Efficient Processing of Updates in Dynamic XML Data

Changqing Li
Department of CS
National University of Singapore
lichangq@comp.nus.edu.sg

Tok Wang Ling
Department of CS
National University of Singapore
lingtw@comp.nus.edu.sg

Min Hu
Department of COFM
National University of Singapore
g0406391@nus.edu.sg

Abstract

It is important to process the updates when nodes are inserted into or deleted from the XML tree. All the existing labeling schemes have high update cost, thus in this paper we propose a novel Compact Dynamic Binary String (CDBS) encoding to efficiently process the updates. CDBS has two important properties which form the foundations of this paper: (1) CDBS supports that codes can be inserted between any two consecutive CDBS codes with the orders kept and without re-encoding the existing codes; (2) CDBS is orthogonal to specific labeling schemes, thus it can be applied broadly to different labeling schemes or other applications to efficiently process the updates. We report our experimental results to show that our CDBS is superior to previous approaches to process updates in terms of the number of nodes to re-label and the time for updating.

1. Introduction

With the rapidly increasing popularity of XML [6] for data representation and exchange, there is a lot of interest in query processing over data that conforms to an *ordered tree-structured* data model. With the tree model, data objects, e.g. elements, attributes, text data, etc., are modeled as the nodes of a tree, and relationships are modeled as the edges to connect the nodes of the tree. Figure 1 shows an ordered XML tree.

XPath [4] and XQuery [5] are two main XML query languages that express the structure of XML documents as linear paths or twig patterns. For example, the XPath query:

/book[/title]//section[2]/preceding-sibling::section

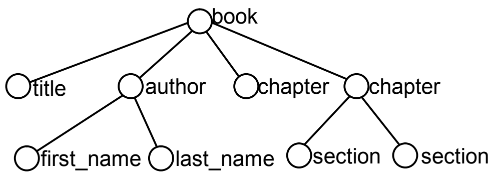

Figure 1. An ordered XML tree

finds all the *section* nodes that are siblings of *section[2]* and these *section* sibling nodes should be before *section[2]* ("*preceding*-sibling"). Meanwhile, *section[2]* should be a descendant of *book* ("//"). In addition, *book* should satisfy the restriction that it has a child *title* ("/").

No matter the query is a linear path or a twig pattern, the core operation is to efficiently determine the ancestor-descendant, parent-child, sibling and ordering relationships. To facilitate the determination of these relationships, several labeling (numbering) schemes, e.g. containment [1, 11, 18], prefix [8, 13, 15] and prime [16], have been proposed. Based on the labels only, the ancestor-descendant and parent-child relationships can be fast determined.

If the XML is static, the existing labeling schemes can efficiently process different queries. However if the XML is dynamic, how to efficiently update the labels of the labeling schemes becomes to an important research topic.

As we know, the elements in the XML are intrinsically ordered, which is referred to as document order, i.e. the element sequence in the XML document. The relative order of two paragraphs in the XML is important because the order may influence the semantics, thus the standard XML query languages (e.g. XPath [4] and XQuery [5]) require the output of queries to be in document order by default. Hence it is very important to maintain the document order when the XML is updated.

Some researches [2, 8, 13, 14, 15, 16] have been done to maintain the document order in XML updating. However the update costs of these approaches are still expensive. Therefore in this paper we focus on how to dramatically decrease the update cost.

The main contributions of this paper include:

- We propose a novel *Compact Dynamic* Binary String (CDBS) encoding, which supports that CDBS codes can be inserted between any two *consecutive* CDBS codes *with the orders kept and without re-encoding* the existing codes.
- CDBS is *orthogonal* to specific labeling schemes, thus it can be applied *broadly* to different labeling schemes to avoid the re-labeling in XML updates.
- We design algorithms to implement our CDBS and formally analyze the total code size of our CDBS,

which shows that our CDBS encoding is a very *compact* encoding, yet it efficiently supports updates.
- We conduct comprehensive experiments to demonstrate the benefits of our CDBS over the previous approaches to process *updates*.

The rest of the paper is organized as follows. Section 2 reviews the related work. In Section 3, we illustrate that the most important feature of this paper is that we compare labels based on the *lexicographical order*; an algorithm that can insert a binary string between two binary strings with the lexicographical orders kept is also proposed in this section which is the *first foundation* of this paper. We propose our *Compact Dynamic* Binary String (CDBS) encoding in Section 4. In Section 5, we indicate that our CDBS encoding can be applied *broadly (the second foundation)* to different labeling schemes, and show how our CDBS processes the XML updates. In Section 6, we discuss how to completely avoid the re-labeling. The experimental results are reported in Section 7, and we conclude in Section 8.

2. Background and related work

2.1. Containment labeling scheme

Zhang et al [18] use a labeling scheme in which every node is assigned three values: "start,end,level" (Figure 2). For any two nodes u and v, u is an ancestor of v iff u.start < v.start and v.end < u.end. In other words, the interval of v is *contained* in the interval of u. Node u is a parent of node v iff u is an ancestor of v and v.level – u.level = 1.

Although the containment scheme is efficient to determine the ancestor-descendant (A-D) relationship, the insertion of a node will lead to a re-labeling of all the ancestor nodes of this inserted node and all the nodes after this inserted node in document order (if we do not re-label, not only can not the document order be maintained, but also the containment scheme can not work correctly to determine the A-D etc. relationships). This problem may be alleviated if the interval size is increased with some values unused [11]. However, large interval size wastes a lot of numbers which causes the increase of storage, while small interval size is easy to lead to re-labeling.

To solve the re-labeling problem, [2] uses Float-point values for the "start"s and "end"s of the intervals. It seems that Float-point solves the re-labeling problem [15]. But in practice, Float-point is represented in a computer with a fixed number of bits [2, 15]. As a result, at most 18 nodes can be inserted at a fixed place [2] since [2] uses the consecutive integer values at the initial labeling. Even if [2] uses values with large gaps, it still can not avoid the re-labeling due to the float-point precision. Thus using real values instead of integers only provides limited benefits for the label updating [15, 16].

2.2. Prefix labeling scheme

In the prefix labeling scheme, the label of a node is that its parent's label concatenates its own label (self_label). For any two nodes u and v, u is an ancestor of v iff label(u) is a prefix of label(v). Node u is a parent of node v iff label(v) has no prefix when removing label(u) from the left side of label(v).

DeweyID [15] (see Figure 3) labels the nth child of a node with an integer n, and this n should be concatenated to its parent's label and the delimiter (e.g. ".") to form the complete label of this child node. In practice, DeweyID uses UTF8 [17] encoding to process delimiters.

When a node is inserted, DeweyID needs to re-label the sibling nodes after this inserted node and the descendants of these siblings to maintain the document order. [8] uses Binary String to label the XML tree, but it has very large label sizes and can *not avoid* re-labelings.

OrdPath [13] is similar to DeweyID, but it only uses the odd numbers at the initial labeling. When the XML tree is updated, it uses the even number between two odd numbers to concatenate another odd number. OrdPath wastes half of the total numbers. The query performance of OrdPath is worse than that of DeweyID since OrdPath needs more time to decide the prefix levels based on the even and odd numbers. Below is an example.

Example 2.1 Given three DeweyID labels "1", "2" and "3", we know that they are siblings. But for OrdPath, its labels are "1", "3", "5" etc.; when inserting a label between "1" and "3", it uses the even number between "1" and "3" i.e. "2" to concatenate another odd number i.e. "1" as the label of this inserted node, i.e. the inserted label is "2.1". In OrdPath, "2.1" is at the same level as "1", 3" etc., i.e. "2.1" is a sibling of "1" and "3". This makes OrdPath slow to determine the sibling, parent-child etc. relationships in XML query processing. Therefore OrdPath gets better update performance by decreasing the query performance. That is not what we expect.

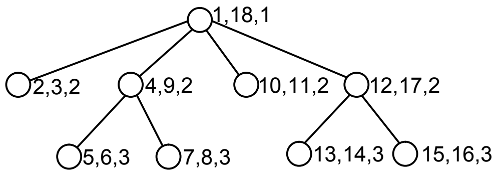

Figure 2. Containment scheme

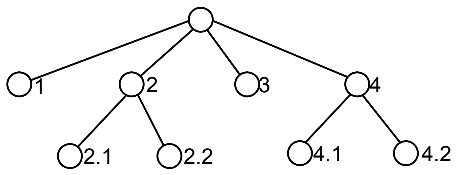

Figure 3. Prefix scheme

2.3. Prime labeling scheme

Wu et al [16] use Prime numbers to label XML trees. The root node is labeled with “1” (integer). Based on a top-down approach, each node is given a unique prime number (self_label) and the label of each node is the product of its parent node’s label (parent_label) and its own self_label. For any two nodes u and v, u is an ancestor of v iff label(v) mod label(u) = 0. Node u is a parent of node v iff label(v)/self_label(v) = label(u).

Prime uses the SC (Simultaneous Congruence) values in Chinese Remainder Theorem [3, 16] to determine the document order, i.e. *SC* mod *self_label = document order* (the details can be found in [16]). When the document order is changed, Prime only needs to re-calculate the SC values instead of re-labeling, but the re-calculation is much more time consuming.

2.4. Motivation

Although Prime supports order-sensitive updates without re-labeling the existing nodes, it needs to re-calculate the SC values based on the new ordering of nodes. The re-calculation is very time consuming.

The main idea of other labeling schemes [2, 11] (except Prime) is to leave some unused values for the future insertions. When the unused values are used up later, they have to re-label the existing nodes, i.e. they can not avoid the re-labeling in XML updates.

Though OrdPath [13] is dynamic to some extent to process the updates, it needs to use the addition and division operations to calculate the even number between two odd numbers (especially when many numbers are deleted and the updates are frequent insertions), thus the update cost of OrdPath is not so cheap.

In addition, the better update performance of OrdPath does not come without a cost. It wastes half of the total numbers which makes its label size larger, and it needs more time to determine the prefix levels based on the even and odd numbers in the XML query processing.

The method in [14] is used to balance the query and update performance, but not to avoid re-labeling.

In this paper, we propose a *novel Compact Dynamic* Binary String (CDBS) encoding. The size of our CDBS is as small as the binary number encoding of *consecutive* integer numbers. As we know, there is no gap between two *consecutive* integer numbers; that means our CDBS is the most compact and it need not leave unused values for the future insertions. Yet our CDBS supports that CDBS codes can be inserted between any two *consecutive* CDBS codes. This is the most important benefit of our CDBS over the previous approaches. In addition, our CDBS can be applied *broadly* to different labeling schemes to process updates.

3. Lexicographical order

The most important feature of our approach is that we compare labels based on the ***lexicographical order*** rather than the numerical order. In this section, we firstly introduce the definition of lexicographical order for binary strings and then propose an algorithm that can always insert a binary string between two lexicographically ordered binary strings. This algorithm is the *foundation* of this paper which guarantees that we can update the XML without re-labeling the existing nodes.

Definition 3.1 (Lexicographical order $\prec$) *Given two binary strings S_L and S_R (S_L represents the left binary string and S_R represents the right binary string), S_L is said to be lexicographically equal to S_R iff they are exactly the same. S_L is said to be lexicographically smaller than S_R ($S_L \prec S_R$) iff*

(a) the lexicographical comparison of S_L and S_R is bit by bit from left to right. If the current bit of S_L is 0 and the current bit of S_R is 1, then $S_L \prec S_R$ and stop the comparison, or

(b) S_L is a prefix of S_R.

Example 3.1 Given two binary strings “0011” and “01”, “0011” $\prec$ “01” lexicographically because the comparison is from left to right, and the 2nd bit of “0011” is “0”, while the 2nd bit of “01” is “1”. Another example, “01” $\prec$ “0101” because “01” is a prefix of “0101”.

Next based on Algorithm 1, Theorem 3.1 and Example 3.2, we illustrate how to insert a binary string S_M (S_M represents the middle binary string) between two lexicographically ordered binary strings S_L and S_R such that $S_L \prec S_M \prec S_R$ lexicographically.

Theorem 3.1 *Given any two binary strings S_L and S_R which are both ended with “1” and $S_L \prec S_R$, we can always find a binary string S_M based on Algorithm 1 such that $S_L \prec S_M \prec S_R$ lexicographically.*

Due to space limitations, we omit the proofs of all the lemmas, theorems and corollaries in this paper. All the proofs can be found in a long version [9] of this paper.

Algorithm 1: AssignMiddleBinaryString(S_L, S_R)

Input: $S_L \prec S_R$; S_L and S_R are both ***ended with “1”***
Output S_M such that $S_L \prec S_M \prec S_R$ lexicographically

```
Description:
1: if size($S_L$) $\geq$ size($S_R$) then   //Case (1)
2:     $S_M = S_L \oplus$ “1”   //$\oplus$ means concatenation
3: else if size($S_L$) < size($S_R$) then   //Case (2)
4:     $S_M = S_R$ with the last bit “1” changed to “01”
5: end if
6: return $S_M$
```

Example 3.2 To insert a binary string between "0011" and "01", the size of "0011" is 4 which is larger than the size 2 of "01", therefore we directly concatenate one more "1" after "0011" (see lines 1 and 2 in Algorithm 1). The inserted binary string is "00111", and "0011" $\prec$ *"00111"* $\prec$ "01" lexicographically. To insert a binary string between "01" and "0101", the size of "01" is 2 which is smaller than the size 4 of "0101", therefore we change the last "1" of "0101" to "01", i.e. the inserted binary string is "01001" (see lines 3 and 4 in Algorithm 1). Obviously "01" $\prec$ "*01001*" $\prec$ "0101" lexicographically.

Next we use an example to show why we require the last bit of the binary string to be "1".

Example 3.3 Suppose there are two binary strings "0" and "00". "0" $\prec$ "00" lexicographically because "0" is a prefix of "00", but we can not insert a binary string S_M between "0" and "00" such that "0" $\prec$ S_M $\prec$ "00". Hence we require the binary strings to be ended with "1".

Algorithm 1 is the *foundation* of this paper which can help to process updates efficiently.

When the labeling scheme is a *prefix* scheme, based on *Theorem 3.1*, we can insert one label between two labels without re-labeling the existing nodes. When the labeling scheme is a *containment* scheme, we may need to insert the "start" and "end" two values at one place. The following *Corollary 3.3* guarantees that two labels can be inserted between two labels without re-labeling.

Lemma 3.2 *The S_M returned by Algorithm 1 is ended with "1".*

Corollary 3.3 *Given any two binary strings S_L and S_R which are both ended with "1" and $S_L \prec S_R$, we can always find two binary strings S_{M1} and S_{M2} such that $S_L \prec S_{M1} \prec S_{M2} \prec S_R$ lexicographically.*

Theorem 3.1 and Corollary 3.3 guarantee that we have low update costs in XML updating.

Algorithm 1 proposed in this paper is *dynamic* and can be applied to any two ordered binary strings (ended with "1") for insertions. On the other hand, to maintain the high query performance, we should not increase the label size when reducing the update cost (see Section 4).

4. A compact dynamic binary string encoding

In this section, we propose a ***Compact*** Dynamic Binary String encoding, called CDBS. All the codes (binary strings) of CDBS are ended with "1". CDBS encoding is as compact as the traditional binary number encoding of consecutive integer numbers, and based on Algorithm 1, CDBS supports *updates* efficiently.

Table 1. Binary and our CDBS encodings

Integer number	V-Binary	V-CDBS	F-Binary	F-CDBS
1	1	00001	00001	00001
2	10	0001	00010	00010
3	11	001	00011	00100
4	100	0011	00100	00110
5	101	01	00101	01000
6	110	01001	00110	01001
7	111	0101	00111	01010
8	1000	011	01000	01100
9	1001	0111	01001	01110
10	1010	1	01010	10000
11	1011	10001	01011	10001
12	1100	1001	01100	10010
13	1101	101	01101	10100
14	1110	1011	01110	10110
15	1111	11	01111	11000
16	10000	1101	10000	11010
17	10001	111	10001	11100
18	10010	1111	10010	11110
Total size (bits)	64	64	90	90

We firstly use an example to illustrate how our CDBS encodes a set of numbers, and use examples to simply analyze the total size of the CDBS codes. Next the formal encoding algorithm in Section 4.1 and the formal size analysis in Section 4.2 will be easier to understand.

Table 1 shows the binary number encoding and our CDBS encoding of 18 numbers. We choose 18 as an example because the total "start" and "end" values in Figure 2 are 18. In fact, CDBS can encode any number (not only 18; see the formal algorithm in Section 4.1).

When encoding 18 integer numbers in binary, they are shown in Column 2 (V-Binary Column) of Table 1 which are with Variable lengths, called V-Binary.

Column 3 (V-CDBS Column) of Table 1 shows our CDBS, called V-CDBS because its codes are also with Variable lengths. The following steps show the details of how to get our V-CDBS codes (binary strings).

Step 1: In the encoding of the 18 numbers, we suppose there is one more number before number 1, say number **0**, and one more number after number 18, say number **19**.

Step 2: We firstly encode the middle number with binary string "***1***". The middle number is **10** where 10 is calculated in this way, 10 = round(0+(19–0)/2). The V-CDBS code of number 10 is "1" (see Table 1).

Step 3: Next we encode the middle number between 0 and 10, and between 10 and 19. The middle number between 0 and 10 is 5 (5=round(0+(10-0)/2)) and the middle number between 10 and 19 is 15 (15=round(10+(19-10)/2)).

Step 4: To encode number 5, the code size of number 0 is 0 (the V-CDBS code of number 0 *corresponding to* $\boldsymbol{S_L}$ *in Algorithm 1* is empty now), and the code size of number 10 is 1 (the V-CDBS code of number 10 *corresponding to* $\boldsymbol{S_R}$ *in Algorithm 1* is "1" now with size 1 bit). This is **Case (2) where size(S_L) < size(S_R)** (see *Algorithm 1*). Thus based on lines 3 and 4 in Algorithm 1, the V-CDBS code of the number **5** is "***01***" ("1"$\rightarrow$"01").

Step 5: To encode number 15, the 10th code (S_L) is "1" now with size 1 bit, and the 19th code (S_R) is empty now with size 0 bit. This is **Case (1) where size(S_L) $\geq$ size(S_R)** (see *Algorithm 1*). Therefore based on lines 1 and 2 in Algorithm 1, the V-CDBS code of the number **15** is "***11***" ("1"$\oplus$"1"$\rightarrow$"11").

Step 6: Next we encode the middle numbers between 0 and 5, between 5 and 10, between 10 and 15, and between 15 and 19, which are the numbers 3, 8, 13 and 17 respectively. The encodings of these numbers are still based on Case (1) or Case (2).

In this way, all the numbers except 0 will be encoded because the *round* function will reach the larger value (divided by 2), and we need to discard the V-CDBS code for number 19 since number 19 does not exist actually.

Also we can encode the integer numbers 1-18 with Fixed length binary numbers, called F-Binary (F-Binary Column of Table 1). Since 18 needs 5 bits to store, zero or more "0"s should be added *before* each code of V-Binary. On the other hand, when representing our CDBS using Fixed length, called F-CDBS, we concatenate "0"s *after* the V-CDBS codes (F-CDBS Column of Table 1).

Example 4.1 It can be seen from Table 1 that V-Binary has one code "1" with size 1 bit, two codes "10" and "11" with sizes 2 bits, four codes "100", "101", "110" and "111" with sizes 3 bits, etc., and the total size of V-Binary is 64 bits. Also we can see that our V-CDBS has one code "1" with size 1 bit, two codes "01" and "11" with sizes 2 bits, four codes "001", "011", "101" and "111" with sizes 3 bits, etc., and the total size of our V-CDBS is also 64 bits. This means that our V-CDBS is as compact as the traditional binary number encoding of consecutive integer numbers. Note that there are no gaps between two consecutive integer numbers, hence our V-CDBS is the most compact. It is similar for F-Binary and our F-CDBS.

Example 4.2 Table 1 shows that V-Binary has smaller total code size than F-Binary. However, we also need to store the size of each V-Binary code, the maximal size for a code is 5, e.g. the size of "10010" is 5. We need to store this 5 using fixed length of bits ("101"; 3 bits). The sizes of other codes should also be stored using fixed length of bits (3 bits), therefore the total code size for V-Binary is $\mathbf{3}\times 18+64=118$ bits which is larger than the bits required by F-Binary. It is similar for our V-CDBS and F-CDBS.

4.1. Encoding algorithm

Because F-CDBS is that some "0"s are concatenated after the V-CDBS codes, we focus on V-CDBS to introduce the algorithm.

Algorithm 2 is the V-CDBS encoding algorithm. We use the SubEncoding procedure to get all the codes of the numbers (except 0).

SubEncoding is a recursive procudure, the input of which is an array codeArr, the left position "P_L" and the right position "P_R" in the array codeArr. This procedure assigns codeArr[P_M] (corresponding to S_M in Algorithm 1) using the AssignMiddleBinaryString algorithm (Algorithm 1), then it uses the new left and right positions to call the SubEncoding procedure itself, until each (except the 0th) element of the array codeArr has a value.

Note that S_L and S_R in the input of Algorithm 1 can be empty when Algorithm 1 is called by SubEncoding here. If S_L and S_R are both empty, their sizes are both equal to 0, and S_M is "1" based on lines 1 and 2 in Algorithm 1. If S_L is empty and S_R is not empty, size(S_L) < size(S_R), and we process S_M based on lines 3 and 4 in Algorithm 1; $S_M \prec S_R$ after insertion. If S_L is not empty and S_R is empty, size(S_L) > size(S_R), and we process S_M based on lines 1 and 2 in Algorithm 1; $S_L \prec S_M$ after insertion.

Theorem 4.1 *Given a number N, Algorithm 2 can encode all the numbers from 1 to N with our V-CDBS codes.*

Lemma 4.2 *All the V-CDBS codes are ended with "1".*

Theorem 4.3 *All the V-CDBS codes are lexicographically ordered.*

Example 4.3 The V-CDBS codes in Table 1 are lexicographically ordered from top to bottom.

Algorithm 2: Encoding (*N*)

```
Input: A positive integer $N$
Output: The V-CDBS codes for numbers 1 to $N$
Description:
  1: suppose there is one more number before the first
     number, called number 0, and one more number after the
     last number, called number ($N$+1)
  2: SubEncoding(codeArr, 1, $N$)
     //here codeArr is an array with size ($N$+2)
  3: discard the 0th and ($N$+1)th elements of the codeArr

Procedure SubEncoding (codeArr, $P_L$, $P_R$)
/*SubEncoding is a recursive procedure; codeArr is an array,
$P_L$ is the left position, and $P_R$ is the right position*/
  1: $P_M$ = round(($P_L$+$P_R$)/2)
  2: if $P_L$+1<$P_R$ then
  3:   codeArr[$P_M$]=
           assignMiddleBinaryString(codeArr[$P_L$], codeArr[$P_R$])
  4:   SubEncoding(codeArr, $P_L$, $P_M$)
  5:   SubEncoding(codeArr, $P_M$, $P_R$)
  6: end if
```

Lemma 4.2 and Theorem 4.3 guarantee that the *conditions* in Theorem 3.1 and Corollary 3.3 are satisfied, therefore we can *insert without re-labeling* in updates based on *CDBS*.

4.2. Size analysis

The size in this paper refers to bits and the log in this paper is used as the logarithm to base 2.

V-Binary For V-Binary, one number is stored with one bit ("1"; see Table 1), two numbers are stored with two bits ("10" and "11"), four numbers are stored with three bits ("100", "101", "110" and "111"), ···, therefore the total size of V-Binary is

$$1\times 1+2\times 2+2^2\times 3+2^3\times 4+\cdots+2^n\times(n+1)$$
$$=n\times 2^{n+1}+1 \qquad (1)$$

Suppose the total number of codes is N, which should be equal to $2^0+2^1+\cdots+2^n=2^{n+1}-1$. Thus formula (1) becomes to

$$N\log(N+1)-N+\log(N+1) \qquad (2)$$

V-CDBS When considering our V-CDBS, it has one number stored with one bit ("1"), two numbers stored with two bits ("01" and "11"), four numbers stored with three bits ("001", "011", "101" and "111"), ···, therefore our V-CDBS has the same total code size as V-Binary.

In addition, since V-Binary and our V-CDBS are with variable length, we need to store the size of each code. A fixed-length number of bits are used to store the size of each code. The maximal size for a code is $\log(N)$. To store this size, the bits required are $\log(\log(N))$, and the total bits required to store the sizes of all the variable codes are $N\log(\log(N))$. When taking formula (2) into account, the total sizes of V-Binary and our V-CDBS are both

$$N\log(N+1)+N\log(\log(N))-N+\log(N+1) \qquad (3)$$

F-Binary To store N numbers with fixed lengths, the size required is

$$N\log(N) \qquad (4)$$

The size of the F-Binary code also needs to be stored, but needs to be stored only once, which needs size $\log(\log(N))$. Therefore the total size for F-Binary is

$$N\log(N)+\log(\log(N)) \qquad (5)$$

F-CDBS has the same total code size as formula (5).

Note that for simplicity, we omit the *ceiling* functions on the *log* functions in all the formulas.

Theorem 4.4 *V-CDBS and F-CDBS are the most compact variable and fixed length binary string encodings which support updates efficiently.*

5. Applying CDBS to different labeling schemes for update processing

In this section, we mainly illustrate how our V-CDBS can be applied to different labeling schemes. F-CDBS is similar since it is that some zeros are concatenated after the V-CDBS codes.

5.1. Applications of CDBS

We firstly describe a property which is the *second foundation* of this paper (the first one is Theorem 3.1).

Property 5.1 *Our V-CDBS (F-CDBS) is orthogonal to specific labeling schemes, thus it can be applied broadly to different labeling schemes or other applications which need to maintain the order in updates.*

When we replace the "start" and "end" values 1-18 of the containment scheme [18] in Figure 2 (similar for other containment schemes [1, 7, 11]) with our V-CDBS codes in Table 1 and based on the lexicographical comparison, a V-CDBS based containment labeling scheme is formed, called *V-CDBS-Containment*.

Similarly, we can replace the integer numbers (see Figure 3) in the prefix labeling scheme [15] with our V-CDBS codes, then a V-CDBS based prefix labeling scheme is formed, called *V-CDBS-Prefix*. We use the following example to show *V-CDBS-Prefix*.

Example 5.1 From Figure 3, we can see that the root has 4 children. To encode 4 numbers based on Algorithm 2, the V-CDBS codes will be "*001*", "*01*", "*1*" and "*11*". Similarly if there are two siblings, their self_labels are "*01*" and "*1*". Figure 4 shows *V-CDBS-Prefix*.

Similarly we can apply our V-CDBS to the prime labeling scheme to record the document order. But because Prime employs the modular and division operations to determine the ancestor-descendant etc. relationships, its query efficiency is quite bad (see Section 7 for the experimental results). Therefore we do not discuss in detail how our V-CDBS is applied to Prime.

Till now, the question will be asked that our V-CDBS only has the orders but does not have the exact position of each code, which is a deficiency when compared to the V-Binary codes. For example, from a V-Binary code "110", we can immediately know that "110" corresponds to the integer number 6. However, if we delete the V-Binary

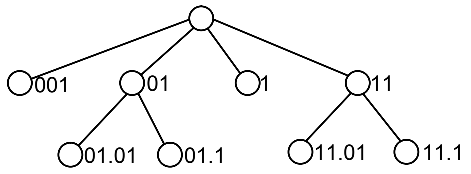

Figure 4. V-CDBS-Prefix scheme

codes “100” and “101”, “110” is now not the 6th number but the 4th number in order. In this paper, we focus on the dynamic XML in which there are a lot of deletions and insertions, therefore *V-Binary does NOT have merits over our V-CDBS in processing the n^{th} position label*.

In addition, it is not to say that our V-CDBS can not immediately get the exact position in the static environment. Based on an inverse processing of Algorithm 2, we can get the exact position of each V-CDBS code by calculations only. However, the static XML is not an emphasis of this paper, thus we do not show the details on calculating the V-CDBS positions.

5.2. Processing of updates

5.2.1. Updates based on V-CDBS-Prefix and V-CDBS-Containment. The deletion of a node will not affect the *relative* orders of the nodes in the XML. Hence we mainly discuss how to process the insertions based on V-CDBS.

If we want to insert a sibling node before “01.01” in Figure 4, the *self_label* of the inserted node is “001” (see lines 3 and 4 in Algorithm 1; *complete label* is “01.001”). Theorem 3.1 guarantees that we need not re-label the existing nodes but we can keep the orders. The insertions at other places also need not re-label the existing nodes.

Similarly if we insert a sibling node before “5,6,3” in Figure 2, we should insert two values (“start” and “end”) between the start of “4,9,2” i.e. “4” and the start of “5,6,3” i.e. “5”. The existing schemes can not insert a number between “4” and “5”, but our V-CDBS codes for “4” and “5” are “0011” and “01” (see Table 1), and Corollary 3.3 guarantees that we can insert *two* binary strings between “0011” and “01” with the orders kept. The inserted two binary strings can be either “00111” and “001111”, or “001101” and “00111” (see Algorithm 1). That means we need not re-label the existing nodes, but we can keep the containment scheme working correctly.

After insertion, we can further insert other nodes.

5.2.2. Frequent updates. The size analysis in Section 4.2 is based on the initial encoding. Algorithm 2 shows that our encoding algorithm is step by step insertions of nodes evenly at different places. Therefore if a sequence of nodes are inserted randomly at different places of the XML, the size analysis in Section 4.2 is still valid, and the *query performance will not be decreased*.

For the case that nodes are always inserted at a fixed place (skewed insertion) of the XML, the size of our V-CDBS increases fast. [8] proves that any deterministic labeling scheme which does not re-label nodes must in the worst case assign *one* label with size *O(N)*. Our V-CDBS can not escape from this claim also, i.e. *O(N)* is the upper bound of the size of each V-CDBS code. OrdPath [13] also has this skewed insertion problem.

6. Completely avoid re-labeling

CDBS proposed in Section 4 still can *not* completely avoid re-labelings in XML updates. Here is an example.

Example 6.1 The size field of each V-CDBS code is stored with fixed length (e.g. 3; see Example 4.2). If many nodes are inserted into the XML tree, the size of the length field (e.g. 3) is not enough for the new labels, then we have to re-label all the existing nodes. Even if we increase the size of the length field (e.g. 3) to a larger number, it still can not completely avoid the re-labeling, and it will waste the storage space. This is called the *overflow* problem in this paper. Similarly *F-CDBS* and *OrdPath* [13] (when separating different labels based on sizes, e.g. separate “1.1” and “1.3”) will encounter the overflow problem also.

To completely avoid the re-labeling, we use the quaternary encoding approach in [10], called QED. In QED, four quaternary numbers “0”, “1”, “2” and “3” are used, and each quaternary number is stored with 2 bits, i.e. “00”, “01”, “10” and “11”. Only “1”, “2” and “3” will appear in the QED code itself; we use “0” as the separator to separate different codes, and “0” will never encounter the overflow problem, thus QED can completely avoid the re-labeling in updates. The details of QED can be found in [10].

On the other hand, though QED can completely avoid the re-labeling, it is not the most compact, i.e. its size is larger than the size of V-CDBS, and its update cost is larger than V-CDBS, i.e. it needs to modify the last 2 bits of the neighbor label, while V-CDBS only needs to modify the last 1 bit.

7. Experimental evaluation

7.1. Experimental setup

7.1.1. Test bed and datasets. We evaluate and compare the performance of different labeling schemes. The scheme names containing a “CDBS” or “QED” are all our schemes; all the others are prior schemes. The schemes with a “-Prefix” at the end of the scheme names are prefix schemes, and with a “-Containment” at the end of the scheme names are containment schemes.

All the schemes are implemented in Java and all the experiments are carried out on a 3.0 GHz Pentium 4 processor with 1 GB RAM running Windows XP Professional.

Table 2 shows the characteristics of the test datasets. D1 to D6 are all real-world XML data from [12]. We choose these datasets because they have different characteristics, i.e. their file number, fan-out, depth, and total number of nodes are quite different.

Table 2. Test datasets

Datasets	Topics	# of files	Max/average fan-out for a file	Max/average depth for a file	Total # of nodes for each dataset
D1	Movie	490	14/6	5/5	26044
D2	Department	19	233/81	4/4	48542
D3	Actor	480	37/11	5/5	56769
D4	Company	24	529/135	5/3	161576
D5	Shakespeare's play	37	434/48	6/5	179689
D6	NASA	1882	1188/9	7/5	370292

7.1.2. Overview of experiments. In Section 7.2, we test how our CDBS (V-CDBS and F-CDBS) works on the static XML data which shows that our CDBS does not work worse even in the static environment of XML. When nodes are intermittently inserted into the XML, in Section 7.3 we show that our CDBS works much better (11 times) compared to the existing labeling schemes except OrdPath and Float-point. Though it seems that OrdPath and Float-point also work well in intermittent insertions, in Section 7.4 we illustrate that our approach is much better (more than 300 times) to process the frequent updates than OrdPath and Float-Point.

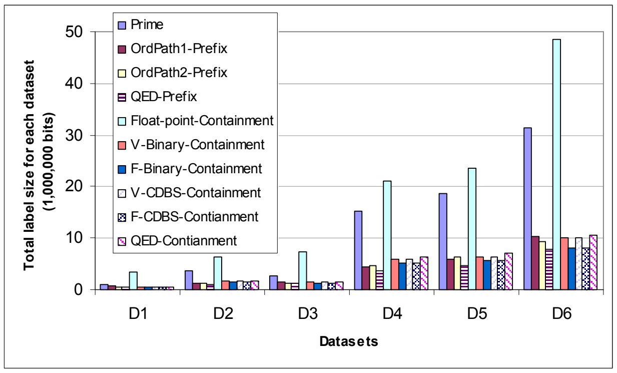

Figure 5. Label sizes of different labeling schemes on D1-D6

7.2. Performance study on static XML

We compare the label size and query performance of different labeling schemes.

7.2.1. Label size. Figure 5 shows the label sizes of different labeling schemes on *datasets D1-D6*.

Prime has very large label size because it skips a lot of numbers to get the prime numbers and it uses the product of the parent label and the self label as the label of a node, which both make its label size very large.

For the prefix schemes, our CDBS can be encoded with the UTF8 [17] encoding or the OrdPath [13] encoding to process the delimiters. If we use UTF8 to process the delimiters, our CDBS(UTF8)-Prefix has the same label size as DeweyID(UTF8)-Prefix. If we use OrdPath encodings to process the delimiters, our CDBS(OrdPath)-Prefix has smaller label size than OrdPath-Prefix because we do not waste the even numbers. Since the UTF8 and OrdPath encodings are existing techniques, we do not compare the UTF8 or OrdPath encoding of our CDBS, DeweyID and OrdPath prefix schemes. In [10], we propose the QED encoding. The "0" in QED can be used as the delimiters. Our QED-Prefix has smaller label size than the prefix schemes DeweyID(UTF8)-Prefix [15, 17] and Binary-String-Prefix [8]. In later comparisons, we only compare the prefix schemes OrdPath-Prefix [13] and our QED-Prefix because they are dynamic. OrdPath-Prefix has two kinds of codes (see [13] for the details), which are represented with names OrdPath1-Prefix and OrdPath2-Prefix. Figure 5 shows that our QED-Prefix has smaller label size than OrdPath1-Prefix and OrdPath2-Prefix.

For the containment schemes, Float-point-Containment has larger label size than other containment schemes. Our V-CDBS-Containment has the same label size as V-Binary-Containment, and our F-CDBS-Containment has the same label size as F-Binary-Containment; these results indicate that our V-CDBS and F-CDBS encodings are the most compact variable and fixed length dynamic encodings. The label size of QED-Containment is larger than the label size of V-CDBS-Containment.

7.2.2. Query performance. As described in [15], we scaled up (replicated) D5 10 times to test the queries. Table 3 shows the ordered and un-ordered *queries (Q1-Q6)* and the number of nodes retrieved.

Figure 6 shows the response time (seconds) of queries *Q1-Q6*.

Prime has very large response time because it has very large label size and it employs the modular and division operations to determine the ancestor-descendant etc. relationships which are very expensive.

OrdPath1-Prefix and OrdPath2-Prefix have worse query performance than our QED-Prefix because their label sizes are larger and it is slow for them to decode the OrdPath1 and OrdPath2 codes and slow to separate the prefix levels (see [13] for the details).

Table 3. Test queries on the scaled D5

Queries		# of nodes Retrieved
Q1	/play/act[4]	370
Q2	/play//personae[./title]/pgroup[.//grpdescr]/persona	2690
Q3	/play/personae/persona[12]/preceding-sibling::*	4240
Q4	//act[2]/following::speaker	184060
Q5	//act/scene/speech	309330
Q6	/play/*//line	1078330

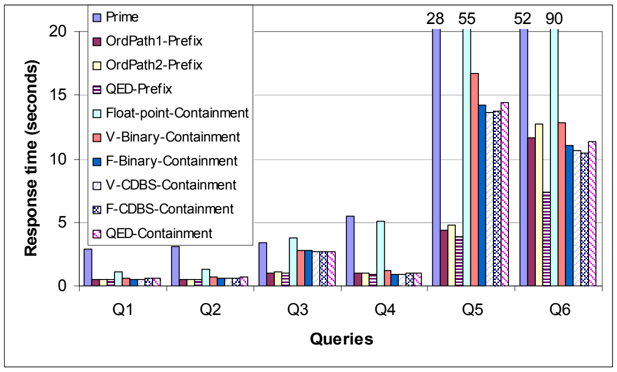

Figure 6. Response time of queries Q1-Q6

Float-point-Containment has much larger response time due to its larger label size. Our CDBS-Containment ("V-" and "F-") has smaller response time than Binary-Containment ("V-" and "F-") because our encodings can directly compare labels from left to right no matter the labels have variable lengths or fixed lengths. The response time of QED-Containment is larger than that of V-CDBS-Containment.

7.3. Performance study on intermittent updates

Same as [16], we select one XML file Hamlet in D5 to test the update performance (it is similar for other XML files). Hamlet has 5 *act* elements. We test the following *5 cases* (see Table 4 and Figure 7): inserting an *act* element before *act[1]*, inserting an act element before *act[2]*, ···, and inserting an act element before *act[5]*.

Table 4 shows the number of nodes for re-labeling when applying different labeling schemes. V-Binary-Containment and F-Binary-Containment need to re-label many nodes in the 5 cases. Note that the Hamlet file has totally 6636 nodes. Though V-Binary-Containment and F-Binary-Containment are the most compact, they need to re-label the existing nodes at each time when a node is inserted into the XML.

For Prime, the number of SC values that are required to re-calculate is counted in Table 4. Because Prime uses each SC value for every five nodes [16], the number of SC values required to re-calculate is 1/5 of the number of nodes required by V-Binary-Containment and F-Binary-Containment to re-label. Note that it is impossible to use a single SC value for all the nodes in the XML since the SC value will be too large a number.

In the five cases, OrdPath-Prefix (OrdPath1-Prefix and OrdPath2-Prefix; without overflow here), our QED-Prefix, Float-point-Containment (less than 18 nodes at a fixed place), our CDBS-Containment ("V-" and "F-"; without overflow here), and our QED-Containment need not re-label any existing nodes. Our V-CDBS-Containment and F-CDBS-Containment are the most compact, yet they need *not* re-label the existing nodes in intermittent updates.

Next we study the *total time* (processing time + I/O time) for updates. Figure 7 shows the ***LOG_2*** *of the total update time (ms) (Y-axis)* of different labeling schemes. The total time required by *Prime* to re-calculate the SC values is much larger (at least 191 times) than the time required by *Binary-Containment* ("V-" and "F-") to re-label the nodes. Prime theoretically is a good scheme for updates, but it is not practicable. In contrast, the total update time of OrdPath-Prefix, our QED-Prefix, Float-point-Containment, our CDBS-Containment ("V-", "F-"), and our QED-Containment is less than 1/5 of the total update time of *Binary-Containment*. Note that the update time of our *CDBS-Containment, QED-Containment, and QED-Prefix* is only 1/11 of the update time of *Binary-Containment*.

Table 4. Number of nodes to re-label in updates

Labeling schemes	Number of nodes to re-label (5 cases)				
	1	2	3	4	5
Prime	1320	1025	787	487	261
OrdPath1-Prefix	0	0	0	0	0
OrdPath2-Prefix	0	0	0	0	0
QED-Prefix	0	0	0	0	0
Float-point-Containment	0	0	0	0	0
V-Binary-Containment	6596	5121	3932	2431	1300
F-Binary-Containment	6596	5121	3932	2431	1300
V-CDBS-Containment	0	0	0	0	0
F-CDBS-Containment	0	0	0	0	0
QED-Containment	0	0	0	0	0

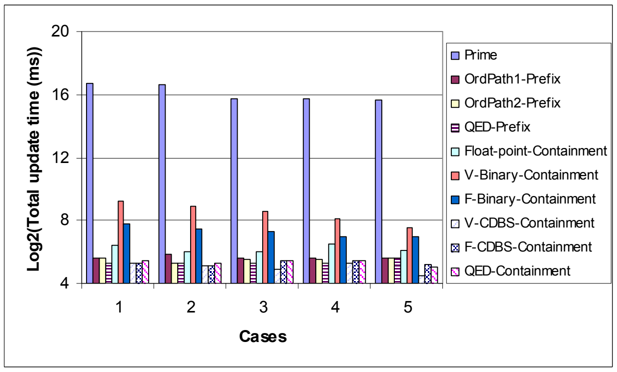

Figure 7. Total time for updates

Figure 7 shows that the update time differences among OrdPath, Float-point and our approaches are not very large (2 times only). This is because the update time of OrdPath, Float-point and our approaches is mainly the I/O time in intermittent updates. When considering the processing time only, our approaches are much better because we only needs to modify the last 1 or 2 bits of the neighbor label which is much cheaper. Section 7.4 illustrates the update time differences among OrdPath, Float-point and our approaches.

7.4. Performance study on frequent updates

When nodes are intermittently inserted into the XML, *Prime* and *Binary-Containment* ("V-" and "F-") have much larger update time, thus it will be a disaster for them to execute the frequent and tiny insertions, which makes them impossible to answer any queries in the frequent insertion environment.

In [10], we show that our QED works much better than (more than 300 times) OrdPath-Prefix and Float-point-Containment to process frequent updates.

Compared with QED [10] which needs to modify the last 2 bits of the neighbor label to get the inserted label, our V-CDBS only needs to modify the last 1 bit which is cheaper. If the frequent updates happen uniformly at different places of the XML data, it is not easy to lead our V-CDBS to re-labeling. Thus our V-CDBS works better to process the uniformly frequent insertions than QED because its label size and update cost are smaller.

On the other hand, if the insertions are always at a fixed place, it is easy to lead our V-CDBS to re-labeling due to the overflow problem. Therefore it is better to use our QED in [10] to process the updates if the insertions are always at a fixed place, i.e. skewed insertions, because our QED can completely avoid the re-labeling.

8. Conclusion and future work

In this paper, we have proposed a *novel dynamic* binary string encoding which is orthogonal to specific labeling schemes, thus it can be applied *broadly* to different labeling schemes or other applications to efficiently maintain the orders in updates.

Our CDBS ("V-" and "F-") is as compact as the most compact binary ("V-" and "F-") number encoding, thus it will not decrease the query performance, yet our CDBS supports intermittent and uniformly frequent updates efficiently. Our V-CDBS only needs to modify the last 1 bit of the neighbor label to get the inserted label which is the cheapest approach compared to all the other existing approaches.

We will further discuss how to efficiently process the skewed insertion problem in the future.

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. *In Proc. of ACM SIGMOD*, pp253-262, 1989.

[2] T. Amagasa, M. Yoshikawa, and S. Uemura. QRS: A Robust Numbering Scheme for XML Documents. *In Proc. of ICDE*, pp705-707, 2003.

[3] J. A. Anderson and J. M. Bell. Number Theory with Application, *Prentice-Hall*, New Jersey, 1997.

[4] A. Berglund et al. XML path language (XPath) 2.0. *W3C working draft 04*, 2005.

[5] S. Boag et al. XQuery 1.0. *W3C working draft 04*, 2005.

[6] T. Bray et al. Extensible markup language (XML) 1.0 third edition *W3C recommendation*, 2000.

[7] V. Christophides et al. On labeling schemes for the semantic web. *In Proc. of WWW*, pp544-555, 2003.

[8] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. *In Proc. of PODS*, pp271-281, 2002.

[9] C. Li. Querying and Updating XML Data Based on Labeling Schemes. *Ph.D Thesis of National University of Singapore.* 2005.

[10] C. Li and T. W. Ling. QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. *In Proc. of CIKM*, pp501-508, 2005.

[11] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. *In Proc. of VLDB*, pp361-370, 2001.

[12] NIAGARA Experimental Data. Available at: http://www.cs.wisc.edu/niagara/data.html

[13] P. E. O'Neil et al. ORDPATHs: Insert-Friendly XML Node Labels. *In Proc. of ACM SIGMOD*, pp903-908, 2004.

[14] A. Silberstein, H. He, K. Yi, and J. Yang. BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data. *In Proc. of ICDE*, pp285-296, 2005.

[15] I. Tatarinov et al. Storing and querying ordered XML using a relational database system. *In Proc. of ACM SIGMOD*, pp204-215, 2002.

[16] X. Wu, M. L. Lee, and W. Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. *In Proc. of ICDE*, pp66-78, 2004.

[17] F. Yergeau, UTF-8, A Transformation Format of ISO 10646. *Request for Comments (RFC) 2279*, January 1998.

[18] C. Zhang et al. On Supporting Containment Queries in Relational Database Management Systems. *In Proc. of ACM SIGMOD*, pp425-436, 2001.