A Two-Dimension XML Encoding Method based on Variable Length Binary Code

Jie Chen[1], Wenxin Liang[2*], Haruo Yokota[3]
[1,2] School of Software, Dalian University of Technology
[3] Tokyo Institute of Technology
Email: [1] chen@mail.dlut.edu.cn, [2] wxliang@dlut.edu.cn, [3] yokota@cs.titech.ac.jp

Abstract— **Recently, the researchers have proposed a number of labeling schemes. In these labeling schemes, the approach which can extract structural information between nodes and process query efficiently is more outstanding. However, most of these labeling schemes do not well support update operations. To achieve update-friendly operations, some of the methods keep intervals between labeling numbers, but it requires whole relabeling when the intervals are used up. Several labeling schemes support dynamic XML documents, but most of these labeling schemes allow only leaf node insertions. OrdPathX supports both leaf node insertions and internal node insertions. Inspired by the method of inserting internal nodes of OrdPathX and extending the C-DO-VLEI code, in this paper we propose two dimensions VLEI code. We discuss how this labeling scheme labels nodes and how we can get the structural information of nodes from their labels. We design experiments to evaluate the efficiency of producing labels, the storage consumption and the querying performance of two dimensions VLEI code we proposed, and compare those with the OrdPathX.**

Index Terms— **XML, Labeling Scheme, Performance Evaluation, Internal Node Insertion**

I. INTRODUCTION

Recently, XML has become a standard language for data representation and exchange over the Internet, and is more and more widely used in various applications. An XML document can be represented as a nested tree. Figure 1 shows an example XML document, which is used throughout this paper. Figure 2 is the corresponding XML tree. According to the different structures in the XML tree, nodes can be classified into three kinds of types: 1) element node, 2) attribute node and 3) text node.

In the past few years, researchers have proposed many labeling schemes. These labeling schemes can be divided into four categories, namely, sub-tree labeling, prefix-based labeling, multiplicative labeling and hybrid labeling [3], and researchers analyze how each approach works, as well as its advantages and disadvantages [3], such as Tree Traversal Order [2] and Dewey Order [9]. Tree traversal order is one of the interval encoding of sub-tree labeling. In this scheme, each node is labeled with a pair of unique integers consisting of preorder and postorder traversal sequences. This label scheme can determine the Ancestor-Descendant (A-D) relationship easily, but the Parent-Child (P-C) relationship can not be

```
<BOOK ISBN ="1-55860-438-3 ">
   <SECTION>
      <TITLE> Bad Bugs</TITLE>
           Nobody loves bad bugs.
      <FIGURE CAPTION ="Sample bug" />
   </SECTION>
   <SECTION>
      <TITLE> Tree Frogs </TITLE>
           All right-thinking people.
      <BOLD> love</BOLD>
         tree frogs.
   </SECTION>
</BOOK>
```

Figure 1. Example XML document.

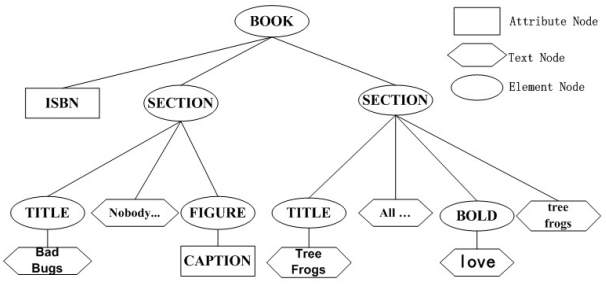

Figure 2. Example XML tree.

determined directly. Dewey Order uses delimiter "." to separate the label of an ancestor node at each level of the tree. In this method, each label of an ancestor node is a prefix of its descendants, the P-C, A-D and sibling relations can be determined by comparing the labels of two nodes. However, both of these two methods are unsuitable and inefficient for a dynamic XML document, because when a new node is inserted into the tree using these two methods, a large number of nodes need to be re-labeled, in order to remain the structural information.

To reduce the cost of insertion, in practice, the update-friendly labeling scheme is more useful. Vector Order [11], VLEI (Variable Length Endless Insertion) code [4], ORDPATH [8] and Extended Preorder Traversal [6] are all update-friendly labeling schemes. [12] compares Vector Order with some other update-friendly labeling schemes. In this paper, we only focus on the methods combined with Dewey Order. ORDPATH is one

*Corresponding author.

doi:10.4304/jsw.6.12.2426-2433

of the famous update-friendly labeling schemes based on the prefix-based labeling scheme Dewey Order. ORDPATH uses the variable-length bit string to represent labels. ORDPATH is conceptually similar to the Dewey Order described in [9]. In ORDPATH, only positive and odd numbers are assigned for the initial labeling, such as the code of 1.1.1.3 and 1.5. Even and negative numbers can be preserved for future insertions. In the actual encoding, labels are represented as the compressed ORDPATH format. Because in practical applications, many large XML documents need to be handled, therefore, it is necessary to ensure enough storage for extracting node's structural information and update operations. (DO-VLEI) code [**?**] and ORDPATH [**?**] have been proposed. The DO-VLEI labeling method inherits features of the Dewey Order method, but reduces the update cost for insertion operations. It uses the VLEI code [4] for expressing the sibling order by a unique magnitude relationship, which enables the unlimited insertion of new nodes with no relabeling of other nodes being required. Experiments in [7] indicates that the DO-VLEI code outperforms the ORDPATH in both structural information extraction and storage consumption.

However these update-friendly labeling schemes are allow only leaf node insertions, if the intermediate node is inserted, the methods cannot guarantee there is no need to recalculate the labels of large number of nodes. In practical applications, it is no doubt that the situations of inserting intermediate nodes are possible, therefore [1] proposed a two-dimensional encoding method called OrdPathX. OrdPathX is a two-dimensional labeling scheme based on the "careting-in" technique of ORDPATH. It can handle both internal and leaf node insertions efficiently. Because one-dimensional VLEI code outperforms ORDPATH, we want to design a two-dimensional VLEI code which support intermediate node insertion based on one-dimensional VLEI code. According to the analysis of advantages and disadvantages of various labeling schemes, we design experiments to compare the two-dimensional VLEI code with OrdPathX, in order to prove whether the two-dimensional VLEI code outperforms OrdPathX. Experiments show that the two-dimensional VLEI code we proposed outperforms OrdPathX in both structural information extraction and storage consumption.

In this paper, we mainly introduce the two-dimensional VLEI code and discuss how the proposed labeling scheme labels nodes and how to handle the issue of re-labeling when inserting intermediate nodes in the vertical dimension. Then we compare the two-dimensional VLEI code with OrdPathX in the following three respects:

1) The label construction speed.
2) Whether we can get node's information from the generating label efficiently, the information concludes node depth information, the node labels and node names of the parent node and the ancestor node.
3) The average length of the labels, which determines whether the labeling scheme saves storage space.

Experimental results show that the two-dimensional VLEI code outperforms OrdPathX in these three respects, it is mainly because the compressed two-dimensional VLEI code's length is shorter than the compressed OrdPathX, thus reducing storage consumption, and compress efficiently. Because in OrdPathX when compressing and decoding each component must refer to the prefix schema and even-numbered and negative integer component values are reserved for later insertions, so it is slower than the two-dimensional VLEI code which does not use the prefix schema in the label construction and the structural information queries.

The remainder of the paper is organized as follows: Section II introduces related researches. Section III describes our proposed two-dimensional VLEI code. Section IV shows the method for extracting useful structural information from the two-dimensional VLEI code. Section V describes the experiments and evaluation of our proposed labeling method. Section VI concludes the paper.

II. Related Work

Dewey Order is a simple prefix-based labeling scheme, and many other labeling schemes are based on Dewey Order, Dewey Order [9] uses "." delimiter to separate its parent label and its own label, It is defined as follows:
1. The label of the root node $C_{root} = 1$.
2. The label of a non-root node C= $C_{parent}.C_{child}$, where C_{parent} denotes the label of its parent node, C_{child} denotes the order of the node in the sibling node.

We can get node's depth information, the label of its parent node and ancestor nodes and sibling (preceding or following) relationship from the label. But Dewey Order is not an update-friendly labeling scheme, therefore, researchers proposed ORDPATH [8] and DO-VLEI [4], and both of the two schemes are based on the Prefix-based labeling scheme Dewey Order. The two methods do not have re-labeling problem after insertion. We will introduce the two methods next.

A. ORDPATH

ORDPATH is implemented in Microsoft®SQL $Server^{TM}$ 2005, and is used in the added attribute HierarchyID in the latest release of Microsoft®SQL $Server^{TM}$ 2008. ORDPATH is an update-friendly labeling scheme, based on Dewey Order, in Dewey Order, node's label is made from the label of the parent node, a "." delimiter and a brother codes. In ORDPATH, it is similar to Dewey Order during the initial labeling, but only positive, odd integers are assigned. Figure 3 is an XML tree labeling by ORDPATH, even number and negative integer component values are reserved for further node insertions. For example, when inserting a node between nodes labeled "1.3.1" and "1.3.3", the new node will be labeled by "1.3.2.1", in which "2" is a placeholder not increasing the depth of node, is assigned to the node. The node of "1.3.2.3" is inserted between

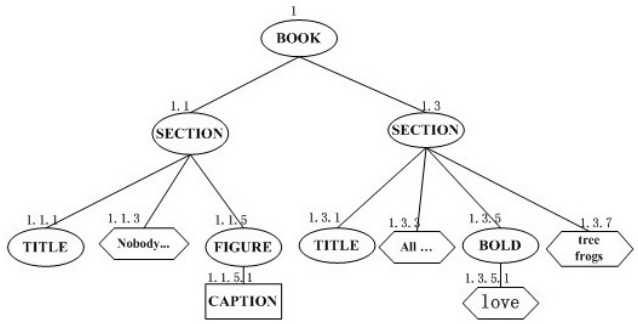

Figure 3. Labeling by ORDPATH.

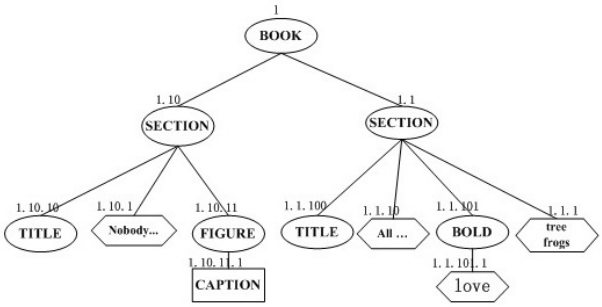

Figure 4. Labeling by DO-VLEI code.

nodes labeled "1.3.2.1" and "1.3.3". ORDPATH can support update operations without relabeling. If given the label of a node, its parent nodes, ancestor nodes and depth information can be got, and the order of nodes also can be quickly got by comparing the labels.

Note that the dots in the label are just for readers' easy understanding. In the actual encoding, we use the variable-length bit string to represent labels, called C-ORDPATH. In the C-ORDPATH, label is expressed as successive variable-length Li/Oi bit strings, and each Li/Oi is used to represent an integer. according to a prefix schema [8]. Li represents the length of Oi, and it is a prefix-free encoding. Oi represents the value of the component. For example, "1.3.1" is compressed into 0110101 according to the prefix schema in [8]. Because 1 is represented as 01(01 is Li, and it represents the length of Oi is 0, so 1 does not have Oi), and 3 is represented as 101(10 is Li, and it represents the length of Oi is 1, so the Oi of 3 is 1 according to the prefix schema), "1.3.1" is represented as 0110101.

B. DO-VLEI Code

DO-VLEI [4] is the labeling scheme combining VLEI code with Dewey Order. We will introduce VLEI code first.

VLEI code is a variable-length bit string starting with 1 and is composed by 0, 1, the relationship between codes satisfies the following definition.

Definition 1: the relationship between VLEI codes: v is a VLEI code, and the following condition is satisfied: $v \cdot 0 \cdot \{0|1\}* < v < v \cdot 1 \cdot \{0|1\}*$

The new code will be smaller than the original label if 0 is added behind a VLEI code. The new code will be larger than the original label if 1 is added behind a VLEI code. So that there is no need to change other nodes' labels when inserting a new node and label-generating is also easy. The method combining VLEI code with Dewey Order is DO-VLEI code. It is defined as follows:

1. The DO-VLEI code of the root node $C_{root} = 1$.

2. The DO-VLEI code of a non-root node C= $C_{parent}.C_{child}$, where C_{parent} denotes the DO-VLEI code of its parent and C_{child} denotes the order of the node in the sibling node.

Figure 4 is an example XML tree labeling by DO-VLEI code. Each component in DO-VLEI code is a VLEI code, separated by ".". When generating each component in DO-VLEI code, we have to use an algorithm [5] that mapping a natural number to VLEI code. The algorithm is given for generating each component of node when labeling the label of each node during the initial XML tree load.

We need to use the algorithm in [5] when inserting nodes. The algorithm is about how to decide the VLEI code when inserting nodes. When inserting a node v between two sibling nodes the VLEI code are v_l and v_r (v_l is the left sibling, v_r is the right sibling, $v_l < v_r$) respectively, then it's VLEI code is decided like this: If the length of v_l is greater than the length of v_r, the VLEI code of v is v_l1. Otherwise, if the length of v_l is less than or equal to the length of v_r, the label of v is v_r0. So the VLEI code can ensure inserting nodes without changing other nodes' labels and the insertion label is unique. In the DO-VLEI code, insert operation effects only the last component, the head components only need to inherit the DO-VLEI code of the parent node.

C. C-DO-VLEI Code

The same as ORDPATH, in order to save storage space, in the actual storing, we need to compress DO-VLEI code, and the encoding scheme called C-DO-VLEI code [7]. A DO-VLEI code is composed of three elements: ".1", "0" and "1", at most 2 bits can represents three different types of elements. ".1", "0" and "1" are represented by 10, 0 and 11 respectively when compressing DO-VLEI code. This compressed code called C-DO-VLEI Code. For example, 1.10 .11 is compressed into 111001011. 0, 10, and 11 are prefix codes, so that C-DO-VLEI codes can be uniquely decoded into the original DO-VLEI codes. Scan C-DO-VLEI Code front to back when decoding, look at the number of consecutive 1s before the 0. If there is an odd number of consecutive 1s appearing before 0, the last two bits 10 represent .1.Otherwise, if there is an even number of consecutive 1s before 0, all of 1s are decoded into 1, and the final "0" is decoded into 0. We can see that in compressing and decompressing DO-VLEI code is easier than ORDPATH, it doesn't require any prefix schema, and it has already been proved in [7] that C-DO-VLEI code outperforms C-ORDPATH in the storage consumption and query performance.

D. OrdPathX

Both ORDPATH and DO-VLEI Code are update-friendly labeling schemes. It does not require relabeling for other nodes when inserting leaf node, but they cannot handle internal node insertion effectively. OrdPathX [1] can handle both internal and leaf node insertions efficiently. In OrdPathX, the label consists of an Augmented OrdPath(AO), possibly followed by a Parent Height (PH):$OrdPathX = AO.PH$, where an AO is a sequence of chunks and each chunk is consisted of zero or more even components followed by an odd component. Between each pair of consecutive chunks there may exist an Incremental Height (IH). OrdPathX is inspired by the "careting-in" technique of ORDPATH. In OrdPathX, it is the same as ORDPATH during the initial load, and all of the labels of nodes don't have IH and PH. The label of node v is $L.C$, and the label of its parent node v' is L. Now we are to insert nodes u_l and u_2 between them. First, consider the insertion of u_1, the label of u_1 is $L.(1).C$ where "(1)" is an IH. Moreover, v relabeled as $L.C.[1]$ where "[1]" is the PH of v's label. It indicates that v has a parent-insertion node above it. We can see that if a node's label has PH, a node has been inserted as its parent, and its PH value is equal to the IH value of its parent node's label. Then insert u_2 as the new parent of u_1. The label of u_2 is $L.(3).C$ where (3) is the IH value of u_2, so u_1 should be relabeled as $L.(1).C.[3]$.

In OrdPathX, the use of dots and brackets in the labels are just for readers' easy reading. In the actual encoding, label is represented as the form of variable-length bit string. Each component is encoded using the $IiLiOi$ format where Ii is the component type indicating whether the component is an ORDPATH, IH or PH. Ii uses a fixed size of 2 bits to represent since there are three different types of label components. The compression of $LiOi$ is the same as ORDPATH. We can use 01 to represent IH, 10 to represent PH and 00 to represent ORDPATH. We can see that in OrdPathX both compressing and decompressing need to refer to the prefix schema, and the selection of the prefix schema is also important.

III. 2-D VLEI CODE

In this section, we propose 2-D VLEI Code based on DO-VLEI Code [4]. It not only supports leaf node insertion but also supports internal node insertion.

A. Labeling Method

Because DO-VLEI Code and ORDPATH are both based on Dewey Order, they are very similar in leaf node insertion. The 2-D labeling scheme OrdPathX based on ORDPATH can handle internal node insertion, and can extract the structural relationship between nodes from labels. Therefore, according to the method that OrdPathX handles internal node insertion, we proposed 2-D VLEI code based on one-dimensional VLEI code to achieve internal node insertion efficiently.

The label in OrdPathX consists of an Augmented OrdPath(AO), possibly followed by a Parent Height (PH):

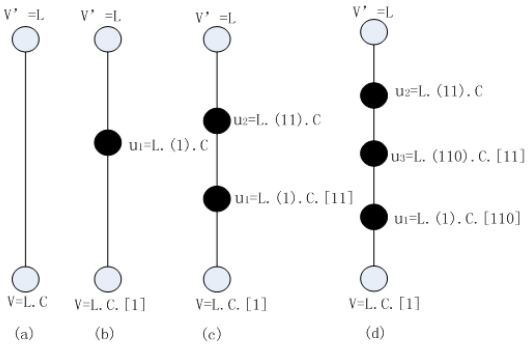

Figure 5. An example of internal node insertion by 2-D VLEI code.

$OrdPathX = AO.PH$. The 2-D VLEI code we proposed is based on OrdPathX, so there are IH and PH in 2-D VLEI code, but IH and PH here are components, not chunks.

B. Internal Node Insertions

In 2-D VLEI code, all the labels is the same as DO-VLEI Code during the initial load, and all of the labels of nodes don't have IH and PH, and use the same method inserting leaf nodes. Therefore, this section focuses on the method of internal node insertion.

The label of node v is $L.C$, and the label of its parent node v' is L (Figure 5(a)). Now we are to insert nodes u_1, u_2, and u_3 between them. First, consider the insertion of u_1, We label u_1 as $L.(1).C$ where "(1)" is an IH, as shown in Figure 5(b). The value of IH is the first VLEI code 1 according to the insertion method of VLEI code, and then v is relabeled as $L.C.[1]$ where $[1]$ is a PH. Next, insert u_2 between u_1 and v', we label u_2 as $L.(11).C$ and also re-label u_1 as $L.(1).C.[11]$, as shown in Figure 5(c). Finally we insert node u_3 between u_1 and u_2. Because the IH of u_1 and u_2 are (1) and (11) respectively, the IH of u_3 is the value (110) which is bigger than (1) and smaller than (11) according to the insertion method of VLEI code. We label u_3 as $L.(110).C.[11]$ and re-label u_1 as $L.(1).C.[110]$, as shown in Figure 5(d). And this IH is the Significant Incremental Height (SIH) because the component's presence implies that the node is a parent-insertion, similar as the conception of SIH in [1].

There is a special situation of leaf node insertion. If v is a parent-insertion node, we cannot handle the insertion as the normal method, because v has a child, say w, which was present before v was inserted. The label of v is $L.(IH).C.[PH](PH$ can be empty). If u is inserted on the left of w, we label u as $L.(IH).C.LS$ (Left Sibling, the new node is the left brother of the present node w). An example is shown in Figure 6. If we insert a leaf node as the child of node $L.(1).1$ on the left of the node whose label is $L.1.[1]$, the new node's label is $L.(1).1.1$. If u is inserted on the right of w, u is labeled as $L.(IH).C.D$. In Figure 6, the new node is labeled as $L.(1).1.1$. Assuming that the new child node is v', if

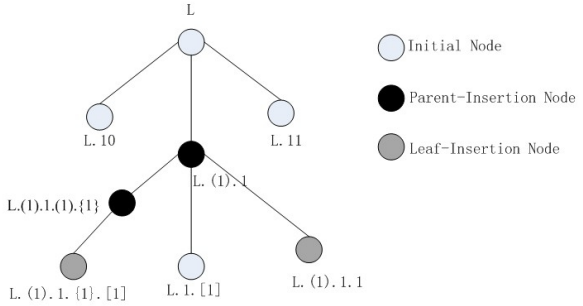

Figure 6. A special example of leaf node insertion.

we are to insert node u between v and v', the new node is labeled as $L.(IH).C.(IH).LS$ or $L.(IH).C.(IH).D$, the PH value of v is equal to the last IH value of u. If we insert a node between the node $L.(1).1.1$and the node $L.(1).1$, the new node is labeled as $L.(1).1.(1).1$, the leaf node is relabeled as $L.(1).1.1.[1]$,as shown in Figure 6.

C. Structural Information

We can get the inter-node relationships between two nodes from their 2-D VLEI code (namely, P-C, A-D and siblings (preceding sibling and following sibling)):

P-C relation: Given two nodes u and v, if we want to verify whether u is v's parent node, we just see if the $L.(IH).C$ part of v's parent matches the $L.(IH).C$ part of u's label. We can deduce the $L.(IH).C$ part of v's parent as follows. If ph is non-empty, then the $L.(IH).C$ part of v's parent is $L.(ph).C$. Otherwise, if ph is empty, then the $L.(IH).C$ part is L. Note that it is impossible to deduce the PH component of the parent.

A-D relation: Given two nodes u and v, if we want to verify whether v is u's ancestor node, we consider two cases: (1) If v's label is a prefix of u's label, then v is an ancestor of u. (2) If u and v have the same sequence of components except for the SIH and PH and the SIH of v is lexicographically larger than the SIH of u, then v is an ancestor of u.

Sibling Relation: To determine if two nodes u and v are siblings, we check if they have the same parent. If we want to detect preceding/following sibling relation, we consider two cases: (1) If both u and v do not have a PH in their labels, we just compare their last component in lexicographic order. (2) If either u or v has a PH component, then the node (say v) with a PH component has one or more nodes inserted above it. If there is LS component in u's label, u precedes v. Otherwise, u follows v. The node u precedes v if the LS component of u is less than the LS component in v when both of the nodes have LS.

D. Compressed 2-D VLEI code

In DO-VLEI Code, ".1", "0" and "1" are compressed into (10), (0) and (11) respectively. In the 2-D VLEI Code, there are PH, IH, LS and normal VLEI code components, so we need 2 bits to separate 4 types of components. We use (01) to represent PH, use (10) to represent IH, use (11) to represent LS and use (00) to represent normal VLEI code. The question is where we should put the two bits that will not cause conflict in the future decoding. We put these two bits after the (10) which represents ".1" of DO-VLEI code. If we want to decide the type of component when decoding, we should scan two bits after we determine the position of "." and decide the type of component. Algorithm 1 shows the details of compressing 2-D VLEI Code. For example, 1.100.101.110.1 is compressed into 11100000100001110001101000. Because all of the DO-VLEI codes begin with 1 and are normal VLEI Code, the first component does not use additional two bits to represent the type of component. $1.(11).100.11.[1]$ is compressed into 111010111000001011111001. The first two bits of each component are 10(except the first component), and the two bits following represent the type of component. Therefore, the compressed 2-D VLEI code can be uniquely decoded into the original 2-D VLEI codes.

Algorithm 1 The Algorithm of compressing 2-D VLEI Code.

Input:
The label of 2-D VLEI Code before Compressing, $vlei$;
Output:
The Compression of 2-D VLEI Code, $cvlei$;

```
for i = 0 to i < vlei.length() do
  if vlei.charAt(i) ==' 1' then
    cvlei = cvlei + "11"
  else {vlei.charAt(i) ==' 0' }
    cvlei = cvlei + "0"
  else {label.charAt(i) ==' .'&&label.charAt(i+1) ==' (' }
    cvlei = cvlei + "10" + "10"
    i = i + 2
  else {label.charAt(i) ==' .'&&label.charAt(i+1) ==' [' }
    cvlei = cvlei + "10" + "01"
    i = i + 2
  else {label.charAt(p) ==' .'&&label.charAt(i+1) ==' {'}
    cvlei = cvlei + "10" + "11"
    i = i + 2
  else {label.charAt(p) ==' .'}
    cvlei = cvlei + "10" + "00"
    i = i + 1
  end if
end for
return cvlei;
```

TABLE I.
EXPERIMENTAL ENVIRONMENT

CPU	$Pentium(R)DualCoreE5300(2.60GHz)$
Memory	DDR2 2048MB
OS	Windows XP Professional
Memory	DDR2 2048MB
Java	1.6.0_02
DB	Office Access 2003

IV. STRUCTURAL INFORMATION EXTRACTION USING 2-D VLEI CODE

In this section, we will discuss the method of extracting node's information from 2-D VLEI Code. Structural information includes depth information, the label and name of parent node, the label and name of any ancestor nodes. We can also decide the inter-node relationships (P-C relation, A-D relation and sibling relation) between two nodes according to their labels.

Because the 2-D VLEI Code is also prefix-based labeling scheme, it is important to find the locations and count the number of the delimiters in 2-D VLEI Code. If we get the number of the delimiters, we will know the depth of node, and we only need to extract the prefix code before the rightmost delimiter to get the label of parent node. We will get node's name after querying the table that storing the labels and names of all the nodes for each XML document. To get any level ancestor nodes information, we repeat the method of parent node information extraction. In Compression of 2-D VLEI code part of Section III, we have described the method of finding the location of the delimiters and decoding labels. If given two nodes' labels, we should decode labels fist and then decide their inter-node relationships according to the method described in Structural Information of Section III.

V. EXPERIMENTAL EVALUATION

Experimental environment is shown in Table 1. Because the 2-D VLEI Code and OrdPathX are both support internal node insertions, we perform experiment to compare their performance, such as the speed of producing the node's label, the storage consumption and the efficiency of extracting information from the label. The XML documents used for the experiments were sourced from the XML Data Repository [10], we select 13 XML documents, and Table 2 shows the details of these XML documents, including the document name, document size, and the number of elements, and the maximum depth. The XML document sizes range from 1KB to 1.7MB.

A. Efficiency of Producing Labels

First, to compare the 2-D VLEI code with OrdPathX in the efficiency of producing labels, we designed experiment to calculate the label of 2-D VLEI code and OrdPathX of all the nodes in XML document, at the same time record the time of computing labels of all nodes of a XML document. Figure 7 shows the label generation time ratio for computing labels of all nodes of a XML document, using the 2-D VLEI code and OrdPathX. From this figure, we can see that the execution times for computing labels of all nodes using the 2-D VLEI code are about 15% less than the execution times using OrdPathX. The main reason is that compressing each component in OrdPathX needs to find a suitable record in the prefix schema.

TABLE II.
XML DOCUMENTS

document(.XML)	size(byte)	elements	maxdepth
region	787	21	3
nation	4,568	126	3
ubid	20,320	342	5
321gone	24,516	311	5
yahoo	25,421	342	5
supplier	29,250	801	3
ebay	35,562	156	5
reed	283,655	10546	4
SigmodRecord	478,416	11526	6
customer	515,660	13501	3
part	618,181	20001	3
wsu	1,647,864	74557	4
mondial-3.0	1,784,825	22423	5

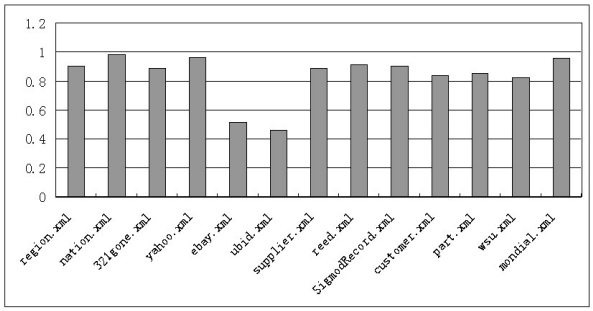

Figure 7. Label generation time ratio of 2-D VLEI code to OrdPathX.

B. Average Label Size

We then calculated and compared the average label size of each XML document labeled by the 2-D VLEI code and OrdPathX. Figure 8 shows the average label size ratio of the 2-D VLEI code to OrdPathX for each XML document and from which we can learn that the average label size using the 2-D VLEI code is about 15% on the average smaller than that using the OrdPathX. This is because that the compression of .1, 0 and 1 is as short as possible in 2-D VLEI code and then the total size of 2-D VLEI code will be the shortest. Moreover, in OrdPathX, skipping even numbers makes ORDPATH labels less compact, and the reserved space is not possible to be reduced, so this makes the compressed OrdPathX label longer. It means that the proposed labeling scheme also outperforms the OrdPathX in storage consumption.

C. Label Information Extraction

We also performed experiments to calculate the time of extracting information from the stored labels of each document. The information includes: 1) node's information (the depth, label after decoding, the node's name); 2) information of the parent node (the depth, label after

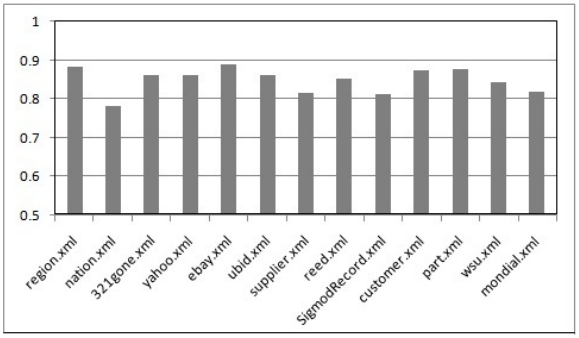

Figure 8. Label size ratio of 2-D VLEI code to OrdPathX.

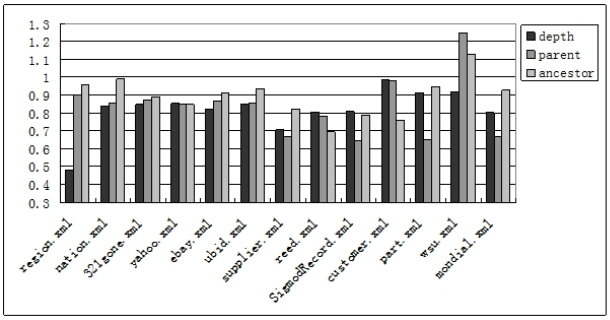

Figure 9. Execution time ratio of 2-D VLEI code to OrdPathX.

decoding, the node's name); 3) information of the ancestor nodes (the depth, label after decoding, the node's name). Figure 9 shows the execution time ratio for extracting self information, information of the parent node, and information of the ancestor nodes using the 2-D VLEI code and OrdPathX. From the experimental results we can see 2-D VLEI code can achieve high performance in structural information extraction. This is because the delimiter detection in OrdPathX requires traversal from the head through the whole code and refers to the prefix schema for determining each delimiter. While using the 2-D VLEI Code does not need to refer to any prefix schema, just counting the number of consecutive "1".

D. Efficiency of Inserting Labels

In the experiment, we performed parent insertions. We selected 30% of the total nodes in each XML documents and insert one parent for each selected node. Figure 10 shows the total insertion time ratio of 2-D VLEI code to OrdPathX, from which we can learn that the insertion time using 2-D VLEI code is about 47% on the average smaller than that using the OrdPathX, which means that the proposed method also outperforms the OrdPathX method in internal node insertion.

VI. Conclusion

Inspired by the method of inserting internal nodes of OrdPathX, in this paper we proposed 2-D VLEI code based on the DO-VLEI code. The 2-D VLEI code supports internal node insertions and does not need any prefix schema when compressing and decoding. We performed

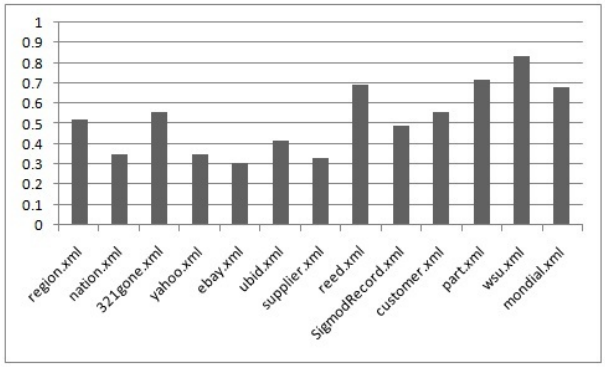

Figure 10. Insertion time ratio of 2-D VLEI code to OrdPathX.

experiments to evaluate the efficiency of producing labels, the storage consumption and the querying performance of 2-D VLEI code we proposed, and compared those with the OrdPathX. And we also compared the 2-D VLEI code with OrdPathX in internal node insertion. Our experimental results indicate that the 2-D VLEI code outperforms the OrdPathX in the above-mentioned four aspects.

Acknowledgment

This work was partially supported by SRF for ROCS, SEM, Doctoral Fund of Ministry of Education of China, the Fundamental Research Funds (DUT10JR02) for the Central Universities, China.

References

[1] Jing Cai and Chung Keung Poon. OrdPathX: Supporting Two Dimensions of Node Insertion in XML Data. In *Proceedings of DEXA*, pages 332–339, 2009.

[2] Paul Frederick Dietz. Maintaining Order in a Linked List. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 122–127, 1982.

[3] Su Cheng Haw and Chien Sing Lee. Node Labeling Schemes in XML Query Optimization: A Survey and Trends. *IETE TECHNICAL REVIEW*, 26(2):88–100, 2009.

[4] Kazuhito Kobayashi, Wenxin Liang, and Dai Kobayashi. VLEI code: An Efficient Labeling Method for Handling XML Documents in an RDB. In *Proceedings of ICDE*, pages 386–387, 2005.

[5] Kazuhito Kobayashi and Haruo Yokota. Evaluation of XML Laveling Methodss using Endless Insertable Code VLEI. In *DEWS2004*, 2004.

[6] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the VLDB*, pages 361–370, 2001.

[7] Wenxin Liang, Akihiro Takahashi, and Haruo Yokota. A Low-Storage-Consumption XML Labeling Method for Efficient Structural Information Extraction. In *Proceedings of DEXA*, pages 7–22, 2009.

[8] Patrick O'Neil, Elizabeth O'Neil, and Shankar Pal. ORDPATHs:Insert-Friendly XML Node Labels. In *Proceedings of ACM SIGMOD Conference*, pages 903–908, 2004.

[9] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proceedings of ACM SIGMOD Conference*, pages 204–215, June 2002.

[10] Xml data repository. http://www.cs.washington.edu/research/xmldatasets/.
[11] Liang Xu, Zhifeng Bao, and Tok Wang Ling. A Dynamic Labeling Scheme Using Vectors. In *Proceedings of DEXA*, pages 130–140, 2007.
[12] Liang Xu, Tok Wang Ling, and Huayu Wu. Labeling dynamic xml documents:an order-centric approach. *IEEE Transactions on Knowledge and DataEngineering*, 2010.

Jie Chen received her B. Sc. in Software Engineering from School of Software, Dalian University of Technology in 2010. She is currently a master student with Dalian University of Technology. Her main research interests include XML databases and XML labeling.

Wenxin Liang received his B.E. and M.E. degrees from Xi'an Jiaotong University, China in 1998 and 2001, respectively. He received the Ph.D. degree in Computer Science from Tokyo Institute of Technology in 2006. He was a Postdoc Research Fellow, CREST of Japan Science and Technology Agency (JST) and a Guest Research Associate, GSIC of Tokyo Institute of Technology from Oct. 2006 to Mar. 2009. His main research interests include XML Data Processing and Management, XML Storage, Indexing, Labeling and Querying Techniques, XML Keyword Search, Web-based IR, Knowledge Discovery and Management, etc. He is currently an associate professor at School of Software, Dalian University of Technology, China. He is a senior member of China Computer Federation (CCF), and a member of IEEE, ACM, ACM SIGMOD Japan Chapter and Database Society of Japan (DBSJ).

Haruo Yokota received the B.E., M.E., and Dr. Eng. degrees from Tokyo Institute of Technology in 1980, 1982, and 1991, respectively. He joined Fujitsu Ltd. in 1982, and was a researcher at ICOT for the Japanese 5th Generation Computer Project from 1982 to 1986, and at Fujitsu Laboratories Ltd. from 1986 to 1992. From 1992 to 1998, he was an Associate Professor in Japan Advanced Institute of Science and Technology (JAIST). He is currently a Professor at Department of Computer Science in Tokyo Institute of Technology. His research interests include general research area of data engineering, information storage systems, and dependable computing. He is an associate editor of the VLDB Journal, a chair of ACM SIGMOD Japan Chapter, a trustee member of IPSJ and the Database Society of Japan (DBSJ), a fellow of IEICE and IPSJ, and a member of JSAI, IEEE, IEEE-CS, ACM and ACM-SIGMOD.