EBSL: Supporting Deleted Node Label Reuse in XML*

Martin F. O'Connor and Mark Roantree

Interoperable Systems Group, School of Computing,
Dublin City University, Dublin 9, Ireland
{moconnor,mark.roantree}@computing.dcu.ie

Abstract. Recently, there has been much research into the specification of dynamic labeling schemes supporting XML updates. The primary design goal of any dynamic labeling scheme is to limit the growth rate in node label size, and consequently increase query performance and reduce update costs. The ability to reuse deleted node labels is a key property in achieving this goal. In this paper, we review the existing dynamic labeling schemes that provide this functionality and identify their shortcomings. We present our own dynamic labeling scheme that guarantees every delete node label can be reused. Further, we provide a deleted node label reuse strategy that best suits the nature of node insertions and deletions in an XML tree.

1 Introduction

In recent years, there has been considerable research on the development of new dynamic labeling schemes capable of supporting XML updates. To date, most of the analysis of such labeling schemes has been limited to the computation complexity of the update cost, the impact on label size under various update scenarios and to comparative performance analysis with other labeling schemes. In our previous work [15], we provided a holistic evaluation framework based on the desirable properties that are characteristic of a *good* dynamic labeling scheme for XML. In this paper, we focus on one such desirable property - the ability to reuse deleted node labels.

Almost all dynamic labeling schemes for XML published to date [16], [20], [2], [21], [11], [19], [4], [5], [9], [1] are not truly dynamic in that they support insertion updates of nodes only. When a node is deleted, the node label is marked as deleted. Subsequently, if we want to insert a new node at the same position in the XML tree as the previously deleted node, a new node label is generated. The deleted node label is not reused and is thus, wasted. In a highly dynamic environment with frequent node insertions and deletions, such as online transaction processing systems of heterogeneous data deployed in XML repositories, the inability to reuse deleted node labels leads to a rapid increase in label sizes. Large

* Funded by Enterprise Ireland Grant No. CFTD/07/201

node label sizes result in slower label comparison operations and consequently to slower query evaluations and slower update performance.

To incorporate deleted node label reuse as a property of a dynamic labeling scheme, is not a trivial task. Between any two consecutive nodes, there may have been an arbitrary number of node deletions. The ability to detect, identify and reclaim a deleted node label must be provided from the information encoded in the label alone. The labeling scheme may not rely on external indices to keep track of nodes as they are deleted and inserted. Furthermore, it is not sufficient to reclaim deleted node labels simply because they exist. A deleted node label may be longer in size than a newly generated label. In such scenarios, it is preferable to generate and insert the smaller node labels first and to only reuse the larger labels when no smaller labels are available. Consequently, it is necessary to identify the deleted label and determine the size of the deleted label in order to determine if it is suitable for reuse in the current scenario. All of this functionality must be provided from the information encoded in the labels alone while maintaining document order and guarantying that all nodes labels are unique throughout the XML tree whether they are newly generated or recently reused.

Contribution. In this paper, we present our new dynamic labeling scheme called Enhanced Binary String Labeling (EBSL) that provides full support for the reuse of deleted node labels during insertion operations. EBSL does not require the relabeling of existing nodes nor the recalculation of any values when inserting new nodes in an XML tree. To the best of our knowledge, EBSL is the only dynamic labeling scheme which guarantees that every deleted node label can be reused. Finally, EBSL supports a deleted label reuse strategy that best suits the nature of node deletions and node insertion in an XML tree.

This paper is structured as follows: in §2, we review the three dynamic labeling schemes that claim to support deleted node label reuse. In §3, we present our new dynamic labeling scheme, namely EBSL, and the underlying properties that facilitate deleted node label reuse. These properties ensure every deleted node label can be reused and the size of the reclaimed label can always be determined from the neighboring labels. In §4, we present our algorithms which perform the detection, identification and selection of the appropriate deleted node label for reuse. Finally in §5, our conclusions are presented.

2 Related Research

Several surveys have been performed that provide an overview and analysis of the principle dynamic labeling schemes for XML proposed to date [17], [6], [18], [15]. To the best of our knowledge, there are only three published dynamic labeling schemes for XML that support the reuse of deleted node labels while maintaining document order [12], [8], [13].

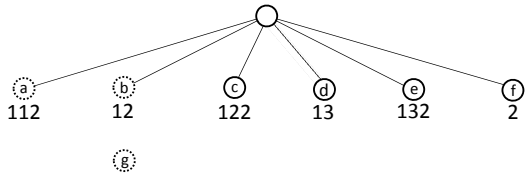

Fig. 1. Reusing a deleted node label in a QED labeled tree.

2.1 Extended QED Labeling Scheme

The Extended QED labeling scheme [12] uses a quaternary code to encode node labels. 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, 11. However, the number 0 is reserved as a separator and the remaining numbers 1, 2 and 3 are used to encode the node labels. The authors provide an algorithm `AssignInsertedCodeWithReuse` to extend the QED labeling scheme to support the reuse of deleted node labels and the algorithm has the property of always selecting the smallest deleted node label available.

Consider an XML tree which has been initially labeled with 16 nodes. The first 6 nodes are illustrate in Fig. 1. Now delete the first two nodes a and b. Insert a new node g before the current leftmost node c. According to algorithm `AssignInsertedCodeWithReuse`, the new node g will be assigned the label 112. The smallest deleted label (12) of node b was not reused.

Perhaps a more significant problem with algorithm `AssignInsertedCodeWithReuse` is that the same label is assigned to two different nodes. Consider an XML tree which has been initially labeled with 16 nodes. The first 4 nodes (a, b, c and d) are illustrated in Fig. 2. Insert a new node g to the left of the current leftmost node a. Then insert a new node h between node g and node a. The new node h will be assigned the label 1112. But this label has already been assigned to node g. The assignment of the same node label to two different nodes violates the properties of unique node identity as required by the XPath data model.

2.2 Improved Binary String Labeling Scheme

In [8], the authors propose a dynamic labeling scheme called IBSL (Improved Binary String Labeling) which supports node updates without the need to relabel existing nodes. IBSL, an extension of their earlier work [7], is a binary string

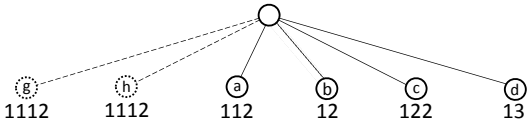

Fig. 2. Duplicate label assignment in a QED labeled tree.

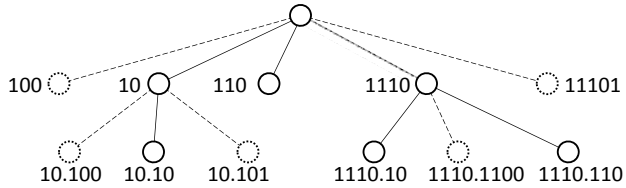

Fig. 3. IBSL Labeled XML tree.

prefix-based labeling scheme and introduces new insertion algorithms to permit the reuse of deleted node labels in their original position. Fig. 3 illustrates an IBSL labeled XML Tree, the dotted circles indicating newly inserted nodes in an existing tree.

In [8], the authors present an algorithm (Algorithm 3) to insert a new node label with the smallest length between two adjacent node labels. The algorithm is designed to process three generic cases: inserting a new leftmost node (case 1); inserting a new node between two adjacent nodes (case 2); and finally inserting a new rightmost node (case 3). The second case of inserting a new node between two consecutive nodes is broken down further into 3 subcases. Thus, five distinct case scenarios are presented in all. However, it can be shown that the algorithm fails to reuse deleted labels in four of these scenarios. We will highlight two of these now.

A comment on the convention we use is necessary at this point. Every node insertion is considered to be an insertion between two consecutive nodes. N_{left} is the node label on the left, N_{right} is the node label on the right, and N_{left} is lexicographically less than N_{right}. When inserting a new node to the right of the current rightmost node, N_{left} is said to be not empty and N_{right} is empty. The new node to be inserted is referred to as N_{new}. In our algorithms, we use the symbol $\oplus$ to denote the concatenation of two binary strings.

Case 1: : Inserting a new node to the left of the current leftmost node. Consider an XML tree which has been initially labeled with just two nodes a and b. Insert the following new nodes in the order they are listed: node c and node d, as illustrated in Fig. 4. Delete node c (the current leftmost node is now node d). Finally, insert a new node e to the left of the current leftmost node d. Node

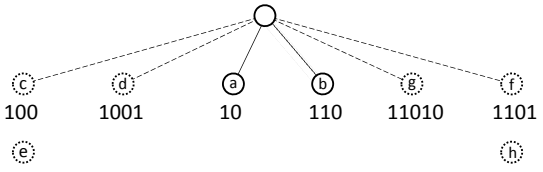

Fig. 4. Case 1 and 3: Inserting a new leftmost node and new rightmost node after a node deletion.

label e is assigned the label 10010 and not assigned the deleted label of node c (100). Thus, algorithm 3, case 1 did not reuse the deleted node label.

Case 3: : Inserting a new node to the right of the current rightmost node. Consider an XML tree which has been initially labeled with just two nodes a and b. Insert the following new nodes in the order they are listed: node f and node g, as illustrated in Fig. 4. Delete node f (the current rightmost node is now node g). Finally, insert a new node h to the right of the current rightmost node g. Node label $h = 110101$. There is a typographical error in case 3 [8]. However, there is a clear symmetry between case 1 and case 3 and we surmise the authors meant to write that N_{new} should be lexicographically greater than N_{left}. In either case, N_{new} does not reuse the deleted node label that belonged to node f (1101).

2.3 V-CDBS Labeling Scheme.

To the best of our knowledge, there is only one published dynamic labeling scheme, called V-CDBS [13], that supports the reuse of deleted node labels. V-CDBS is a binary string dynamic labeling scheme. Referring to the authors illustrated example of V-CDBS codes in Table 2 in [13] and using their algorithm 3 `AssignMiddleBinaryStringWithSmallestSize` [13], when N_{left}=0001 and N_{right}=001, inserting a new node will result in N_{new}=0001011. The valid label with the smallest size and lexicographically ordered between N_{left} and N_{right} is 00011 and is not reused.

The algorithm `AssignMiddleBinaryStringWithSmallestSize` can never gu arantee to *always* select the middle binary string with the smallest size due to the intrinsic properties of the assign initial labels algorithm employed by the V-CDBS labeling scheme. The V-CDBS assign initial labels algorithm is presented as algorithm 2 in [13]. It adopts a recursive divide-and-conquer approach whereby given N nodes to label, it first identifies and labels the middle node between 0 and N. The middle node is then used to divide the search space and the algorithm continues recursively. The middle node is selected using the formula: middle(start,end) = round(start + (end - start)/2). The first node is assumed to begin at position zero and consequently the round function guarantees that every node from 1 to N will be labeled. For example, middle(0,4) = round(0 + (4-0)/2) = 2. However, middle(0,3) = round(0 + (3-0)/2) = 2 also. The use of the round function introduces an approximation function into the V-CDBS assign initial labels algorithm. Thus, the V-CDBS labels are not assigned in a deterministic manner. In other words, the label value of node n is not and cannot be determined solely from the label values of node n+1 or node n-1. The V-CDBS encoding algorithm can guarantee lexicographical order but cannot guarantee the accurate calculation of the size of a node label n or indeed the label n itself, based solely on the node labels adjacent to node n. It follows that when node n is deleted, the V-CDBS labeling scheme cannot guarantee the accurate calculation of the deleted node label n (and its size), and consequently cannot guarantee that the deleted node label n will be reused.

3 Enhanced Binary String Labeling Scheme (EBSL)

In this section, we introduce our new dynamic labeling scheme for XML called the Enhanced Binary String Labeling scheme (EBSL). EBSL is based on the IBSL labeling scheme. EBSL does not require the relabeling of existing nodes nor the recalculation of any values when inserting new nodes in an XML tree. EBSL fully supports the reuse of deleted node labels when inserting new nodes into positions that previously contained deleted nodes. EBSL guarantees that every deleted node label can be reused. That is to say, there are no (simple or complex) insertion/deletion scenarios that will result in a deleted node label remaining unused when it would be appropriate to reuse that label. EBSL may be deployed using the prefix-based approach and thus, support ancestor-descendant, parent-child and sibling-ordered XPath evaluations. Finally, EBSL supports a deleted label reuse strategy that best suits the nature of node deletions and insertions in an XML tree. Conceptually, the construction of a dynamic labeling scheme for XML (that permits the reuse of deleted nodes labels) may be viewed as a three stage process:

1. The `AssignInitialLabels` stage.
2. The `SimpleInsertion` stage.
3. The `InsertionWithDeletedLabelReuse` stage.

Before we present the algorithms facilitating the construction process, it is necessary to introduce our customized definition of lexicographical order, a key property facilitating the reuse of deleted node labels.

3.1 Lexicographical Order

EBSL compares node labels using lexicographical order and not numerical order. Our customized definition of lexicographical order differs from existing approaches [10], [6], [14], [3], [21], [7], [13] and is also different to the definition employed by the IBSL labeling scheme [8].

Definition 1. *(Lexicographical order $\prec$) Given two consecutive binary strings S_{left} and S_{right} (S_{left} represents the left binary string, S_{right} represents the right binary string), S_{left} is said to be lexicographical equal to S_{right} iff they are exactly the same. S_{left} is said to be lexicographically less than S_{right} ($S_{left} \prec S_{right}$) iff*

1. *the lexicographical comparison of S_{left} and S_{right} is bit by bit from left to right. If the current bit of S_{left} is 0 and the current bit of S_{right} is 1, then $S_{left} \prec S_{right}$ and stop the comparison, or*
2. *$len(S_{left}) < len(S_{right})$, S_{left} is a prefix of S_{right}, the first extra bit of S_{right} = 1 (i.e.: substring(S_{right}, $len(S_{left})+1$, $len(S_{left})+1$) = 1), then $S_{left} \prec S_{right}$ and stop the comparison, or*
3. *$len(S_{left}) > len(S_{right})$, S_{right} is a prefix of S_{left}, the first extra bit of S_{left} = 0 (i.e.: substring(S_{left}, $len(S_{right})+1$, $len(S_{right})+1$) = 0), then $S_{left} \prec S_{right}$ and stop the comparison.*

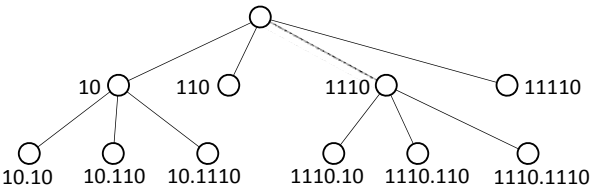

Fig. 5. An EBSL tree labeled using the `AssignInitialLabels` algorithm.

The conventional definition of lexicographical order defines a prefix string to be always lexicographically less than the larger string beginning with that prefix (e.g.: 110 $\prec$ 11001). In our definition of lexicographical order, (condition 3) the larger string containing the prefix is lexicographical less than the prefix string if and only if the subsequent bit immediately after the prefix in the larger string is a 0 bit (e.g.: 11001 $\prec$ 110). Conversely, (condition 2) the larger string containing the prefix is lexicographical greater than the prefix string if and only if the subsequent bit immediately after the prefix in the larger string is a 1 bit (e.g.: 110 $\prec$ 11010).

3.2 The Assign Initial Labels Algorithm

The EBSL `AssignInitialLabels` encoding algorithm is the same as the IBSL `AssignInitialLabels` algorithm [8] and thus is not detailed here. The algorithm takes as input a parent node, and assigns a unique label to every child node of the parent. The first child is always assigned the self_label 10. Thereafter, all subsequent children are deterministically labeled such that the self_label of child i is the computed as the concatenation of a 1 bit and the self_label of child i - *1*. The algorithm may be applied recursively to the XML tree to assign labels to every node in the tree. An example of an EBSL labeled tree is illustrated in Fig. 5.

Definition 2. *($\Re$label) A node label with the properties of a label assigned by the `AssignInitialLabels` algorithm is denoted as an $\Re$label. The properties that uniquely characterize an $\Re$label are:*

1. *The node label begins with a prefix binary string consisting of one or more consecutive 1 bits, and*
2. *The node will contain a single 0 bit and this 0 bit will be the last bit in the node label.*

Every node label assigned by the `AssignInitialLabels` algorithm is an $\Re$*label*. Definition 2 defines the unique characteristics of an $\Re$*label*. Examples of an $\Re$*label* are 10, 110, 1110, 11110, 111110 and so on. All $\Re$*label* node labels are lexicographically greater than the first child label 10. $\Re$*label* node labels will always and only ever be assigned when inserting a new node to the right of the current rightmost node. The ability to identify a node label with the properties

Algorithm 1: Simple Insertion Algorithm

input : left self_label N_{left}, right self_label N_{right}
output: New self_label N_{new} such that $N_{left} \prec N_{new} \prec N_{right}$

```
1 begin
2   Case 1: N_left is empty but N_right is not empty
    /* Insert a new node before the current leftmost node.                    */
3   N_new ⟵ N_right ⊕ 0 ;                          // ⊕ means concatenation.
4   Case 2: N_left is not empty but N_right is empty
    /* Insert a new node after the current rightmost node.                    */
5   N_new ⟵ 1 ⊕ N_left;
6   Case 3: N_left is not empty and N_right is not empty
    /* Insert a new node between two existing nodes.                          */
7   if (len(N_left) ≤ len(N_right)) then  N_new ⟵ N_right ⊕ 0;
8   else if (len(N_left) > len(N_right)) then  N_new ⟵ N_left ⊕ 1;
9 end
```

of an $\Re label$ is a key requirement in order to guarantee that every node label can be reclaimed and reused in the face of arbitrary nodes insertions and node deletions.

3.3 The Simple Insertion Algorithm

Algorithm 1 is the EBSL `SimpleInsertion` label encoding algorithm. The algorithm takes as input two node labels, N_{left} and N_{right}, and generates a new node label N_{new} such that $N_{left} \prec N_{new} \prec N_{right}$. The `SimpleInsertion` algorithm assumes no nodes have been deleted in the XML tree. This assumption is important so as to permit the clear specification of the rules governing the creation of a new node label when inserted between two existing consecutive node labels (and when no deleted labels are available to be reused).

It should also be noted that although cases 1 and 3 of our `SimpleInsertion` algorithm is the same as the IBSL simple insertion algorithm, case 2 is different. Concerning case 2, the IBSL simple insertion algorithm assigns $N_{new} = N_{left} \oplus 1$. Our `SimpleInsertion` algorithm case 2 assigns $N_{new} = 1 \oplus N_{left}$. This change fundamentally distinguishes the EBSL labeling scheme from the IBSL labeling scheme in a dynamic scenario, because new node insertions to the right of the current rightmost node will now end in a 0 bit, and not a 1 bit. This will directly influence lexicographical order evaluations and consequently the label values of new node inserted after the rightmost node. Algorithm 1 case 1 introduces a new category of labels called $\ell label$.

Definition 3. *($\ell label$) A node label with the properties of a label assigned by Case 1 of the `SimpleInsertion` algorithm (algorithm 1) is denoted as an $\ell label$. The properties that uniquely characterize an $\ell label$ are:*

1. *The node label begins with a single 1 bit, and*
2. *All subsequent bits in the node label consists of a sequence of two or more consecutive 0 bits.*

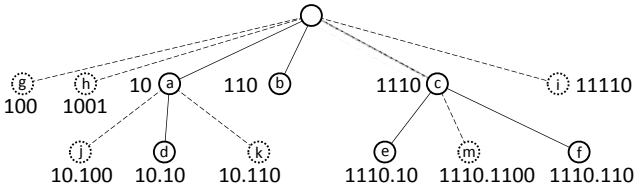

Fig. 6. An EBSL labeled tree with new nodes inserted (dotted circles) using the `SimpleInsertion` algorithm (algorithm 1).

Examples of an $\ell label$ are 100, 1000, 10000, 100000, 1000000 and so on. All $\ell label$ node labels are lexicographically less than the first child node 10. $\ell label$ node labels will only ever be assigned when inserting a new node to the left of the current leftmost node. The ability to identify a node label with the properties of an $\ell label$ is a key requirement in order to guarantee that every node label can be reused.

We provide an illustration of an EBSL labeled tree in Fig. 6 with nodes a through f assigned by the `AssignInitialLabels` algorithm and nodes g through m inserted in alphabetical order using the `SimpleInsertion` algorithm. The `SimpleInsertion` algorithm case 3 can never generate a new node label that has the characteristics of either an $\Re label$ or an $\ell label$. It should be observed that an arbitrary EBSL node label will be an $\Re label$ label or an $\ell label$ label or neither of the two. The three cases in the `SimpleInsertion` algorithm are closed (i.e.: both N_{left} and N_{right} cannot be empty).

4 Reusing Deleted Node Labels

In this section, we present our algorithms to facilitate the insertion of a new node into an XML tree while permitting the reuse of deleted node labels. Before we present these algorithms, it is good to summarize what we know thus far so as to identify the conditions in which these algorithms must operate.

An arbitrary node label in an EBSL labeled tree will always fall into one and only one of the following three lexicographical categories:

1. The node label will be lexicographically greater than ($\succ$) the node label 10.
2. The node label will be lexicographically less than ($\prec$) the node label 10.
3. The node label will be lexicographically equal to the node label 10.

When we consider a node insertion algorithm, there are always three high level scenarios to be processed:

1. Insertion of a new node after the current rightmost node.
2. Insertion of a new node before the current leftmost node.
3. Insertion of a new node between two existing nodes with non-empty labels.

In our EBSL deterministic labeling scheme, case 1 is the most important scenario and case 3 will always rely on case 1 to identify and reclaim a deleted

Algorithm 2: Insert New Node After Rightmost Node.

```
   /* This algorithm inserts a new node after the current rightmost node N_left and, if it
      exists, reuses the deleted node label to the right of N_left that was originally used
      to create N_left.                                                                  */
   input  : left self_label N_left, N_left is not empty
   output: New self_label N_new such that N_left ≺ N_new
 1 begin
 2     if (N_left == 10) then
 3         N_new ⟵ 1 ⊕ N_left ;                  // Apply SimpleInsertion algorithm, Case 2
 4         return N_new;
 5
       /* The Following IF statement is processed when N_left ≺ 10.                      */
 6     else if (prefix of N_left == 100) then
 7         N_new ⟵ SelectNewRightmostNodeLessThan10(N_left);
 8         return N_new;
 9
       /* The Following IF statement is processed when 10 ≺ N_left.                      */
10     else if (prefix of N_left == 11) then
11         N_new ⟵ SelectNewRightmostNodeGreaterThan10(N_left);
12         return N_new;
13     end
14 end
```

node label. Case 2 has a natural symmetry with case 1. Due to space restrictions, case 2 is not included in this paper. We first present the algorithms for case 1, followed by the algorithm for case 3.

4.1 Inserting a New Node after the Rightmost Node

Algorithm 2 is the EBSL `InsertNewNodeAfterRightmostNode` encoding algorithm that supports the reuse of deleted node labels. Algorithm 2 takes one node label as input, the non-empty left self_label N_{left}. This algorithm will output a new node label N_{new} such that N_{new} will be the reclaimed deleted node label to the right of N_{left} that was originally used to create N_{left} or a newly generated node label if there is no deleted node label available to be reused.

Essentially, the purpose of algorithm 2 is to determine the prefix of N_{left}, and based on the prefix, to call a function which will determine exactly what the new node label should be. If the node label N_{left} has the prefix 100, the function `SelectNewRightmostNodeLessThan10` will be invoked and will identify and reclaim a deleted node label if one exists, otherwise it will generate a new node label. If the node label N_{left} has the prefix 11, the function `SelectNewRightmostNodeGreaterThan10` will be invoked and will identify and reclaim a deleted node label if one exists, otherwise it will generate a new node label. The details of these functions will be presented later in this section.

4.2 Function SelectNewRightmostNodeLessThan10

Algorithm 3 contains the pseudocode of the function `SelectNewRightmostNode LessThan10`. This function receives as input the current rightmost node label

Algorithm 3: SelectNewRightmostNodeLessThan10.

```
   /* This algorithm takes as input the current rightmost node N_left such that N_left ≺
      10, and selects a deleted node label lexicographically greater than N_left. We are
      certain there exists at least one deleted node label to the right of N_left because
      the first self_label assigned by the AssignInitialLabels algorithm is always 10 and
      the current rightmost node N_left ≺ 10, therefore label 10 has been deleted.      */
   input : left self_label N_left, N_left is not empty, N_left ≺ 10
   output: New self_label N_new such that N_left ≺ N_new
 1 begin
   |  /* Remember N_left has prefix 100.                                              */
 2 |  if (lastbit of N_left == '0') then
 3 |  |  N_new ⟵ substring(N_left, 1, len(N_left) - 1);
 4 |
 5 |  else if (lastbit of N_left == '1') then
 6 |  |  N_temp ⟵ N_left;
 7 |  |  while (lastbit of N_temp == '1') do
   |  |  |  /* Remove all consecutive 1 bits from the end of label.                  */
 8 |  |  |  N_temp ⟵ substring(N_temp, 1, len(N_temp) - 1);
 9 |  |  end
10 |  |  N_new ⟵ substring(N_temp, 1, len(N_temp) - 1);
   |  |  /* Reclaim the deleted node label on RHS originally used to create N_left.  */
11 |  end
12 end
```

N_{left} with a non-empty self_label such that $N_{left} \prec 10$. The purpose of this function is to identify and reclaim the deleted node label (if it exists) to the right of N_{left} that was originally used to create N_{left}. If a deleted node label is not available to be reused, the function should generate a new node label. From a high level point of view, there are just two cases to consider when inserting a new node to the right of the current rightmost node N_{left} when $N_{left} \prec 10$:

1. The first case is when the node label N_{left} was itself created as a result of an insertion operation to the left of a leftmost $\ell label$ node and therefore N_{left} also has the properties of an $\ell label$ (e.g.: 100, 1000, 10000, 100000). In this case, N_{new} will be selected such that N_{new} is the $\ell label$ lexicographically greater than N_{left} where length(N_{left}) = m and length(N_{new}) = m-1. For example, if the current rightmost node label $N_{left} = 1000$, then $N_{new} = 100$.
2. The second case exploits the fact that all other node labels $\prec 10$ must have resulted from an insertion operation between two existing nodes with non-empty labels. Therefore, if N_{left} is the current rightmost node, and N_{left} was created as the result of an insertion operation between two nodes with non-empty labels, then we know for certain at least one deleted node label exist to the right of N_{left} (i.e.: the deleted node label to the right of N_{left} that was original used to create N_{left}). In this second case, we will reuse the deleted node label to the right of N_{left} that was originally used to create N_{left}.

Algorithm 4: SelectNewRightmostNodeGreaterThan10.

```
   /* This algorithm takes as input the current rightmost node N_left such that 10 ≺
      N_left.                                                                    */
   input  : left self_label N_left, N_left is not empty, 10 ≺ N_left
   output: New self_label N_new such that N_left ≺ N_new
 1 begin
       /* Remember N_left has prefix 11.                                        */
 2     if (lastbit of N_left == '0') then
 3         N_temp ⟵ substring(N_left, 1, len(N_left) - 1);
           /* Remove the last 0 bit from the label N_left.                      */
 4         if (AllBitsAreOne (N_temp)) then
               /* Confirms N_left is an ℜlabel.                                 */
 5             N_new ⟵ N_temp ⊕ 10;
               /* Insert new label according to simple insertion rules, Case 2. */
 6         else
 7             N_new ⟵ N_temp;
               /* Otherwise, reclaim deleted node label on RHS originally used to create
                  N_left by simply removing the last 0 bit (line 3).           */
 8         end
 9
10     else if (lastbit of N_left == '1') then
           /* In this case, the N_left label was originally created by appending a 1 bit
              to a label on the LHS of N_left. However, we want to find the deleted node
              label on the RHS that was originally used to create N_left. Therefore we mus
              first remove all consecutive 1 bits, and then finally remove the last 0 bit
              to obtain the deleted node label.                                */
11         N_temp ⟵ N_left;
12         while (lastbit of N_temp == '1') do
               /* Remove all consecutive 1 bits from the end of label.          */
13             N_temp ⟵ substring(N_temp, 1, len(N_temp) - 1);
14         end
15         N_new ⟵ substring(N_temp, 1, len(N_temp) - 1);
           /* Remove the last 0 bit to reclaim the deleted node label on RHS originally
              used to create N_left.                                           */
16     end
17 end
```

4.3 Function SelectNewRightmostNodeGreaterThan10

Algorithm 4 contains the pseudocode of the function `SelectNewRightmostNode GreaterThan10`. This function receives as input the current rightmost node label N_{left} with a non-empty self_label such that $10 \prec N_{left}$. The purpose of this function is to identify and reclaim the deleted node label to the right of N_{left} that was originally used to create N_{left}. If a deleted node label is not available to be reused, the function generates a new node label. From a high level point of view, there are just two cases to consider when inserting a new node to the right of the current rightmost node N_{left} when $10 \prec N_{left}$:

1. The first case is when the node label N_{left} was itself created as a result of an insertion operation to the right of a rightmost $\Re label$ and therefore N_{left} also has the properties of an $\Re label$ (e.g.: 110, 1110, 11110, 111110). In this case, N_{new} will be selected such that N_{new} is the smallest $\Re label$ lexicographically greater than N_{left}. For example, if the current rightmost node label $N_{left} =$ 110, then $N_{new} = 1110$.

Algorithm 5: Insert New Node Between Two Existing Nodes (Reuse)

```
/* This algorithm inserts a new node between two existing consecutive nodes with
   non-empty labels.                                                          */
input : left self_label N_left is not empty, right self_label N_right is not empty
output: New self_label N_new such that N_left ≺ N_new ≺ N_right
1 begin
2 |   N_temp ⟵ InsertNewNodeAfterRightmostNode (N_left);
3 |   if (N_temp ≺ N_right) then
4 |   |   N_new ⟵ N_temp;
5 |   else
6 |   |   N_new ⟵ SimpleInsertion (N_left, N_right);
7 |   end
8 |   return N_new
9 end
```

2. The second case exploits the fact that all other node labels $\succ 10$ must have resulted from an insertion operation between two existing nodes with non-empty labels. Therefore, if N_{left} is the current rightmost node, and N_{left} was created as the result of an insertion operation between two nodes with non-empty labels, then we know for certain there must be at least one deleted node label available to the right of N_{left} (i.e.: the deleted node label to the right of N_{left} that was originally used to create N_{left}). In this second case, we will reuse the deleted node label to the right of N_{left} that was originally used to create N_{left}.

4.4 Inserting a New Node between Two Existing Nodes with Non-Empty Labels

Algorithm 5 is the EBSL `InsertNewNodeBetweenTwoExistingNodes` encoding algorithm that supports the reuse of deleted node labels. Algorithm 5 initially invokes the function `InsertNewNodeAfterRightmostNode` passing the node label N_{left} as a parameter and returns a temporary node label N_{temp}. If N_{left} is an $\Re label$, then N_{temp} will be assigned the smallest $\Re label$ lexicographically greater than N_{left} (e.g.: if $N_{left} = 110$, then $N_{temp} = 1110$). If N_{left} is an $\ell label$, then N_{temp} will be assigned the $\ell label$ lexicographically greater than N_{left} such that length(N_{left}) = m and length(N_{temp}) = *m-1* (e.g.: if $N_{left} = 1000$, then $N_{temp} = 100$). If N_{left} is neither an $\Re label$ nor an $\ell label$, then N_{temp} will be assigned the node label to the right of N_{left} that was originally used to create N_{left} (e.g.: if $N_{left} = 11001$, then $N_{temp} = 110$). Finally, the node label N_{temp} will fall under one of three lexicographic conditions:

1. If the N_{temp} label is lexicographically less than N_{right}, then N_{temp} is a deleted node label and is available for reuse. Therefore, N_{new} is assigned the label of N_{temp}.
2. If the N_{temp} label is lexicographically equal to N_{right}, then the N_{temp} label is already in use and assigned to N_{right}. Therefore, N_{new} is assigned a new node label generated by the `SimpleInsertion` algorithm.

3. If the N_{temp} label is lexicographically greater than N_{right}, then there are no deleted node labels available between N_{left} and N_{right}. Therefore, N_{new} is assigned a new node label generated by the `SimpleInsertion` algorithm.

It may be observed that the `InsertNewNodeBetweenTwoExistingNodes` algorithm does not select the smallest deleted node label available between two given consecutive node labels. In an XML tree, when given κ nodes to insert, the κ nodes must be inserted in document order. The labeling scheme cannot arbitrarily decide the order in which to insert the nodes. By initially selecting the shortest deleted node label available, the V-CDBS labeling scheme ensures the node labels between N_{left} and the shortest deleted node label will remain unused when inserting a contiguous sequence of nodes between two consecutive node labels. Due to the deterministic labeling property of our EBSL labeling scheme, the `InsertNewNodeBetweenTwoExistingNodes` algorithm will always select the deleted node label immediately to the right of N_{left} if it exists. Otherwise, it will always select the $\Re label$ or $\ell label$ to the immediate right of N_{left} when N_{left} is an $\Re label$ or $\ell label$ respectively. Consequently the deterministic property of the EBSL labeling scheme always guarantees that every deleted node label can be reused. Holistically, the EBSL labeling scheme will always select the shortest deleted node labels available when inserting a sequence of nodes between two given nodes that contain a sequence of deleted node labels.

5 Conclusions

In this paper, we presented our Enhanced Binary String Labeling scheme (EBSL) supporting XML updates. EBSL does not require the relabeling of existing nodes nor the recalculation of any values when inserting new nodes in an XML tree. EBSL guarantees every deleted node label can be reused, all assigned nodes labels are unique and document order is maintained. EBSL supports a deleted label reuse strategy that best suits the nature of node insertions and node deletions in an XML tree.

As part of our future work, we will perform an analysis of the label size under various update scenarios and evaluate the computational complexity of our algorithms. We will also attempt to extract the underlying principles facilitating the reuse of deleted node label in our EBSL labeling scheme with the goal of specifying the core properties such that any binary string dynamic labeling scheme can support deleted node label reuse if they adapt their labeling scheme to encapsulate these properties. We will investigate the specification of an update operator to efficiently process bulk node insertions. This should be possible as every node is deterministically created based on the labels of the adjacent nodes. We will also investigate various label reuse strategies for the bulk update operator.

References

1. Alkhatib, R., Scholl, M.H.: Compacting XML Structures Using a Dynamic Labeling Scheme. In: BNCOD. pp. 158–170 (2009)
2. Amagasa, T., Yoshikawa, M., Uemura, S.: QRS: A Robust Numbering Scheme for XML Documents. In: ICDE. pp. 705–707 (2003)
3. An, D.C., Park, S.M., Park, S.: Efficient Secure Labeling Method under Dynamic XML Data Streams. In: IWSEC. pp. 246–260 (2008)
4. Böhme, T., Rahm, E.: Supporting Efficient Streaming and Insertion of XML Data in RDBMS. In: DIWeb. pp. 70–81 (2004)
5. Duong, M., Zhang, Y.: LSDX: A New Labelling Scheme for Dynamically Updating XML Data. In: ADC. pp. 185–193 (2005)
6. Härder, T., Haustein, M.P., Mathis, C., Wagner, M.: Node Labeling Schemes for Dynamic XML Documents Reconsidered. Data Knowl. Eng. 60(1), 126–149 (2007)
7. Ko, H.K., Lee, S.: An Efficient Scheme to Completely Avoid Re-labeling in XML Updates. In: WISE. pp. 259–264 (2006)
8. Ko, H.K., Lee, S.: A Binary String Approach for Updates in Dynamic Ordered XML Data. IEEE Trans. Knowl. Data Eng. 22(4), 602–607 (2010)
9. Kobayashi, K., Liang, W., Kobayashi, D., Watanabe, A., Yokota, H.: VLEI code: An Efficient Labeling Method for Handling XML Documents in an RDB. In: ICDE. pp. 386–387 (2005)
10. Li, C., Ling, T.W.: An Improved Prefix Labeling Scheme: A Binary String Approach for Dynamic Ordered XML. In: DASFAA. pp. 125–137 (2005)
11. Li, C., Ling, T.W.: QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. In: CIKM. pp. 501–508 (2005)
12. Li, C., Ling, T.W., Hu, M.: Reuse or Never Reuse the Deleted Labels in XML Query Processing Based on Labeling Schemes. In: DASFAA. pp. 659–673 (2006)
13. Li, C., Ling, T.W., Hu, M.: Efficient Updates in Dynamic XML Data: from Binary String to Quaternary String. VLDB Journal 17(3), 573–601 (2008)
14. Min, J.K., Lee, J., Chung, C.W.: An Efficient XML Encoding and Labeling Method for Query Processing and Updating on Dynamic XML Data. Journal of Systems and Software 82(3), 503–515 (2009)
15. O'Connor, M.F., Roantree, M.: Desirable Properties for XML Update Mechanisms. In: EDBT/ICDT Workshops (2010)
16. O'Neil, P.E., O'Neil, E.J., Pal, S., Cseri, I., Schaller, G., Westbury, N.: ORDPATHs: Insert-Friendly XML Node Labels. In: SIGMOD Conference. pp. 903–908 (2004)
17. Sans, V., Laurent, D.: Prefix based Numbering Schemes for XML: Techniques, Applications and Performances. PVLDB 1(2), 1564–1573 (2008)
18. Su-Cheng, H., Chien-Sing, L.: Node Labeling Schemes in XML Query Optimization: A Survey and Trends. IETE Technical Review 26, 88–100 (2009)
19. Thonangi, R.: A Concise Labeling Scheme for XML Data. In: International Conference on Management of Data (COMAD '06). Computer Society of India (December 2006)
20. Wu, X., Lee, M.L., Hsu, W.: A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In: ICDE. pp. 66–78 (2004)
21. Xu, L., Ling, T.W., Wu, H., Bao, Z.: DDE: From Dewey to a Fully Dynamic XML Labeling Scheme. In: SIGMOD Conference. pp. 719–730 (2009)