Reuse or Never Reuse the Deleted Labels in XML Query Processing Based on Labeling Schemes

Changqing Li[1], Tok Wang Ling[1], and Min Hu[2]

[1] Dept. of CS, National University of Singapore, Singapore, 117543
{lichangq, lingtw}@comp.nus.edu.sg
[2] Dept. of COFM, National University of Singapore, Singapore, 119260
g0406391@nus.edu.sg

Abstract. To facilitate the XML query processing, several kinds of labeling schemes have been proposed. Based on the labeling schemes, the ancestor-descendant and parent-child relationships in XML queries can be quickly determined without accessing the original XML file. Recently, more researches are focused on how to update the labels when nodes are inserted into the XML. However how to process the deleted labels are not discussed previously. We think that the deleted labels can be processed in two different directions: (1) reuse all the deleted labels to control the label size increasing speed and improve the query performance; (2) never reuse the deleted labels to query different versions of the XML data based on labeling schemes. In this paper, we firstly introduce our previous work, called QED, which can completely avoid the re-labeling in XML updates. Secondly based on QED we propose a new algorithm, called Reuse, which can reuse all the deleted labels to control the label size increasing speed; meanwhile the Reuse algorithm can completely avoid the re-labeling also. Thirdly to query different versions of the XML data, we propose another new algorithm, called NeverReuse, which is the only approach that never reuses any deleted labels. Extensive experimental results show that the algorithms proposed in this paper can control the label size increasing speed when reusing all the deleted labels, and is the only approach to query different versions of the XML data based on labeling schemes.

1 Introduction

XML query processing has been thoroughly studied in the past few years. Many techniques, e.g. structural index [11, 12] and labeling scheme [1, 6, 18], have been proposed to facilitate XML queries. The structural index is a structure summary from the original data which can help to traverse the hierarchy of XML, but this traversal is costly and the overhead of the traversal can be substantial if the path lengths are very long or unknown. On the other hand, The labeling scheme requires smaller storage space, yet it can efficiently determine the ancestor-descendant (A-D) and parent-child (P-C) relationships between any two elements of the XML. In this paper, we focus on the labeling scheme.

Recently how to efficiently update the XML has gained a lot of attention [2, 6, 8, 13, 14, 15, 18]. Different algorithms have been proposed to decrease the update cost,

M.L. Lee, K.L. Tan, and V. Wuwongse (Eds.): DASFAA 2006, LNCS 3882, pp. 659 – 673, 2006.

however the updates are focused on how to process the labels when a node is *inserted* into the XML. How to process the deleted labels is not considered in the previous researches.

We think that the deleted labels can be processed in two different directions: (1) reuse all the deleted labels to control the label size increasing speed; (2) never reuse the deleted labels to query different versions of the XML. Thus the objective of this paper is to propose algorithms to process the deleted labels in these two different directions, and meanwhile to keep the low update cost.

The main contributions of this paper are summarized as follows:

- We propose a new algorithm which can reuse all the deleted labels. In this way, we control the label size increasing speed when nodes are deleted and inserted in the XML data. This Reuse algorithm need not re-label the existing nodes in updates.
- We propose another new algorithm that is the first one which never reuses the deleted labels; it is the only approach that truly maintains different XML label versions and supports the query of different XML versions based on labeling schemes.
- We conduct several experiments which show that the Reuse algorithm can efficiently control the label size increasing speed, and the NeverReuse algorithm can truly maintain the different label versions (the labels in different versions are unique).

The rest of the paper is organized as follows. Section 2 reviews the related work. We introduce our previous work on how to update the XML without re-labeling the existing nodes and give the motivation of this paper in Section 3. In Section 4, We propose the algorithm to reuse all the deleted labels which can control the label size increasing speed in the update environment with both insertions and deletions. We propose another algorithm which never reuses the deleted labels and accordingly supports the query of different XML versions based on labeling schemes in Section 5. The experimental results are illustrated in Section 6, and we conclude in Section 7.

2 Related Work

2.1 XML Labeling Schemes

We present three families of XML labeling schemes, namely containment [1, 9, 19, 20], prefix [6, 13, 15] and prime [18].

Containment Scheme. Zhang et al [20] use a labeling scheme in which every node is assigned three values: “start”, “end” and “level”. 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.

In Figure 1, “5,6,3” is a child of “2,7,2” since interval [5, 6] is contained in interval [2, 7] and levels 3 – 2 = 1.

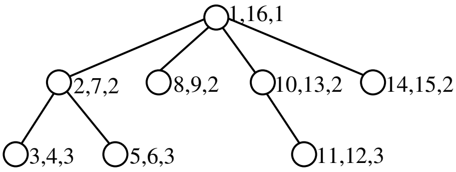


Fig. 1. Containment scheme

Although the containment scheme is efficient to determine the ancestor-descendant 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. This problem may be alleviated if the interval size is increased with some values unused. However, large interval sizes waste a lot of numbers which causes the increase of storage, while small interval sizes are easy to lead to re-labeling.

To solve the re-labeling problem, [2] uses Float-point values for the "start" and "end" of the interval. It seems that Float-point solves the re-labeling problem [15]. But in practice, the Float-point is represented in a computer with a fixed number of bits [2, 15]. As a result, only 18 values [2] can be inserted between any two real values 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. Therefore, using real values instead of integers does not provide any benefit for the node updating [15, 18].

When a node is inserted at a place where a node has ever been deleted, it is natural for the containment scheme to reuse the deleted labels to reduce the storage space. On the other hand, if we need to maintain different XML versions, we should not use the deleted labels in the previous XML versions. All the current containment labeling schemes can not achieve this objective since there are a finite number of values between any two values in a computer; when the finite number of values are used up, the current containment schemes have to reuse the deleted labels or re-label the nodes.

Prefix Scheme. In the prefix labeling scheme, the label of a node is that its parent's label (prefix) concatenates its own (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] labels the nth child of a node with an integer n, and this n should be concatenated to the prefix (its parent's label) to form the complete label of this child node.

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 for the inserted node. OrdPath wastes many numbers which makes its label size larger. The query performance of OrdPath is worse as it needs more time to decide the prefix levels based on the odd and even numbers (see [7] for the experimental results).

Cohen et al [6] uses Binary String to label the nodes (called BinaryString in this paper). The root of the tree is labeled with an empty string. The first child of the root is labeled with "0", the second child with "10", the third with "110", and the fourth with "1110" etc. Similarly for any node u, the first child of u is labeled with label(u)."0", the second child of u is labeled with label(u)."10", and the ith child with label(u)."1^{i-1}0".

When a node is inserted into the XML, DeweyID and BinaryString need to re-label the sibling nodes after this inserted node and the descendants of these siblings. Though OrdPath need not re-label the existing nodes, it increases the label size and decreases the query performance.

It is natural for DeweyID and BinaryString to reuse the deleted labels, but when all the labels between two labels are used up, they have to re-label the existing nodes. Therefore they can not truly maintain the different label versions of the XML since they need re-labeling. OrdPath can reuse the deleted labels, but it does not consider how to avoid reusing the deleted labels.

Prime Scheme. Wu et al [18] 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).

We have compared Prime with other labeling schemes in [7]. Prime has very bad query performance because it has very large label size and it employs the modular and division operations to determine the ancestor-descendant and parent-child relationships. Its update performance is also much worse (see [7]). Therefore we do not discuss Prime further in this paper.

QED. In [7], we propose a compact dynamic binary string approach to efficiently process XML updates, furthermore we propose a dynamic quaternary encoding (called QED) in [8] which can *completely* (no overflow problem) avoid the re-labeling when a node is inserted into the XML. We will in detail introduce QED in Section 3, and the *reus*e and *never reuse* algorithms in Sections 4 and 5 respectively are all based on QED.

2.2 XML Version Control

It is important to maintain and query different versions of the XML [3, 4, 5, 10, 16, 17]. The Reference-Based Version Model [4] discusses the storage performance of multiversion documents. [10] stores the latest version of a document and the sequence of changes of different versions of the XML. [3] queries the historical XML data in which parts of the XML data are always updated. All of these papers are about the version control of the documents. To the best of our knowledge, no one has ever studied how to maintain the different versions of the labels of labeling schemes and how to use the labeling scheme to query different versions of the XML data.

If we want to query different versions of the XML data based on labeling schemes, we should not reuse the deleted labels especially when the deleted labels have order relationships with the new inserted labels.

3 Preliminary and Motivation

Here we introduce our QED [8] based on examples.

3.1 Label the XML Based on QED

Definition 3.1 (Quaternary code). *Four numbers "0", "1", "2" and "3" are used in the code and each number is stored with two bits, i.e. "00", "01", "10" and "11".*

Definition 3.2 (QED code). *QED code is a quaternary code. The number "0" is used as the separator and only "1", "2" and "3" are used in the QED code itself.*

The separator "0" can be used to separate different codes, and it will never encounter the overflow problem, thus QED can completely avoid re-labeling (see [8] for more details). The most important feature of our QED encoding is that it is based on the lexicographical order for efficient updates.

Definition 3.3 (Lexicographical order $\prec$). *Given two Quaternary codes C_A and C_B, C_A is lexicographically equal to C_B iff they are exactly the same. C_A is said to be lexicographically smaller than C_B ($C_A \prec C_B$) iff*

a) "0" $\prec$ "1" $\prec$ "2" $\prec$ "3" (this is always true and is used by condition b)), or

b) compare C_A and C_B symbol by symbol from left to right. If the current symbol of C_A and the current symbol of C_B satisfy (a), then $C_A \prec C_B$ and stop the comparison, or

c) C_A is a prefix of C_B.

From Figure 1, it can be seen that the "start" and "end" values are from 1 to 16. Table 1 shows the QED codes for these 16 numbers, and the following steps are the details of how to get the QED codes. Note that 16 is only an example; any other number is well also. See [8] for the formal QED algorithms.

Step 1. In the encoding of the 16 numbers, we suppose there is one more number before number 1, say number 0, and one more number after number 16, say number 17.

Step 2. The $(1/3)^{th}$ number is encoded with "***2***", and the $(2/3)^{th}$ number is encoded with "***3***". The $(1/3)^{th}$ number is number **6**, which is calculated in this way, 6 = round(0+(17–0)/3). The $(2/3)^{th}$ number is number **11** (11 = round(0+(17–0)×2/3)).

Step 3. The $(1/3)^{th}$ and $(2/3)^{th}$ numbers between number 0 and number 6 are number 2 (2 = round(0+(6–0)/3)) and number 4 (4 = round(0+(6–0)×2/3)). The QED code of number 0 (left code) is now empty with size 0 and the QED code of number 6 (right code) is now "2" with size 1 (here 1 refers to 2 bits). This is **Case (a) where the left code size is smaller than the right code size**. In this case, the $(1/3)^{th}$ code is that we change the last symbol of the *right* code to "1" and concatenate one more "2", i.e. the code of number **2** is "***12***" ("2"→"1" and "1"⊕"2"→"12"), and the $(2/3)^{th}$ code is that we change the last symbol of the *right* code to "1" and concatenate one more "3", i.e. the code of number **4** is "***13***" ("2"→"1" and "1"⊕"3"→"13").

Table 1. QED encoding

Decimal number	QED codes
1	112
2	12
3	122
4	13
5	132
6	2
7	212
8	22
9	23
10	232
11	3
12	312
13	32
14	322
15	33
16	332

Step 4. The $(1/3)^{th}$ and $(2/3)^{th}$ numbers between numbers 6 and 11 are numbers 8 (8 = round(6+(11–6)/3)) and 9 (9 = round(6+(11–6)×2/3)). The QED code of number 6 (left code) is “2” with size 1 (here 1 refers to 2 bits) and the code of number 11 (right code) is “3” with size 1 (here 1 refers to 2 bits). This is **Case (b) where the left code size is larger than or equal to the right code size**. In this case, the $(1/3)^{th}$ code is that we directly contatenate one more “2” after the *left* code, i.e. the code of number **8** is “***22***” (“2” $\oplus$ “2” $\rightarrow$ “22”), and the $(2/3)^{th}$ code is that we directly concatenate one more “3” after the *left* code, i.e. the code of number **9** is “***23***” (“2” $\oplus$ “3” $\rightarrow$ “23”).

Step 5. The $(1/3)^{th}$ and $(2/3)^{th}$ numbers between numbers 11 and 17 are numbers 13 (13 = round(11+(17–11)/3)) and 15 (15 = round(11+(17–11)×2/3)). The code of number 11 (left code) is “3” with size 1 and the code of number 17 (right code) is empty now with size 0. This is still **Case (b)**. Therefore the QED code of number **13** is “***32***” (“3” $\oplus$ “2” $\rightarrow$ “32”), and the code of number **15** is “***33***” (“3” $\oplus$ “3” $\rightarrow$ “33”).

In this way, all the numbers will be encoded.

Lemma 3.1. *All the QED codes are ended with either “2” or “3”.*

Theorem 3.1. *Our QED codes are lexicographically ordered.*

Example 3.1. The QED codes in Table 1 are lexicographically ordered from top to down. “132” $\prec$ “2” lexicographically because the comparison is from left to right, and the 1st symbol of “132” is “1”, while the 1st symbol of “2” is “2”. “23” $\prec$ “232” because “23” is a prefix of “232”.

When we replace the “start”s and “end”s (1-16) in Figure 1 with our QED codes, and based on the lexicographical comparison, a QED containment scheme is formed.

3.2 Order-Sensitive Updates

The following algorithm shows how we can insert nodes without re-labeling the existing nodes.

```
Algorithm 1. AssignInsertedCode
Input: Left_Code, Right_Code
Output: Inserted_Code, such that Left_Code $\prec$ Inserted_Code $\prec$ Right_Code
lexicographically.

Description:
 1: get the sizes, i.e. number of bits, of Left_Code and Right_Code
 2: if size(Left_Code) < size(Right_Code)  //size is the number of bits of the code
 3:     then Inserted_Code = the Right_Code with the last
                              symbol changed to "1" $\oplus$ "2"
 4: else if size(Left_Code) > size(Right_Code)
 5:     if the last symbol of Left_Code is "2"
 6:         then Inserted_Code = the Left_Code with the
                          last symbol changed from "2" to "3"
 7:     else if the last symbol of Left_Code is "3"
 8:         then Inserted_Code = Left_Code $\oplus$ "2"
 9: else if size(Left_Code) = size(Right_Code)
10:     then Inserted_Code = Left_Code $\oplus$ "2"
```

Fig. 2. AssignInsertedCode algorithm

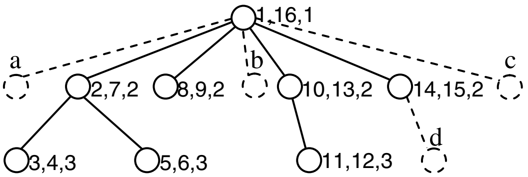


Fig. 3. Update

Example 3.2. When inserting node "**a**" (see Figure 3), we should insert a number between the "start" of the parent "1" (Left_Code) and the "start" of the first sibling "2" (Right_Code). If we use the existing approach, we can not insert a number between "1" and "2", and we must re-label the nodes. However, when referring to Table 1, our QED codes for "1" and "2" are "112" and "12". Based on the AssignInsertedCode algorithm, we insert a QED code between "112" and "12", then the "start" of the inserted node "a" is "**113**" (see lines 4-6 of the AssignInsertedCode algorithm in Figure 2). The "end" of node "a" is an insertion between the new "start" "113" and the "start" of the first sibling "12", thus the "end" of "a" is "**1132**" (see lines 4, 7 and 8 in Figure 2). Obviously, "112"$\prec$ "113" $\prec$ "1132" $\prec$ "12". We need not re-label any existing nodes, but we can keep the containment scheme working correctly. It is similar for the insertions of nodes "b", "c" and "d".

3.3 Motivation

It can be seen that QED avoids the node re-labeling when a node is inserted into the XML. But it has the following deficiencies. QED does not consider how to process the deleted labels. The deleted label with larger size than its neighbor labels will be reused, but the deleted label with smaller size will not be reused.

Example 3.3. When we delete the QED code "12" between "112" and "122" (see Table 1) and want to insert another code at this place, the inserted code will be "1122" (see lines 9 and 10 in Figure 2). The deleted code "12" is not reused because it has smaller size than its neighbors ("112" and "122"), and the re-inserted code "1122" has larger size than the deleted code "12", therefore the size increases fast. On the other hand, if we delete the QED code "122" between "12" and "13" (see Table 1) and insert another code at this place, the inserted code is still "122" (see lines 9 and 10 in Figure 2). The deleted code "122" is reused because it has larger size than its neighbors ("12" and "13").

It is not good to process the deleted labels in this way. If we want to improve the query performance, all the deleted labels should be reused which will hinder the label size from increasing fast. On the other hand, if we want to query different versions of the XML, we should never reuse the deleted labels. That is to say, our current approach is not for the reuse objective and also not for the version control objective. Therefore in Sections 4 and 5, we propose algorithms to reuse the deleted labels and never reuse the deleted labels respectively.

4 Reuse the Deleted Labels (Reuse)

If there are no deletions, the original QED algorithm [8] (Algorithm 1 in this paper) can guarantee that the inserted label has the smallest size between two labels, also the cost of Algorithm 1 is very cheap which only needs to modify the last 2 bits of the neighbor label. However, if there are deletions, Algorithm 1 can not guarantee that the inserted label is with the smallest size. Figure 4 shows the Reuse algorithm. The idea of this algorithm is to find the *smallest* possible code *lexicographically between* two given codes by comparing left_code and right_code symbol by symbol from left to right. From Lemma 3.1, we know that all the QED codes can only be ended with either "2" or "3", therefore the cases and conditions in Figure 4 are complete.

Example 4.1. Suppose the QED code "12" between "112" and "122" (see Table 1) is deleted and we need to insert a new code between "112" and "122". The new inserted code is "1122" based on Algorithm 1 in Figure 2. The deleted code "12" is not reused. On the other hand, based on Algorithm 2 in Figure 4, we compare "112" and "122" symbol by symbol from left to right. The second symbol of left_code ("112") is "1" and the second symbol of right_code ("122") is "2" (see Case (d) in Figure 4). The difference position is at the 2nd symbol (see line 33 in Figure 4), therefore S_L = getPartCode(left_code, P, P) = getPartCode("112", 2, 2) = "1", i.e. the second symbol of "112", and S_R = getPartCode(right_code, P, P) = getPartCode("122", 2, 2) = "2". S_L == "1" and S_R == "2", hence the condition at line 36 in Figure 4 is satisfied. Based on line 37, temp_code = getPartCode(left_code, 1, P-1) $\oplus$ "2" = getPartCode("112", 1, 2-1) $\oplus$ “2” = “1” $\oplus$ “2” = “12”. “12” $\prec$ “122” lexicographically, i.e. temp_code $\prec$ right_code lexicographically, therefore the condition at line 38 is satisfied. As a result, inserted_code = temp_code = “12”. It can be seen the deleted code “12” is reused.

Algorithm 2. AssignInsertedCodeWithReuse

Input: left_code, right_code

Output: inserted_code (reuse the deleted code)

Description:

```
 1: Case (a) left_code and right_code are both empty
 2: inserted_code = “2”
 3: Case (b) left_code is NOT empty but right_code is empty
 4: if “2” does not appear in left_code //left_code contains “1” and “3”, or contains only “3”
 5:   if all the symbols of left_code are “3”
 6:     then inserted_code = left_code $\oplus$ “2”
 7:   else
 8:     then denote the position of the firstly encountered “1” as $P$
 9:          inserted_code = getPartCode(left_code, 1, $P$-1) $\oplus$ “2”
10: else if “1” does not appear in left_code //left_code contains “2” and “3”, or contains only “2”, or
        contains only “3”; //the case that all the symbols of left_code are “3” has been discussed at line 5
11:   then denote the position of the firstly encountered “2” as $P$
12:        inserted_code = getPartCode(left_code, 1, $P$-1) $\oplus$ “3”
13: else if both “1” and “2” appear in left_code //left_code contains “1”, “2” and “3”, or contains only
               “1” and “2”
14:   then denote the position of the firstly encountered “1” as $P_A$, and denote the position of the firstly
               encountered “2” as $P_B$
15:        if $P_A < P_B$
16:          then inserted_code = getPartCode(left_code, 1, $P_A$ -1) $\oplus$ “2”
17:        else if $P_A > P_B$  //note that $P_A$ can not be equal to $P_B$
18:          then inserted_code = getPartCode(left_code, 1, $P_B$ -1) $\oplus$ “3”
19: Case (c) left_code is empty but right_code is NOT empty
20: if “2” does not appear in right_code //right_code contains “1” and “3”, or contains only “3”
21:   then denote the position of the firstly encountered “3” as $P$
22:        inserted_code = getPartCode(right_code, 1, $P$-1) $\oplus$ “2”
23: else if “3” does not appear in right_code //right_code contains “1” and “2”, or contains only “2”
24:   then denote the position of the firstly encountered “2” as $P$
25:        inserted_code = getPartCode(right_code, 1, $P$-1) $\oplus$ “12”
26: else if “2” and “3” both appear in right_code //right_code contains “1”, “2” and “3”, or contains only
               “2” and “3”
27:   then denote the position of the firstly encountered “2” as $P_A$, and denote the position of the firstly
               encountered “3” as $P_B$
28:        if $P_A < P_B$
29:          then inserted_code = getPartCode(right_code, 1, $P_A$ -1) $\oplus$ “12”
30:        else if $P_A > P_B$  //note that $P_A$ can not be equal to $P_B$
31:          then inserted_code = getPartCode(right_code, 1, $P_B$ -1) $\oplus$ “2”
32: Case (d) conditions (a) and (b) in Definition 3.3
33: denote the first difference position of left_code and right_code as $P$; in other words, getPart-
Code(left_code, 1, $P$-1) is equal to getPartCode(right_code, 1, $P$-1) and getPartCode(left_code, $P$, $P$) is
different from getPartCode(right_code, $P$, $P$); denote getPartCode(left_code, $P$, $P$) as $S_L$ ($Symbol_{Left}$) and
getPartCode(right_code, $P$, $P$) as $S_R$ ($Symbol_{Right}$)
34: if ($S_L$ == “1”) and ($S_R$ == “3”)
35:   then inserted_code = getPartCode(left_code, 1, $P$-1) $\oplus$ “2”
36: else if ($S_L$ == “1”) and ($S_R$ == “2”)
```

Fig. 4. AssignInsertedCodeWithReuse algorithm

```
37:   then temp_code = getPartCode(left_code, 1, P-1) $\oplus$ "2"
38:          if temp_code $\prec$ right_code lexicographically
39:              then inserted_code = temp_code
40:          else
41:              then suppose there is a temp_left_code which is equal to getPartCode(left_code, P,
                         left_code.size()) and suppose there is a temp_right_code which is empty
                         //left_code.size() return the total symbol number of left_code
42:                      process temp_left_code and temp_right_code based on Case (b); denote the returned
                         result by Case (b) as temp_code
43:                      inserted_code = getPartCode(left_code, 1, P) $\oplus$ temp_code
44:  else if ($S_L$ == "2") and ($S_R$ == "3")
45:   then temp_code = getPartCode(left_code, 1, P-1) $\oplus$ "3"
46:          if temp_code $\prec$ right_code lexicographically
47:              then inserted_code = temp_code
48:          else
49:              then suppose there is a temp_left_code which is equal to getPartCode(left_code, P,
                         left_code.size()) and suppose there is a temp_right_code which is empty
                         //left_code.size() return the total symbol number of left_code
50:                      process temp_left_code and temp_right_code based on Case (b); denote the returned
                         result by Case (b) as  temp_code
51:                      inserted_code = getPartCode(left_code, 1, P) $\oplus$ temp_code

52:  Case (e) condition (c) in Definition 3.3
53:  left_code is a prefix of right_code; suppose there is a temp_left_code which is empty and suppose
there is a
     temp_right_code which is equal to getPartCode(right_code, left_code.size()+1, right_code.size())
54:  process temp_left_code and temp_right_code based on Case (c); denote the returned result by Case
(c) as temp_code
55:  inserted_code = getPartCode(right_code, 1, left_code.size()) $\oplus$ temp_code

Function getPartCode(code, $P_1$, $P_2$)
  1:  return the symbols of code between position $P_1$ and $P_2$.
```

Fig. 4. (*continued*)

Theorem 4.1. *Algorithms 1 and 2 guarantee that the order can be kept no matter how many codes are inserted at any place of the QED codes.*

Example 4.2. After insertion, the new inserted code based on Algorithm 1 is "1122" and the new inserted code based on Algorithm 2 is "12". Based on Algorithm 1 or 2, we can insert infinite number of QED codes between "112" and "1122" and between "1122" and "122", or between "112" and "12" and between "12" and "122". That means that the Reuse algorithm in this paper can completely avoid the re-labeling also, yet it makes the label size increase slowly.

Theorem 4.2. *Suppose some codes are deleted between left_code and right_code, and suppose the minimum size of these deleted codes is MS. Algorithm 2 guarantees that the inserted code between left_code and right_code is with size MS.*

Example 4.3. Suppose the QED codes "212", "22" and "23" between "2" and "232" (see Table 1) are deleted and we need to insert a new code between "2" and "232". Based on Algorithm 2 in Figure 4, left_code "2" is a prefix of right_code "232", thus it is Case (e). Based on line 53, temp_left_code is empty and temp_right_code = getPartCode(right_code, *left_code.size()*+1, *right_code.size()*) = getPartCode("232", 1+1, 3) = "32". Based on line 54, we need to go to Case (c). The condition at line 26 is satisfied. When we go to line 27, the firstly encountered "2" in "32" is at the 2nd symbol, and the firstly encountered “3” in “32” is at the 1st symbol, i.e. $P_A = 2$ and $P_B = 1$. Thus the condition at line 30 is satisfied, i.e. $P_A > P_B$. Therefore based on line 31, inserted_code = getPartCode(right_code, 1, P_B -1) $\oplus$ “2” = getPartCode(“32”, 1, 1-1) $\oplus$ “2” = “” $\oplus$ “2” = “2”. Next we go back to line 54, and temp_code = “2”. When going to line 55, the final inserted_code = getPartCode(right_code, 1, *left_code.size()*) $\oplus$ temp_code = getPartCode(“232”, 1, 1) $\oplus$ “2” = “2” $\oplus$ “2” = “22”. The deleted code “22” is reused, and it can be seen that the size of “22” is less than or equal to the size of the deleted codes “212” and “23”. That means the deleted code with smaller size is reused firstly. Similarly when we insert a code between “2” and “22”, lines 52, 53, 54, 19, 23, 24, 25 and 55 in Figure 4 will be used and the returned result is “212” which is equal to the deleted code “212”. When we insert a code between “22” and “232”, lines 32, 33, 44, 45, 46 and 47 in Figure 4 will be used and the returned result is “23” which is equal to the deleted code “23”.

Based on Theorem 4.2, the label size will not increase fast.

5 Never Reuse the Deleted Labels (NeverReuse)

If a deleted code has larger size than its neighbors, Algorithm 1 will reuse this deleted code (see the last two sentences of Example 3.3). If we want to query different versions of the XML based on the labeling schemes, Algorithm 1 is not appropriate. We propose another algorithm which never reuses the deleted codes; it truly maintains the different label versions of the XML.

Case (a) in Algorithm 3 (see Figure 5) is easy to understand. Intuitively Case (b) (lines 4-11 in Figure 5) can be summarized as: inserting a code between left_code and the first deleted_code, inserting between any two consecutive deleted codes, and inserting between the last deleted code and right_code using Algorithm 1 (see Figure 2); the final inserted code is the code of all these inserted codes with the smallest size.

Algorithm 3. AssignInsertedCodeWithoutReuse

Input: left_code, right_code, middle_code*s* (between left_code and right_code)
Output: inserted_code, and inserted_code ≠ any one of the deleted_codes

Description:

```
 1: Case (a) process the deleted codes
 2: if middle_codes are deleted
 3:    then do not delete middle_codes physically, but mark them as “deleted”

 4: Case (b) process the inserted code
 5: suppose there are n-2 deleted codes between left_code and right_code
 6: put left_code, deleted_codes, and right_code in an array with size n called LDRcodes
 7: suppose there is another array called temp_inserted_code with size n-1
 8: for (int i=0; i<(n-1); i++) {
 9:     temp_inserted_code[i] = AssignInsertedCode(LDRcodes[i], LDRcodes[i+1])
10: } //AssignInsertedCode is Algorithm 1 in Figure 2
11: inserted_code = min{temp_inserted_code[i] | i∈ [0, n-2]} (min means the minimal size)
```

Fig. 5. AssignInsertedCodeWithoutReuse algorithm

When a node is deleted, it is not physically deleted but instead is marked as “deleted” and is stored ordered with other undeleted nodes. At line 9 of Algorithm 3, we use AssignInsertedCode (Algorithm 1) rather than Algorithm 2 because in fact there are no physical deletions and the cost of Algorithm 1 is smaller than Algorithm 2. We use an example to illustrate Algorithm 3.

Example 5.1. Suppose the QED codes “122”, “13” and “132” between “12” and “2” (see Table 1) need to be deleted. We do not delete them physically, but mark them as “deleted”. When a new code needs to be inserted between “12” and “2”, the new code will be “13” based on Algorithm 1 (Figure 2). The deleted code “13” is reused that is not what we expect. Based on Algorithm 3, we insert codes between left_code “12” and the first deleted_code “122”, between deleted_codes “122” and “13”, between deleted_codes “13” and “132”, and between the last deleted_code “132” and right_code “2”; the inserted codes will be “1212”, “123”, “1312” and “133” based on Algorithm 1. We select the inserted code with the smallest size, e.g. “123”, as the final inserted code. “123” and “133” are the codes between “12” and “2” with the smallest sizes which do not reuse the deleted codes.

Theorem 5.1. *All the deleted codes will NOT be reused based on Algorithm 3.*

Based on Algorithm 3, we will never reuse the deleted codes, as well Algorithm 3 guarantees that a code with smaller size is used before a code with larger size is used.

Algorithm 3 intends to make the label size increase slowly (called NeverReuse-I) However, Algorithm 3 needs more time to calculate the inserted code especially when there are a lot of deleted codes between left_code and right_code. If we want to reduce the insertion time, we can directly use any inserted code (see line 9 in Figure 5) as the final inserted code (called NeverReuse-II), but this can not guarantee that the inserted code is with the smallest size. Furthermore, if a code is required to be inserted between two specific deleted codes (the inserted code should have order relationships with the two specific deleted codes), then insert a code between these two specific deleted codes (called NeverReuse-III) instead of using Algorithm 3. We will further test NeverReuse-I, NeverReuse-II, NeverReuse-III in Section 6.2.

6 Performance Study

The query and update performance of different labeling schemes have been studied in [8]. In this paper, we mainly compare the Reuse and NeverReuse algorithms proposed in this paper with the original QED encoding in [8]. All the experiments 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.

6.1 Performance Study on Reusing the Deleted Codes

Based on the original QED [8], we generate 1,000,000 QED codes. We test the case that codes are deleted then inserted at the *odd* positions of the 1,000,000 codes; after the deletions and insertions, we call these new codes *CodeSet2*; this is case 1. Secondly we test that the codes are deleted then inserted at the *even* positions of

CodeSet2, thirdly *odd* positions of *CodeSet3*, fourthly *even* positions of *CodeSet4*, and so on.

We compare the performance of QED (based on Algorithm 1 in Figure 2) and Reuse (based on Algorithm 2 in Figure 4). Figure 6 shows that the code size of Reuse does not increase in all the ten cases (since we reuse all the deleted codes). On the other hand, the code size of QED increases linearly (for these ten cases) which is fast. Note if there are only insertions (no deletions) at different places of the QED codes, the code size of QED increases logarithmically but not linearly.

The experimental results confirm that our Reuse algorithm (Figure 4) can reuse all the deleted codes, thus it efficiently controls the increasing speed of the code size.

6.2 Performance Study on Never Reusing the Deleted Codes

We delete and insert at any place of the 1,000,000 QED codes generated in Section 6.1. The experimental results confirm that our NeverReuse algorithm(s) (NeverReuse-I, NeverReuse-II, and NeverReuse-III; see the discussions after Theorem 5.1) never reuse any deleted codes, hence the NeverReuse algorithm(s) can truly maintain different label versions of the XML data. There are no other researches about how to never reuse the deleted labels in labeling schemes. Therefore we do not compare different schemes on label version control in the experiments. The other containment and prefix labeling schemes can not truly maintain different XML label versions because they must reuse the deleted labels no matter how large gaps are leaved between two values.

We compare the size and the update time increasing speeds of NeverReuse-I, NeverReuse-II and NeverReuse-III. Figure 7(a) shows that the size (only the size of the inserted codes) differences among the three approaches are not very large though NeverReuse-I is better. On the other hand, Figure 7(b) shows that the update time (only the processing time) of NeverReuse-I increases very fast, but the update time of NeverReuse-II and NeverReuse-III is almost 0 millisecond (ms). Therefore in practice, we suggest using NeverReuse-III because its update time is small, its code size is not large, and the most important reason is that NeverReuse-III can maintain the order relationships among the deleted codes. Maintaining the orders of the deleted codes can only be achieved by our approach.

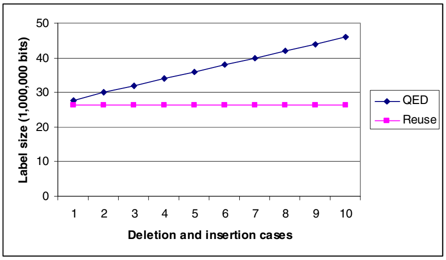
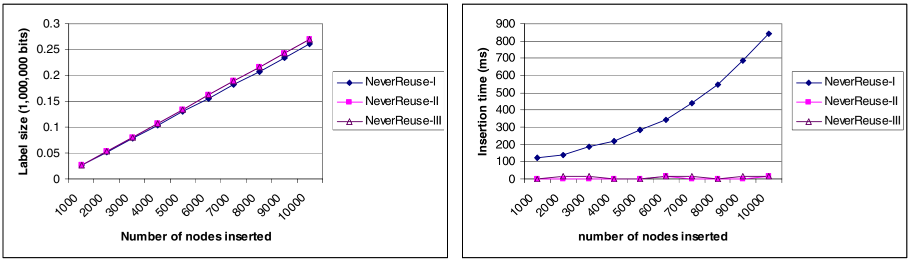


Fig. 6. Sizes of QED and Reuse



(a) Label size

(b) Update time

Fig. 7. NeverReuse

7 Conclusion

In this paper, we propose a new algorithm which can reuse all the deleted labels. In this way, we efficiently control the label size increasing speed. The experimental results also show that with this algorithm we can greatly decrease the label size when a lot of nodes are deleted and inserted. Meanwhile, the Reuse algorithm can completely avoid the re-labeling in XML updates also. In summary, Reuse is more appropriate to efficiently process the updates with both insertions and deletions (QED is more appropriate for the updates with insertions only).

No one has ever studied how to query different versions of the XML based on labeling schemes, therefore in this paper we propose algorithm(s) that never reuse the deleted labels; this truly maintains the different label versions (the labels in different versions are unique). The existing labeling schemes can not truly maintain the label versions since they must re-label the exiting nodes when many nodes are inserted.

References

1. R. Agrawal, A. Borgida, H. V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In Proc. of ACM SIGMOD, pages 253-262, 1989.
2. T. Amagasa, M. Yoshikawa, S. Uemura. QRS: A Robust Numbering Scheme for XML Documents. In *Proc. of ICDE*, pages 705-707, 2003.
3. S. Bose, L. Fegaras. Data Stream Management for Historical XML Data. In *Proc. of ACM SIGMOD*, pages 239-250, 2004.
4. S. Chien, V. J. Tsotras, C. Zaniolo. Efficient Management of Multiversion Documents by Object Referencing. In *Proc. of VLDB*, pages 291-300, 2001.
5. S.Y. Chien, V. Tsotras, C. Zaniolo, D. Zhang. Supporting Complex Queries on Multiversion XML Documents. In *ACM Trans. on Office Information Systems*, pages 1-42, 2005.
6. E. Cohen, H. Kaplan, T. Milo. Labeling Dynamic XML Trees. In *Proc. of PODS*, pages 271-281, 2002.
7. Changqing Li, Tok Wang Ling, Min Hu. Efficient Processing of Updates in Dynamic XML Data. To appear in *Proc. of ICDE*, 2006.

8. Changqing Li, Tok Wang Ling. QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. In *Proc. of CIKM*, pages 501-508, 2005.
9. Q. Li, B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of VLDB*, pages 361-370, 2001.
10. A. Marian, S. Abiteboul, G. Cobena, L. Mignet. Change-Centric Management of Versions in an XML Warehouse. In *Prof. of VLDB*, pages 581-590, 2001.
11. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom. Lore: A Database Management System for Semistructured Data. In *SIGMOD Record 26(3)*, pages 54-66, 1997.
12. S. Nestorov, J. D. Ullman, J. L. Wiener, S. S. Chawathe. Representative Objects: Concise Representations of Semistructured, Hierarchial Data. In *Prof. of ICDE*, pages 79-90, 1997.
13. P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Prof of ACM SIGMOD*, pages 903-908, 2004.
14. A. Silberstein, H. He, K. Yi, J. Yang. BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data. In *Proc. of ICDE*, pages 285-296, 2005.
15. I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. of ACM SIGMOD*, pages 204-215, 2002.
16. F. Wang, C. Zaniolo, X. Zhou, H.J. Moon. Managing Multiversion Documents & Historical Databases: a Unified Solution Based on XML. In *Proc. of WebDB*, pages 151-153, 2005.
17. F. Wang, X. Zhou, C. Zaniolo. An XML-Based Approach to Publishing and Querying the History of Databases. In *World Wide Web Journal*, pages 1-30, 2005.
18. X. Wu, M. L. Lee, W. Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *Proc. of ICDE*, pages 66-78, 2004.
19. M. Yoshikawa, T. Amagasa, T. Shimura, S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. In *ACM Trans. Internet Techn. 1(1)*, pages 110-141, 2001.
20. C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. of ACM SIGMOD*, pages 425-436, 2001.