and held in a coefficient table. Using these three kinds of auxiliary tables, the address of element $(i_1, i_2, \ldots, i_n)$ can be computed as:

(a) Compare $H_1[i_1], H_2[i_2], \ldots, H_{n-1}[i_n]$. If the largest value is $H_k[i_k]$, the subarray corresponding to the history value $H_k[i_k]$, which was extended along dimension k, is known to include the element.

(b) Using the coefficient vector memorized at $C_k[i_k]$, the offset of the element $(i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_n)$ in the subarray is computed according to its addressing function in Eq. (1).

(c) $L_k[i_k]$+(the offset in (b)) is the address of the element.

For example, consider the element $\langle 2, 2 \rangle$ in Fig. 1. Since, $H_1[2] < H_2[2]$, it can be known that the element is involved in the extended subarray S of history value $H_2[2] = 4$. So the first address of S is known to be $L_2[2] = 56$. Since the offset of the element $\langle 2, 2 \rangle$ from the first address of S is 2, the address of the element is determined as 58.

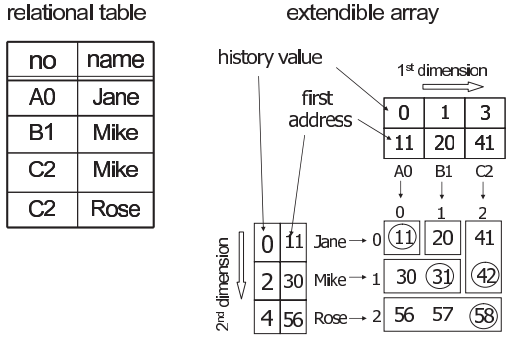

Fig. 2 Relational table implementation using an extendible array.

Note that we can use such a simple computational scheme to access an extendible array element only at the cost of small auxiliary tables.

2.2 History-offset Encoding Method and HOMD Implementation Model

Figure 2 shows a realization of a two column relational table using a two dimensional extendible array. In general, for a relational table R of n columns, each column can be mapped to a dimension of an n dimensional extendible array A and each column value of a tuple in R can be uniquely mapped to a subscript of the dimension. Hence, each tuple in R can be mapped to an n dimensional coordinate of an element in A. Moreover each coordinate of an element in A can be mapped to the history value of the subarray S including the element and the offset value in S as was stated in Section 2.1. In consequence, each tuple in R can be encoded to its corresponding pair of <*history value, offset value*> . We call this tuple encoding method as *history-offset* encoding.

This encoding method is the basis of our XML node labeling scheme which will be presented in the later sections. In the coordinate representation $(i_1, i_2, \ldots, i_n)$ of an array element location, if the number of the dimensions of the extendible array becomes larger, the length of the coordinate becomes longer proportionally and the storage to hold tuples become larger. On the contrary, in our history-offset encoding, even if the number of dimensions becomes larger, the size of the encoded tuple is fixed in short; i.e., only the two kinds of scalar value. It should

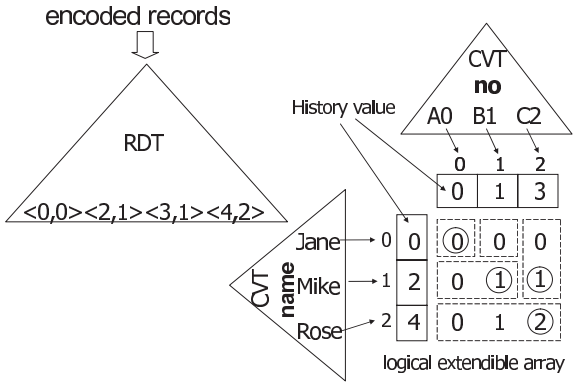

Fig. 3 Relational table implemented by HOMD.

be noted that this encoded tuple also can be easily decoded to the corresponding coordinate. This saves over all storage for a relational table or label storage for XML tree nodes encoded by our history-offset method. Moreover, internal handling of table tuples or XML nodes in DBMS becomes simple and efficient owing to the fixed size reference.

HOMD (History Offset implementation for Multidimensional Datasets) is an implementation scheme of multidimensional datasets using *history-offset* encoding method. **Figure 3** is the HOMD implementation of the relational table in Fig. 2. HOMD is a pair (M, A) where A is an n dimensional extendible array created for R and M is a set of mappings. Each $m_i (1 \leq i \leq n)$ in M maps the i-th attribute values of R to subscripts of the dimension i of A. A will be often called as a *logical extendible array*. m_i is implemented using a single B^+ tree called CVT (key subscript ConVersion Tree), and A is implemented using a single B^+ tree called RDT (Real Data Tree) and n HTs (HOMD tables), each of which is defined in the following.

Definition 1 CVT: CVT_k for the k-th attribute of an n dimensional datasets is defined as a structure of B^+ tree with each distinct attribute value v as a key value and its associated data value is subscript i of the k-th dimension of the logical extendible array A. Hence the entry of the sequence set of the B^+ tree is the pair (v, i). Subscript i references to the corresponding entry of the HOMD table in the next definition.

Definition 2 HT: HT (HOMD Table) corresponds to the auxiliary tables explained in Section 2.1. It includes the history table and the coefficient table. Note that the address table can be void in our HOMD physical implementation.

Definition 3 RDT: The set of the pairs <*history value, offset value*> for all of the effective elements in the extendible array are housed as the keys in a B^+ tree called RDT. Here, the effective elements mean the ones that correspond to the tuples in the relational table.

Note that RDT together with HTs implements the logical extendible array on the physical storage. A key <history value, offset value> occupies fixed size storage and we assume that the *history value* is arranged in front of the $offset\ value$. Hence the keys are arranged in the order of their history values and keys that have the same history value are arranged consecutively in the sequence set of RDT. Note also that since only effective elements are stored in RDT, problem (4) discussed in Section 2.1 can be resolved.

Definition 4 HOMD: For an n dimensional datasets, its HOMD implementation is the set of n CVTs, n HTs and RDT.

3. History-offset Labeling Scheme for Dynamic XML Trees

In this section, using *history-offset* encoding method described in Section 2.2, we propose our labeling scheme for XML tree nodes. Our scheme takes advantage of the history-offset method by embedding XML tree into HOMD data structure.

3.1 Labeling Scheme Based on Fixed Size Multidimensional Array

As is shown in **Fig. 4**, each depth level of an XML tree is mapped to a dimension of a usual fixed size multidimensional array. The number that represents each node's relative position among its siblings is mapped to the subscript value of the corresponding array dimension. Therefore, each node can be uniquely specified by its corresponding n dimensional coordinate in the multidimensional array. It can be labeled with the value computed for the corresponding coordinate using the addressing function of the multidimensional array. Note that each level of an XML tree is mapped to the dimension number of the array in ascending order, and in particular the root node of XML tree is labeled with 0.

In this approach, using the addressing function, the label value of a node can

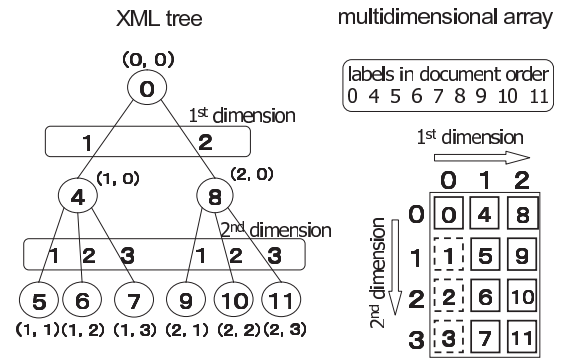

Fig. 4 Labeling XML tree using fixed size multidimensional array.

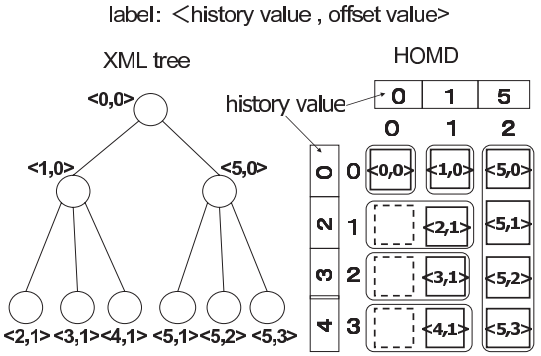

Fig. 5 Labeling XML tree using *history-offset* encoding.

be easily computed from the coordinate of the node. Conversely, the label value can be decoded to the corresponding coordinate using the coefficient vector of the addressing function. Moreover, given the label value of a node, labels of its parent, siblings, and children can be directly known through simple arithmetic operations.

But this approach assumes that the array size is fixed in every dimension. It cannot handle dynamical structural updates on an XML tree, such as new node insertions or existing node deletions that affect the dimension size. For such updates, extension or reduction of the multidimensional array is required. But this results in modification of the addressing function, which triggers re-computation of all node label values. Moreover, when height of an XML tree increases, re-computation of all label values is also caused, since a new dimension of the multidimensional array needs to be created.

Although there are many advantages in the multidimensional array based labeling scheme, it can only handle static XML trees. Similar approaches such as described in Refs. 8)–10) which embed an XML tree into an k-ary tree share the same advantages and disadvantages described above.

3.2 Labeling Scheme Based on History-offset Encoding

To overcome the problem of the fixed size multidimensional array based labeling scheme, we employ the history-offset encoding method described in Section 2.2. An XML tree can be embedded in HOMD data structure described in Section 2.2. An example of an XML tree embedded in HOMD is shown in **Fig. 5**. As described in Section 3.1, also in HOMD, each level of XML tree is mapped to a dimension of HOMD, and each node's relative position among its siblings is uniquely mapped to a subscript of the dimension. Thus, the coordinate of a node can be obtained. Note that for the path expression from the root node to the specified node, each element name in the path expression can be converted to the subscript of the corresponding dimension of HOMD by using its CVT. It is assumed that the element name conversions have already been done as other XML tree node labeling schemes, so the CVTs in HOMD do not matter in the succeeding discussions.

Using our history-offset encoding, a node can be labeled with the pair of history value and offset value which will be stored in RDT of HOMD. A coefficient vector is prepared for each subarray in HOMD, while in a fixed size multidimensional array a single coefficient vector is prepared as a whole. A coefficient vector can be used for computing labels of parent, child and sibling axis nodes. In Fig. 5, assuming that an XML tree grows in pre-order node sequence, extension of subarrays and <history value, offset value> labels of the tree nodes are shown.

Note that in our history-offset encoding based labeling, the extendibility of HOMD makes the relabeling and reorganization of existing nodes unnecessary even if any new nodes are dynamically inserted. Moreover, even if an XML tree grows high, namely the dimensionality of HOMD increases, the size of node's

label value is being fixed in short.

3.3 Handling Insertion and Deletion of an XML Tree Node

XML tree node insertion can be accommodated by an extended subarray in HOMD. The extension would occur when a new node insertion causes the maximum fan out of the level to increase. For example, see the XML tree in Fig. 5. If a new node is inserted on the first level of the XML tree, the first dimension of HOMD would be extended and a subarray of history value 6 is dynamically allocated and attached along the first dimension. The node of label $< 6, 0 >$ can be placed at the position of coordinate $(3, 0)$. On the contrary, node deletion can be accommodated by reduction of a subarray. For example, deletion of all nodes in the subtree rooted at node labeled $< 5, 0 >$ would cause the subarray of history value 5 to be deleted and the first dimension of HOMD would shrink by one.

3.4 Handling Dynamic Changes of XML Tree Height

When XML tree height increases by one, number of dimensions of HOMD also increases by one. Dimensional extension of HOMD can be very efficiently handled. **Figure 6** shows addition of a new dimension against a two dimensional HOMD. When a new dimension is added to an n dimensional HOMD, HOMD table of the new dimension would be created and initialized. For all the existing elements in the original HOMD the subscript of the newly added dimension becomes 0. It should be noted that relabeling of the nodes in the original HOMD is not necessary, since <history value, offset value> labels of any nodes are not necessary to be modified. Conversely, for the case that the height of an XML tree decreases by the deletion of all the nodes on maximum level of the XML tree, dimension of HOMD decreases by one. Note also that relabeling of the nodes in the original HOMD is not necessary.

On the contrary when *drop column* or *add column* command for a relational table is performed, great overhead is required since reorganization of all the existing tuples would be necessary.

3.5 Preserving Order among Siblings in Dynamic Environment

HOMD cannot preserve the logical order (i.e., document order) among siblings in dynamic environment, because the subscript value for each child node of the same parent node are assigned in ascending order of insertion time irrespective of

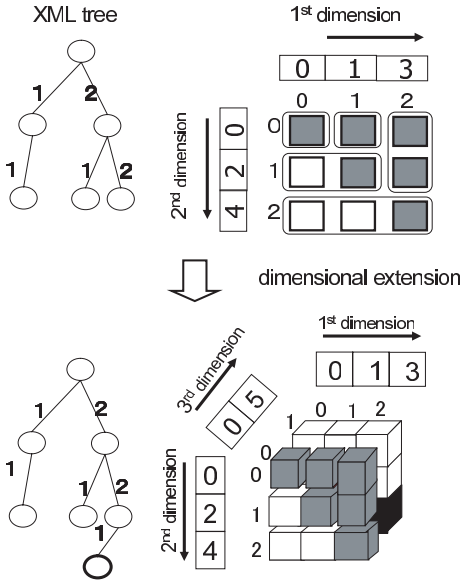

Fig. 6 Dimensional extension in HOMD.

the logical order among them. In order to preserve the logical order among sibling nodes, an additional table called *os* table (*order of siblings* table) is maintained in each parent node. *os* table is a one dimensional array and serves to keep the subscripts in the sibling order. It keeps the subscript of the first child. See **Fig. 7**.

4. Axis Computation

For the labeling schemes presented in Section 3, this section describes computation of axes defined in XPath language[18].

4.1 Labeling Scheme Based on Fixed Size Multidimensional Array

Using the addressing function, address of a node can be computed based on its coordinate as described in Section 3.1, and the computed address can serve as the label of the node. The label of its parent, siblings can be computed based on the label. We call the node labeled with m simply as node m and denote $l(m)$ for the level of the XML tree on which node m is located. Let C_l be the l-th value of the coefficient vector. For node m, the labels of its parent, children, and

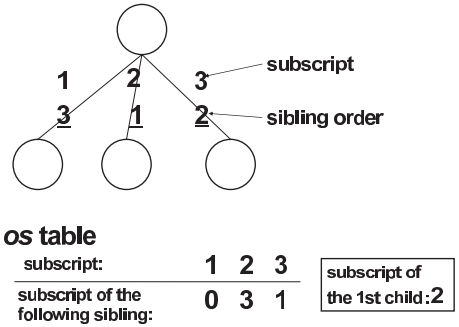

Fig. 7 *os* table.

next sibling can be computed using the below simple formulae:

$$\text{parent: } \begin{cases} 0 & (l(m)=1) \\ m - m\%\ C_{l(m)-1} & (l(m)\geq 2), \end{cases}$$

$$k\text{-th child: } \begin{cases} m + k * C_{l(m)+1} & (l(m)+1 < max\ level) \\ m + k & (l(m)+1 = max\ level) \end{cases}$$

$$\text{next sibling: } \begin{cases} m + C_{l(m)} & (l(m) < max\ level) \\ m + 1 & (l(m) = max\ level) \end{cases}$$

Here $a\%b$ denotes the remainder of m divided by b. **Figure 8** shows several axes of node (2, 2, 0, 0) as being the context node. Note that the level of this context node is 2.

4.2 Labeling Scheme Based on History-offset Encoding

In our history-offset encoding method, if the history values of the labels of the context node and its query node along an axis are the same (e.g., nodes $< 5, 1 >$ and $< 5, 2 >$ in Fig. 5), both nodes can be known to belong to the same subarray. In this case, the label of a query node can be computed based on the similar way as described in Section 4.1, since the size of the subarray is fixed in each dimension.

However, if the history values of the both nodes' labels are different (e.g., nodes $< 2, 1 >$ and $< 3, 1 >$ in Fig. 5), these two nodes are known to belong to different subarrays. In this case, we cannot compute the label of the query node based on the formulae described in Section 4.1. Instead, the label should be calculated based on its corresponding coordinate in the logical extendible array in HOMD.

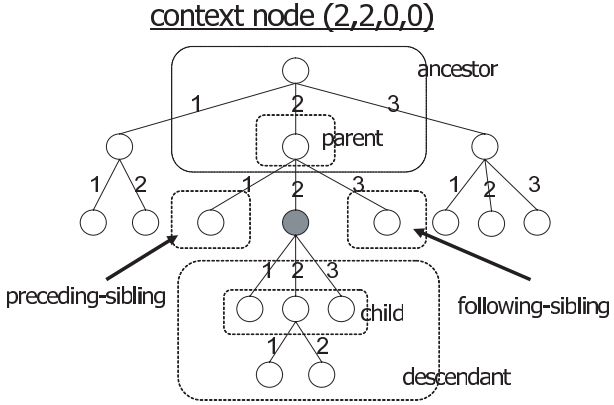

Fig. 8 XML tree axes.

For example, consider the context node (2, 2, 0, 0) in Fig. 8, the coordinate of the parent, namely (2, 0, 0, 0) can be known by replacing coordinate value at the position of the context node level with 0. Then coordinate (2, 0, 0, 0) is encoded to the label <history value, offset value> of the parent node.

Decoding of the label <history value, offset value> of a node to its corresponding coordinate $(i_1, i_2, \ldots, i_n)$ can be performed as:

(i) The dimension p and the subscript i_p corresponding to the subarray including the node can be known from the history value by referring to the history table.

(ii) Let $o(o > 0)$ be the offset value of the label and let $(c_1, c_2, \ldots, c_{n-2})$ be the coefficient vector of the subarray determined in (i), then o can be computed as:

$$o = c_1 i_1 + c_2 i_2 + \ldots + c_{p-1} * i_{p-1} + c_p * i_{p+1} + \ldots + c_{n-2} * i_{n-1} + i_n$$

Therefore, the coordinate i_k can be computed as:

$$i_1 = o/c_1,\ i_n = o\%c_{n-2},$$

$$i_k = \begin{cases} (o\%c_{k-1})/c_k & (k < p), \\ (o\%c_{k-2})/c_{k-1} & (k > p) \end{cases}$$

Conversely, encoding of the coordinate of a node into its label <history value, offset value> can be performed as in the element address computation of an extendible array explained in Section 2.1; the largest history value in step (a) gives *history value* and the offset in step (b) gives *offset value*.

4.3 Determining Document Order Between Two Node Labels

Owing to the *os* table introduced in Section 3.5, the labeling scheme described in Section 4.2 can output the computed labels in document order for the axes shown in Fig. 8 with no additional cost. But, for the two arbitrary node labels l_1 and l_2 in the XML tree, the following separate computation is necessary to determine the order between them. It begins with the computation of the least common ancestor node label of l_1 and l_2. Let $(i_1, i_2, \ldots, i_n)$ and $(j_1, j_2, \ldots, j_n)$ be the decoded coordinates of l_1 and l_2 computed according to the computations described in Section 4.2.

(1) Let $k = 1$, and compute i_k and j_k
(2) while $(i_k == j_k)$ { $k++$; compute i_k and j_k}
    /* finding the least common ancestor */
(3) from the first subscript of the *os* table kept in the node $(i_1, \ldots, i_{k-1}, 0, \ldots, 0)$, traverse the *os* table until the subscript i_k or j_k is found.
(4) if i_k is found, $l_1 < l_2$, otherwise $l_2 < l_1$.

5. Related Work

First we overview two kinds of approach related to our labeling schemes. One is similar to our approach, in which the label of a query node can be obtained by simple computation on the context node label. In Ref. 7), a labeling scheme employing a k-ary complete tree is proposed. An XML tree can be embedded into a k-ary complete tree, in which simple arithmetic operation can afford the label of a query node along an axis like in Section 4.1. But as was stated in the introductory section, this scheme is only for static XML trees, since overall relabeling of XML nodes would be caused by dynamic insertion of a node. Moreover, the complete tree is so sparse in general and the label space would be wastefully consumed. In Refs. 8) and 9) this deficiency of label space consumption is alleviated by setting the least *arity* on each level of the XML tree instead of using a complete tree, in which the *arity* is the same in every level. For the same deficiency, Ref. 10) discusses another approach which is essentially the same as Refs. 8), 9).

These improved approaches can delay the saturation of label space, but the saturation is unavoidable for large or irregular shaped XML trees. In contrast with these approaches, our history-offset labeling scheme can flexibly handle dynamic XML trees, and the label space saturation can be significantly delayed than the above schemes. In our labeling scheme, the scheme proposed in Ref. 15) can be applied for the label space saturation problem.

The other approaches concern to handling structural updates efficiently. Node label encoding schemes for dynamic XML trees such as ORDPATH [11] and DLN [13] have been proposed. In QED [12] structural updates can be efficiently supported based on lexicographical order. Moreover, we can replace the decimal numbers in the prefix labeling scheme for a hierarchical structure represented by a tree with QED codes (called as QED-PREFIX later on). The main benefit of these labeling schemes is that they can maintain document order without relabeling even if structural updates are made against the XML tree. For example in ORDPATH, a new label '1.4.1' can be inserted between two existing labels '1.3' and '1.5'. For DLN, a new label '1.2/1' can be inserted between two existing labels '1.2' and '1.3'. In QED-PREFIX, a new binary string label '10.1010' can be inserted between two existing labels '10.10' and '10.11'. Note that they are based on prefix labeling schemes and the document order among nodes can be determined by comparing their labels in lexicographical order.

Although the approaches of these labeling schemes can flexibly handle structural updates while keeping the document order, the label sizes in them can be easily increased when node insertions are made. Specifically, the label sizes can be extremely increased in the cases of concentrated insertions around the same position. In **Fig. 9**, choosing ORDPATH scheme as an example, we can observe that concentrated node insertions increase label sizes extremely fast. In the presence of such concentrated node insertion sequence, the label size in DLN and

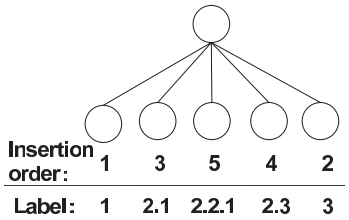

Fig. 9 Concentrated insertions in ORDPATH labeling.

QED code in QED-PREFIX will also be much increased.

Prime number labeling scheme [5] takes an approach similar to our history-offset scheme, where an extra data structure is supplied to know the document order of labels. The scheme will be discussed and evaluated in Section 6.5.

6. Experimental Evaluation

We evaluate and compare the performance of our scheme based on the history-offset encoding described in Section 3, with ORDPATH, QED (QED-PREFIX), and DLN labeling schemes. The experiments were performed on Sun Blade 1000 Workstation of 64 bits (CPU: UltraSparc III (750 MHz), memory size: 512 MB, disk storage: 208 GB, OS: Solaris 8).

6.1 Setup of Experiments

6.1.1 XML Datasets and Storing Node Labels

We used an XML document produced by XMark document generator [19]. Size of the produced XML document is 15,214,640 bytes. From the XML document, we extracted and generated four kinds of XML dataset shown in **Table 1**, which describes the height of the corresponding XML tree, total number of nodes, average level of nodes in the tree.

For each kind of XML dataset, we used SAX [20] to produce file XF for storing the corresponding XML tree shape (i.e., topology). In order to handle dynamic insertions of XML tree nodes and to enable both random and range access of node labels efficiently, we used B^+ trees to store node labels on secondary storage. For history-offset scheme this B^+ tree is RDT and for the other schemes this B^+ tree is called LT (Label Tree). Being XF file as input, the *label generator* computes the labels for the four kinds of labeling scheme according to the two kinds of node insertion order described in the next subsection. An XML tree grows from empty and the final shape of an XML tree after insertion of all nodes is the same irrespective of the node insertion order; i.e., same as specified in XF file. In our experiment however, it should be noted that instead of actually inserting new nodes and constructing an XML tree, the label values of the nodes are computed and inserted into RDT or LT. In the following for the ease of understanding, we often say as if an XML tree is actually constructed.

Table 1 Tested XML trees.

tree	height	total num. of nodes	average node level
tree1	3	62,474	2.89
tree2	5	261,048	4.00
tree3	6	297,163	4.24
tree4	7	334,176	4.55

6.1.2 Node Insertion Order

In the labeling schemes other than history-offset scheme, the label size depends on the order of node insertions. In order to evaluate the required label size and total label storage cost, labels for XML tree nodes are generated according to the following two kinds of *depth-first* node insertion order. One is *random insertion order* and the other is *concentrated insertion order*. At a new node insertion, for the context node, the random insertion order randomly selects one of the child nodes in XF and the concentrated insertion order selects the child node around the same position as is shown in Fig. 9.

6.1.3 Handling Large Size Labels and *os* Tables

As was stated in Section 5, in the labeling schemes other than our history-offset scheme, size of labels becomes very large depending on the node insertion order. In this situation, it is inappropriate to insert large label values of variable size into B^+ tree as keys. Here, if the size of a label is sufficiently small (e.g., within 64 bits), the label value itself is kept in the data part of a B^+ tree called LT, which is stored on the secondary storage. Otherwise, it is stored in a separate disk file called LF and its reference in LF is kept in the data part of LT. An identifier reflecting the document order is provided to each generated label. The identifier is stored on LT as a key with its associated label value or reference to the label value in the data part; the identifiers are arranged in document order on the sequence set of LT. When a large size label is retrieved, being the specified identifier as a key, LT is searched to find the reference to it.

On the other hand, in our history-offset scheme, although label size is always kept fixed in short, *os* table occupies considerably large storage if the number of children is large as will be stated in Section 6.2.1. Therefore similar to the label handling of the other schemes, if the size of *os* table is large, it is stored in the separate disk file called OSF and its reference is held in the data part of RDT.

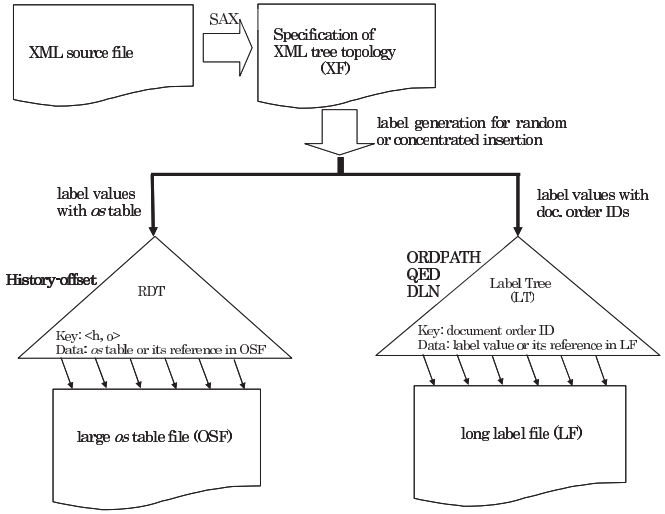

Fig. 10 Relationship among XF, LT, LF and OSF.

If the size is sufficiently small, the *os* table itself is held in the data part.

The relationship among XF, LT, LF and OSF is shown in **Fig. 10**.

6.2 Label Size Evaluation

In this section, storage costs in both random insertion, in which nodes are inserted at randomly chosen positions, and concentrated insertion stated in Section 6.1.2 are evaluated.

6.2.1 Random Insertion

In random insertion, XML tree continues to grow by selecting randomly one of the child nodes of a context node as described in Section 6.1.2.

The maximum label size, average label size and total label size in each labeling scheme are shown in **Fig. 11**. Note that in our history-offset method, the maximum and average label size are the same since the length of a label <history value, offset value> should be fixed. We can observe that the label size in our scheme is less than the maximum size in the other schemes in every XML tree, and ORDPATH and our scheme are superior to the other schemes in the average and total label storage size.

In our history-offset labeling scheme, each non-leaf node maintains a list of

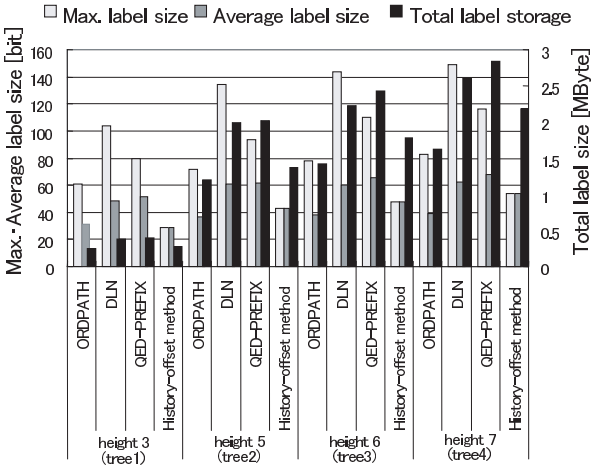

Fig. 11 Label size in random insertion.

subscripts in *os* table to preserve the document order among its children. If the number of the children of a node is n, the storage cost of the *os* table including the subscript of the first child is:

$$\lceil (n+1) log_2(n+1) \rceil.$$

The total storage cost of the *os* tables for all non-leaf nodes in the XML tree is dependent only on the final topology of the XML tree after node insertions and not dependent on the order of the node insertions.

For our history-offset labeling scheme, in addition to the total label storage cost in Fig. 11, extra storage for *os* tables and HOMD tables including history tables and coefficient tables described in Section 2.1 would be included. **Figure 12** shows the overall storage cost including these kinds of extra storage cost. We can observe that the cost of HOMD tables is very small, the cost of *os* tables is lower than that of the total label storage cost, and the overall storage cost is significantly higher than those of the other schemes shown in Fig. 11.

6.2.2 Concentrated Insertion

In concentrated insertion, XML tree continues to grow by selecting one of the child nodes of a context node in concentrated order.

Figure 13 shows the storage size in the labeling schemes other than our

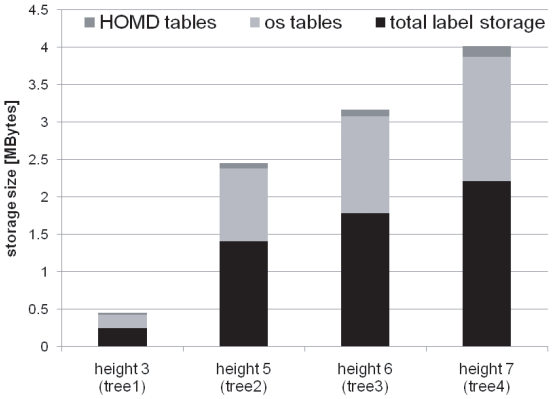

Fig. 12 Overall storage cost for history-offset scheme.

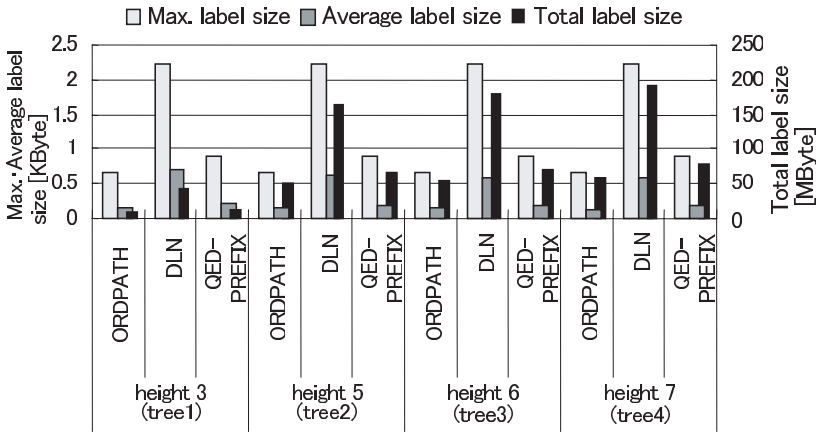

Fig. 13 Label size in concentrated insertion.

history-offset scheme after concentrated insertion. The size in our scheme is almost the same as those of the random insertion shown in Fig. 11. In general the storage cost for generated labels in history-offset scheme is constantly small irrespective of the order and the positions of node insertions. In the labeling schemes other than the history-offset scheme, as can be observed in Fig. 13, the three kinds of label size in concentrated insertion is extremely increased compared with those of random insertion shown in Fig. 11. On the contrary, those in our scheme do not increase and are nearly constant, because the label size in our scheme is heavily dependent on the shape of XML tree regardless of the dynamic insertion order. Moreover, size of the *os* table of a node is also unchanged since its size depends only on the number of its children rather than the insertion order.

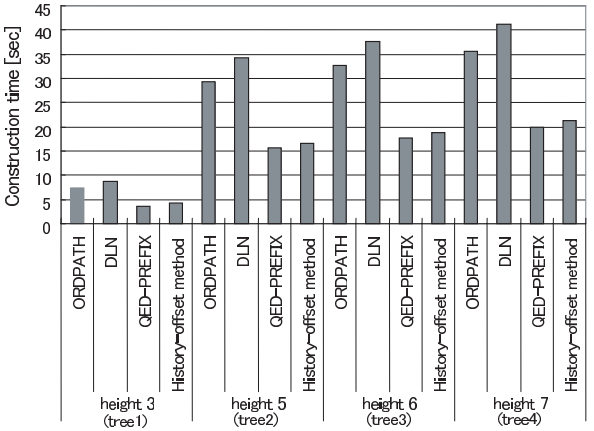

Fig. 14 Construction time.

6.3 Construction Time

For each labeling scheme, **Fig. 14** shows construction time of XML trees constructed by concentrated insertion using XF file described in Section 6.1.1.

As can be seen, our scheme is better than DLN and ORDPATH, and a little worse than QED-PREFIX. DLN and ORDPATH have much larger construction time, because the number of components in a node label delimited by separators (such as '.' and '/' in case of DLN) becomes large, so the number of bit operations required in encoding to binary bit string label value would increase.

On the contrary, QED-PREFIX has smaller construction time. In QED-PREFIX, although size of QED code encoded for each component of a node label would increase, the number of components in a node label is constant irrespective of the insertion order. So the number of bit operations required in encoding to binary bit string label value would be smaller than DLN and ORDPATH.

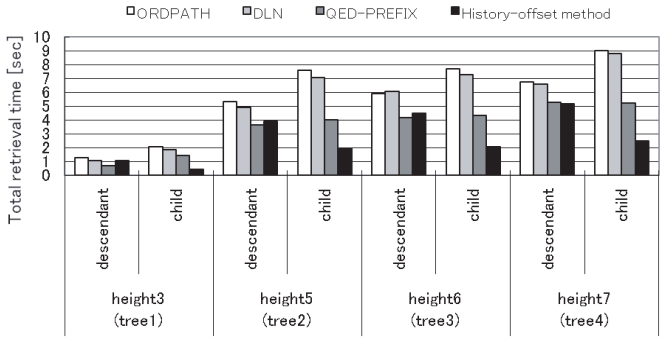

Fig. 15 Query performance of child and descendant axis (concentrated order).

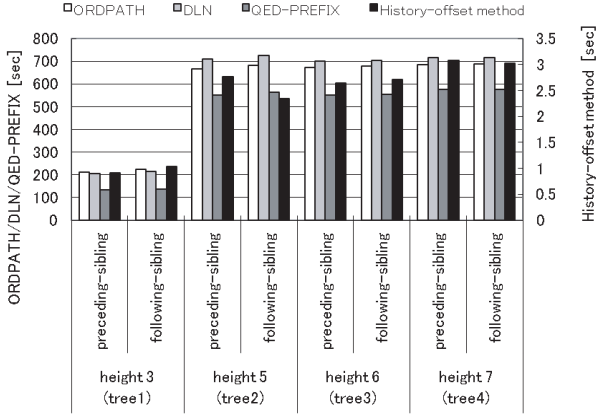

Fig. 16 Query performance of sibling axis (concentrated order).

6.4 Query Performance

For each tree in Table 1, 10% of nodes in the tree are randomly selected as context nodes, and for these context nodes the total sum of the retrieval times are measured for four kinds of axis (child, descendant, preceding-sibling and following-sibling) respectively. For the concentrated insertion order, **Fig. 15** shows the total sum of the retrieval times for child and descendant respectively, and **Fig. 16** shows the sum for preceding-sibling and following-sibling respectively.

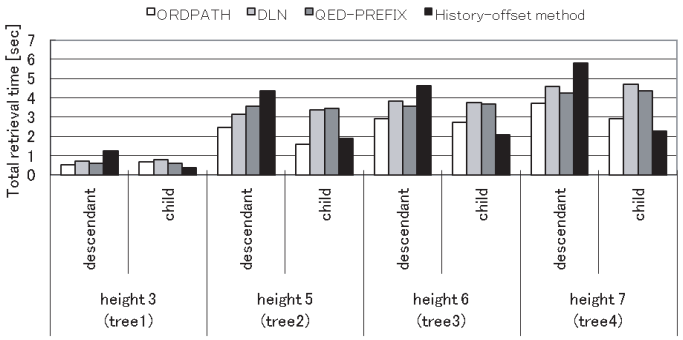

Fig. 17 Query performance of child and descendant axis (random order).

As can be seen, our scheme outperforms the other ones in these axes, especially in sibling axes. This is because retrieval along sibling axis in the other schemes needs to traverse the sequence set of LT B^+ tree explained in Section 6.1.1, which is arranged in document order, to find the label of the next sibling. On the other hand in the history-offset scheme, labels can be directly computed by referring to *os* table. To eliminate this disadvantage of the other schemes, an appropriate index should be created with additional space cost.

Figures 17 and **18** show the query performance in the case of trees constructed by random insertion order. From Fig. 18 we can see that the retrieval times of ORDPATH are much less than those of DLN and QED-PREFIX.In random insertion order, as can be seen from Fig. 11, label sizes of ORDPATH are rather less than those of the other schemes; even in the tree of height 7, the average label size is within 40 bits. The reason of the superiority of ORDPATH is in that most of the labels in ORDPATH are kept on the data part of the label tree LT and the external file LF is scarcely referenced. Actually, the frequency of the references to LF in ORDPATH is under 0.12% of those in DLN and QED. In the history-offset scheme, the label size in the same tree is always 53 bits and smaller than 64 bits, so every label is on the key part of RDT, but *os* tables in the external file OSF are often referenced.

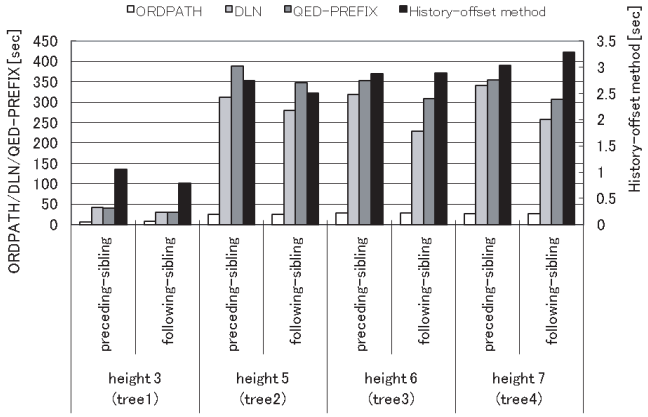

Fig. 18 Query performance of sibling axis (random order).

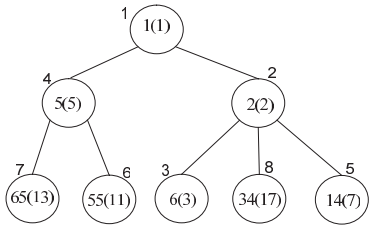

Fig. 19 Top-down labeling with PRIME.

6.5 Comparison with Prime Number Labeling Scheme

As was mentioned in the introductory section, like *os* tables of our history-offset scheme, the prime number labeling scheme (abbreviated as PRIME hereafter) in Ref. 5) employs an auxiliary table named *SC table* in order to determine the document order among XML tree nodes. PRIME takes advantage of the unique property of prime numbers to support order-sensitive queries for dynamic XML trees.

Figure 19 illustrates an example of top-down node labeling with PRIME. Each node is given a unique prime number and the label of each node is the product of its parent node label and its own prime number named *self label*, which is the next prime number supplied by the prime number generator at new node

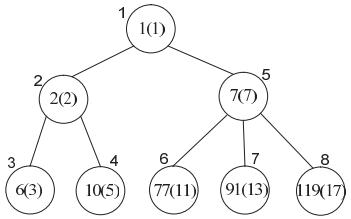

Fig. 20 Labeling in preorder with PRIME.

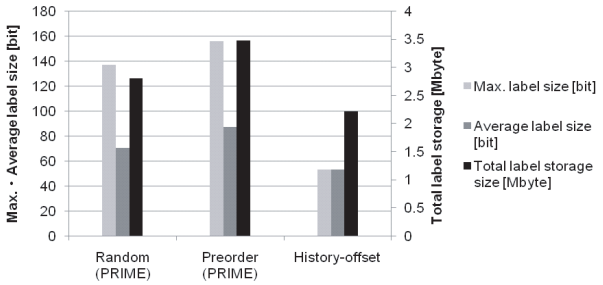

Fig. 21 Label storage costs in PRIME and History-offset (tree4).

insertion. In the figure, the number inside a node represents the node label with its self label in parentheses. The number outside a node represents the insertion order. As can be observed in Fig. 19,

(a) When the XML tree becomes taller and the path length of the inserted node from the root node becomes longer, the label size of the inserted node would become large.

(b) Label itself of a node does not carry the document order information of the node like in the history-offset scheme.

6.5.1 Label Storage Size

In PRIME scheme, assume that for a newly inserted node, a prime number is generated and assigned as its self label in ascending order. If an XML tree grows in preorder as in **Fig. 20**, larger prime numbers are assigned to the nodes on the lower level of the XML tree. In this situation, both of the average label size and total label storage size of the inserted nodes tend to be large. On "tree 4" shown in Table 1, **Fig. 21** shows the maximum and average label size, and total label

storage size in the following three kinds of node labeling. Note again that the storage cost for generated labels in the history-offset scheme remains constant irrespective of the node insertion order.

(1) Labeling in preorder insertion with PRIME scheme
(2) Labeling in random insertion order described in Section 6.1.2 with PRIME scheme
(3) Labeling with the history-offset scheme.

From Fig. 21, we can see that the label size and the total label storage cost in PRIME are worse than our history offset scheme. It is noted that the label size in PRIME is greater than 128 bits, which means that the arithmetic operation such as multiplication or division on label values should be performed by software.

6.5.2 Auxiliary Storage Size

In order to overcome the drawback of PRIME scheme stated in (b) above, the scheme provides an auxiliary data structure, which retains the *document ordering number* of a node. This data structure consists of *SC table* and *order number table*. Using both of the tables, the document order of an arbitrary node can be known. Reference 5) should be referred to for the detail. On "tree4" shown in Table 1, **Fig. 22** shows the overall storage cost in each labeling described in the previous subsection. Note that the size of both *SC table* and *order number table* is proportional to the total number of nodes in the XML tree.

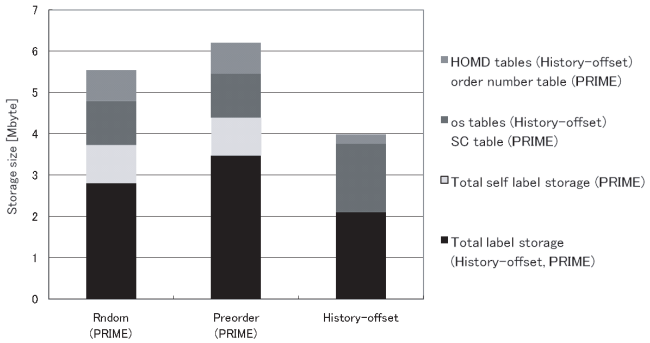

Fig. 22 Overall storage costs in PRIME and History-offset (tree4).

6.5.3 Query Performance

In PRIME, cost of a new node insertion is considerably high. If a new node of document order number n is inserted into an XML tree with total N nodes, the following should be performed. First, by searching *order number table*, order numbers greater than or equal to n are incremented by one. Then the SC values corresponding to the incremented order numbers are recalculated. On the other hand, in our history-offset scheme, the update of *os* table is limited to that of the parent node. As for query performance, in the context of our experiment described in Section 6.4, the following situation in PRIME scheme would considerably deteriorate the query performance:

(1) Label size tends to be larger than that of the history offset scheme, and the arithmetic operations on label values should often be performed by software for a large scale XML document.
(2) Range of the labels to be searched on the sequence set of the label tree is larger than the other schemes, since the label value itself doesn't carry document order information.

In respect to (2) above, for example, in retrieval of the sibling axis of the context node's label lc, first the label of the parent's node has to be found in the label tree, then a label value l on the sequence set is checked to see whether it is a sibling node's label of the context node. The checking is repeated until the end of the sequence set. If l is divisible by the parent node's label value and the quotient is a prime number, l can be judged as a sibling node's label. If l is judged as a sibling node's label, the judgment whether it is a preceding sibling or a following sibling can be done by knowing the document order number which is the remainder of SC value divided by the self label of the node.

In fact, the greater part of the retrieval time in PRIME is consumed by the multiple-precision arithmetic operations such as multiplication and division performed by software. If the total number of nodes N becomes larger, label values also become larger and the arithmetic operations consume much time. In our experiment, even in the tree1 in Table 1 whose height is only 3, the retrieval time for any axis is prohibitively larger than the other schemes due to the multi-precision arithmetic operations with SC value exceeding 64 bits.

6.6 Discussion

In our history-offset labeling scheme, each non-leaf node maintains a list of subscripts named *os* table to preserve the document order among its children. Actually, the overall storage cost including those of the total label storage, *os* tables and HOMD tables is dependent heavily on the topology of the XML tree and is not dependent on the order of node insertions due to the existence of the *os* tables. The experimental results shown in Fig. 11, Fig. 13 and Fig. 12 indicate that in the random insertion order the overall storage cost is greater than those of the other three kinds of labeling scheme, but the cost is far less than them in the concentrated insertion order.

On the other hand, in ORDPATH, DLN and QED-PREFIX, the label value itself represents its relative position in the document order, and for two arbitrary nodes in an XML tree, their ordering in the XML document can be determined by directly comparing the binary string label values in the lexicographical order with no additional information like *os* table in our scheme. But, in these schemes, if new nodes are repeatedly inserted around the same position as in Fig. 9 for example, the label value length of the inserted nodes will become very long as is shown in Fig. 13. This means that the direct comparability of the document order without additional information in these schemes can be achieved by the penalty of label size increase to carry the ordering information in the label themselves. On the contrary, in our history-offset scheme, as was described in Section 4.3, traversing the related *os* tables is necessary to determine the order between two arbitrary node labels.

As for time cost, our scheme can retrieve the sibling node labels quickly by traversing *os* table. On the contrary, the other schemes like ORDPATH, in which labels on the sequence set of LT are arranged in document order, should check through all the node labels in the range of descendant of its parent node on the sequence set. Hence, the retrieval of the sibling node labels is much more time consuming than our scheme as is shown in Fig. 16.

Another important limitation of our scheme includes the saturation of the history-offset space caused by excessive node insertions. **Table 2** shows the growing history value and offset value sizes for the trees in Table 1. In fact, when the maximum level of the XML tree exceeds 9, the history-offset space would

Table 2 History value and offset value sizes in the history-offset encoding.

tree	height	history value size H(bits)	offset value size O(bits)	label size H+O (bits)
tree1	3	13	18	31
tree2	5	13	30	43
tree3	6	13	35	48
tree4	7	13	40	53

be saturated if the space is the 64 bit machine word size. PRIME scheme also shares this kind of limitation. The situation of the scheme is more severe than our scheme. In the scheme, the label value is a product of prime numbers and should be compared as a numerical value, not as a bit string like in ORDPATH. But, if the XML tree grows higher, the maximum label size would quickly exceed the machine word size. Of course if we employ multiple-precision arithmetic operations performed by software, the saturation problem can be resolved, but the query performance would be significantly degraded.

One of the countermeasures against the label space saturation problem in our history-offset scheme is to divide the HOMD data structures into a set of lower dimensional HOMDs [15], and in PRIME scheme is to divide the XML tree into a set of smaller subtrees [5].

7. Conclusion

In this paper, we proposed a novel labeling scheme for dynamic XML trees by using *history-offset* encoding which is used for encoding multidimensional data. The scheme takes advantage from the encoding method by embedding an XML tree into a dynamically extendible multidimensional array. After describing our labeling scheme, label size, label storage cost and node query performance are examined compared with other competing schemes, and proves that our scheme outperforms these schemes in some criteria. Furture work includes the application of the technique proposed in Ref. 15) for the label space saturation problem in our history-offset labeling scheme.

Acknowledgments This work was supported in part by a grant-in-Aid for Scientific Research of Japan Society for the Promotion of Science in Japan (No.8200005).

References

1) Extensible Markup Language (XML). http://www.w3.org/XML/
2) Grust, T.: Accelerating XPath location steps, *Proc. 2002 ACM SIGMOD conference*, pp.109–120 (2002).
3) Cohen, E., Kaplan H. and Milo, T.: Labeling Dynamic XML Trees, *Proc. 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of database systems* (*PODS 2002*), pp.271–281 (2002).
4) Tatarinov, I., Viglas, S., Beyer, K., Shanmugasundaram, J., Shekita, E. and Zhang, C.: Storing and Querying Ordered XML Using a Relational Database System, *Proc. 2002 ACM SIGMOD conference*, pp.204–215 (2002).
5) Wu, X.D., Lee, M.L. and Hsu, W.: A Prime Number Labeling Scheme for Dynamic Ordered XML Trees, *Proc. 20th ICDE Conference*, pp.66–78 (2004).
6) Shimizu, T., Onizuka, M., Eda T. and Yoshikawa, M.: A Survey in Management and Stream Processing of XML Data, *Journal of IEICE Transactions on Information and Systems*, Vol.J90-D, No.2, pp.159–184 (2007).
7) Lee, Y.K., Yoo, S.J., Yoon, K. and Berra, P.B.: Index Structures for structured documents, *Proc. 1st ACM international conference on Digital libraries*, pp.91–99 (1996).
8) Meier, W.: eXist: An Open Source Native XML Database, *Proc. Web, Web-Services, and Database Systems*, pp.169–183 (2002).
9) Meier, W.: eXist. http://exist.sourceforge.net/
10) T. Sato., Satomoto, T., Kobata, K. and Pan, T.H.: Numbering Scheme which Identifies Tree Structures, *Proc. DEWS2002 conference*, A4-8 (2002).
11) O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G. and Westbury, N.: ORDPATHs: insert-friendly XML node labels, *Proc. 2004 ACM SIGMOD conference*, pp.903–908 (2004).
12) Li, C.Q. and Ling, T.W.: QED: A novel quaternary encoding to completely avoid re-labeling in XML updates, *Proc. 14th ACM International Conference on Information and Knowledge Management*, pp.501–508 (2005).
13) Bohme, T. and Rahm, E.: Supporting Efficient Streaming and Insertion of XML Data in RDBMS, *Proc. Third International Workshop on Data Integration over the Web* (*DIWeb*), pp.70–81 (2004).
14) Hasan, K.M.A., Tsuji, T. and Higuchi, K.: An Efficient Implementation for MOLAP Basic Data Structure and Its Evaluation, *Proc. 12th International Conference on Database Systems for Advanced Applications* (*DASFAA'07*), pp.288–299 (2007).
15) Tsuji, T., Kuroda, M. and Higuchi, K.: History offset implementation scheme for large scale multidimensional data sets, *Proc. 2008 ACM Symposium on Applied Computing* (*SAC2008*), pp.1021–1028 (2008).
16) Otoo, E.J. and Merrett, T.H.: A storage scheme for extendible arrays, *Journal of Computing*, Vol.31, pp.1–9 (1983).
17) Otoo, E.J. and Rotem, D.: Efficient Storage Allocation of Large-Scale Extendible Multi-dimensional Scientific Datasets, *Proc. 18th International Conference on Scientific and Statistical Database Management*, p.179–183 (2006).
18) XML Path Language (XPath). http://www.w3.org/TR/xpath
19) XMark — An XML Benchmark Project. http://monetdb.cwi.nl/xml/
20) The Simple API for XML (SAX). http://www.saxproject.org/

(Received September 19, 2009)
(Accepted January 4, 2010)

(Editor in Charge: *Takeo Kunishima*)

Bei Li is currently a candidate of Ph.D. student of Graduate School of Engineering at University of Fukui, Japan. She received her B.E. and M.E. degrees from Xi'an University of Technology and University of Fukui, in 2005 and 2007 respectively. Her research interests include document processing systems.

Katsuya Kawaguchi graduated master course of Graduate School of Engineering at University of Fukui in 2009, and presently with Toshiba Solutions Corporation .

Tatsuo Tsuji received his Ph.D. degree in Information and Computer Science from Osaka University in 1978. In the same year, he joined the Faculty of Engineering at University of Fukui in Japan. Since 1992, he has been a professor in the Information Science Department of the faculty. His current research interests include database implementation schemes, data warehousing systems and document processing.

Ken Higuchi received his B.E., M.E. and D.E. degrees in Communications and Systems Engineering in 1992, 1994 and 1997 respectively from the University of Electro-Communications. He is now an Associate Professor of the Graduate School of Engineering, University of Fukui. His research interests include database management system, parallel processing, and XML document management system.