Efficient Label Encoding for Range-based Dynamic XML Labeling Schemes

Liang Xu, Tok Wang Ling, Zhifeng Bao, Huayu Wu

School of Computing, National University of Singapore
{xuliang, lingtw, baozhife, wuhuayu}@comp.nus.edu.sg

Abstract. Designing dynamic labeling schemes to support order-sensitive queries for XML documents has been recognized as an important research problem. In this work, we consider the problem of making range-based XML labeling schemes dynamic through the process of encoding. We point out the problems of existing encoding algorithms which include computational and memory inefficiencies. We introduce a novel Search Tree-based (ST) encoding technique to overcome these problems. We show that ST encoding is widely applicable to different dynamic labels and prove the optimality of our results. In addition, when combining with encoding table compression, ST encoding provides high flexibility of memory usage. Experimental results confirm the benefits of our encoding techniques over the previous encoding algorithms.

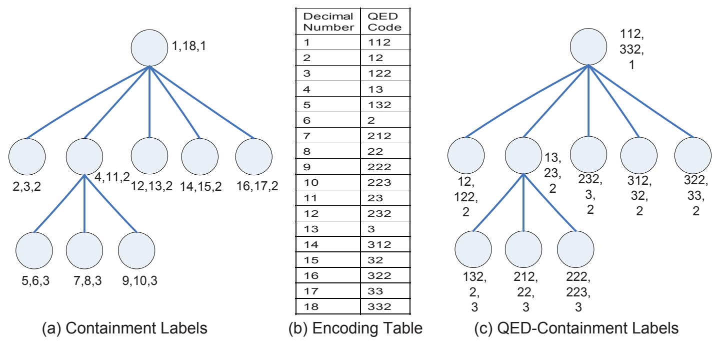
1 Introduction

XML is becoming an increasingly important standard for data exchange and representation on the Web and elsewhere. To query XML data that conforms to an *ordered tree-structured* data model, XML labeling schemes have attracted a lot of research and industrial attention for their effectiveness and efficiency. XML Labeling schemes assign the nodes in the XML tree unique labels from which their structural relationships such as ancestor/descendant, parent/child can be established efficiently.

Range-based labeling schemes[6, 11, 12] are popular in many XML database management systems. Compared with prefix labeling schemes[7, 2, 13], a key advantage of range-based labeling schemes is that their label size as well as query performance are not affected by the structure (depth, fan-out, etc) of the XML documents, which may be unknown in advance. Range-based labeling schemes are preferred for XML documents that are deep and complex, in which case prefix labeling schemes perform poorly because the lengths of prefix labels increase linearly with their depths. However, prefix labeling schemes appear to be inherently more robust than range-based labeling schemes. If negative numbers are allowed for local orders, prefix labeling schemes require re-labeling only if a new node is inserted between two consecutive siblings. Such insertions can be processed without re-labeling based on existing solutions[14, 9]. On the other hand, any insertion can trigger the re-labeling of other nodes with range-based labeling schemes.

The state-of-the-art approach to design dynamic range-based labeling schemes is based on the notion of *encoding*. It is also the only approach that has been proposed which can *completely* avoid re-labeling. By applying an encoding scheme to a range-based labeling scheme, the original labels are transformed to some dynamic format which can efficiently process updates without re-labeling. Existing encoding schemes include CDBS[4], QED[3, 5] and Vector[8] encoding schemes which transform the original labels to binary strings, quaternary strings and vector codes respectively. The following example illustrates the applications of QED encoding scheme to containment labeling scheme, which is the representative of range-based labeling schemes.

Example 1. In Figure 1 (a), every node in the XML tree is labeled with a containment label of the form: *start*, *end* and *level*. When QED encoding scheme is applied, the *start* and *end* values are transformed into QED codes based on the encoding table in (b). We refer to the resulting labels as QED-Containment labels which are shown in (c). QED-Containment labels not only preserve the property of containment labels, but also allows dynamic insertions with respect to lexicographical order[3].



Decimal Number	QED Code
1	112
2	12
3	122
4	13
5	132
6	2
7	212
8	22
9	222
10	223
11	23
12	232
13	3
14	312
15	32
16	322
17	33
18	332

Fig. 1. Applying QED encoding scheme to containment labeling scheme

Formally speaking, we consider an encoding scheme as a mapping f from the original labels to the target labels. Let X and Y denote the set of order-sensitive codes in the original labels and target labels respectively, f maps each element x in X to an element $y = f(x)$ in Y. For the mapping to be both correct and effective, f should satisfy the following properties:

1. **Order Preserving**: The target labels must preserve the order of the original labels, i.e. $f(x_i) < f(x_j)$ if and only if $x_i < x_j$ for any x_i, x_j $\in$ X.
2. **Optimal Size**: To reduce the storage cost and optimize query performance, the target labels should be of optimal size, i.e. the total size of $f(x_i)$ should be be minimized for a given range. To satisfy this property, f has to take the range to be encoded into consideration. The mappings may be different for different ranges.

The following example illustrates how this mapping in Figure 1 (b) is derived based on QED encoding scheme.

Example 2. To create the encoding table in Figure 1 (b), QED encoding scheme first extends the encoding range to (0, 19) and assigns two empty QED codes to positions 0 and 19. Next, the $(1/3)^{th}$ (6=round(0+(19-0)/3)) and $(2/3)^{th}$ (13=round(0+(19-0)$\times$2/3)) positions are encoded by applying an insertion algorithm with the QED codes of positions 0 and 19 as input. The QED insertion algorithm takes two QED codes as input and computes two QED codes that are lexicographically between them which are as short as possible (Such insertions are always possible because QED codes are dynamic). The output QED codes are assigned to the $(1/3)^{th}$ and $(2/3)^{th}$ positions which are then used to partition range (0, 19) into three sub-ranges. This process is recursively applied for each of the three sub-ranges until all the positions are assigned QED codes. CDBS and Vector encoding schemes adopt similar algorithms.

We classify these algorithms i.e. CDBS, QED and Vector, as *insertion-based* algorithms since they make use of the property that the target labels allow dynamic insertions. However, a drawback of the insertion-based approach is that by assuming the entire encoding table fits into memory, it may fail to process large XML documents due to memory constraint. Since the size of the encoding table can be prohibitively large for large XML documents and main memory remains the limiting resource, it is desirable to have a memory efficient encoding algorithm. Moreover, the insertion-based approach requires costly table creation for every range, which is computationally inefficient for encoding multiple ranges of multiple documents.

In this paper, we show that only a single encoding table is needed for the encoding of multiple ranges. As a result, encoding a range can be translated into indexing mapping of the encoding table which is not only very efficient, but also has an adjustable memory usage. The main contributions of this paper include:

– We propose a novel Search Tree-based (ST) encoding technique which has a wide application domain. We illustrate how ST encoding technique can be applied to binary string, quaternary string and vector code and prove the optimality of our results.
– We introduce encoding table compression which can be seamlessly integrated into our ST encoding techniques to adapt to the amount of memory available.
– We propose Tree Partitioning (TP) technique as an optimization to further enhance the performance of ST encoding for multiple documents.
– Experimental results demonstrate the high efficiency and scalability of our ST encoding techniques.

2 Preliminary

2.1 Range-based Labeling Schemes

In containment labeling scheme, every label is of the form (*start*, *end*, *level*) where *start* and *end* define an interval and *level* refers to the level in the

XML document tree. Assume node n has label $(s1, e1, l1)$ and node m has label $(s2, e2, l2)$, n is an ancestor of m if and only if $s1 < s2 < e2 < e1$. i.e. interval $(s1, e1)$ contains interval $(s2, e2)$. n is the parent of m if and only if n is an *ancestor* of m and $l1 = l2 - 1$. Other range-based labeling schemes[11, 12] have similar properties.

Example 3. In Figure 1 (a), node(1,18,1) is an ancestor of node(7,8,3) because 1<7<8<18. Node(4,11,2) is the parent of node(5,6,3) because 4<5<6<11 and 2=3-1.

Although range-based labeling schemes work well for *static* XML documents, insertions of new nodes may lead to costly re-labeling. Leaving gaps[12] only allows limited number of insertions before re-labeling is required. Floating point numbers have been suggested to be used[1]. However, the precision of floating point number is limited by the fixed number of bits in its mantissa. As a result, re-labeling is still necessary when the number of insertions exceeds certain limits.

2.2 Dynamic Formats

Dynamic formats proposed in the literature include binary strings that end with 1[4], quaternary strings that end with 2 or 3[3] and vector codes[8]. They are dynamic in the sense that arbitrary insertions can be made between two consecutive codes without affecting other codes. We use binary strings to illustrate the property of dynamic formats. We include the descriptions of quaternary strings and vector codes in the extended version of this paper[10].

Definition 1. ***(Binary String)*** *Given a set of binary numbers $A = \{0, 1\}$ where each number is stored with 1 bit. A binary string is a sequence of elements in A.*

Binary strings are compared based on lexicographical order. The following theorem formalizes the dynamic property of binary strings that end with 1.

Theorem 1. *Given two binary strings C_l and C_r which both end with 1 such that C_l precedes C_r in lexicographical order (denoted as $C_l \prec C_r$), we can always find C_m which also ends with 1 and $C_l \prec C_m \prec C_r$.*

Theorem 1 can be proved based on Algorithm 1.

Example 4. Given three binary strings 01, 11 and 111, it follows from lexicographical order that 01 $\prec$ 11 $\prec$ 111. Insertion between 01 and 11 will produce 011, since length(01) $\geq$ length(11) (01$\oplus$1, Algorithm 1 line 2). And insertion between 11 and 111 gives 1101, since length(11) $<$ length(111) (111 with the last 1 change to 01, Algorithm 1 line 4).

3 ST Encoding Technique

In this section, we present the details of our ST encoding technique which can be applied to binary string, quaternary string and vector codes, and are called STB, STQ and STV encoding schemes respectively.

Algorithm 1: InsertBinaryString(C_l, C_r)

```
  Data: $C_l$ and $C_r$ which are both binary strings that end with 1 and $C_l \prec C_r$
  Result: $C_m$ which ends with 1 and $C_l \prec C_m \prec C_r$
1 if $length(C_l) \geq length(C_r)$ then
2     $C_m = C_l \oplus 1$                         /* $\oplus$ means concatenation */;
3 end
4 else $C_m = C_r$ with the last number 1 change to 01;
5 return $C_m$;
```

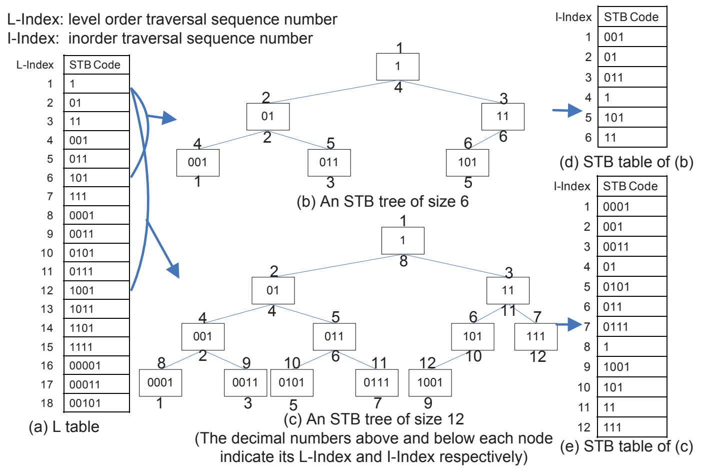
3.1 ST-Binary (STB)

Data structure Our STB encoding is based by the data structure we call STB tree. An **STB tree** is a complete binary tree where each node is associated with a binary string that ends with 1, which we refer to as an STB code. The STB code of the root is 1.

Given a node n in the STB tree, the STB code of its left child lc and right child rc can be derived as follows:

- C_{lc}=C_n with the last 1 replaced with 01
- C_{rc}=$C_n \oplus$ 1 ($\oplus$ means concatenation)

Two STB trees with 6 and 12 nodes are shown in Figure 2 (b) and (c).


L-Index: level order traversal sequence number
I-Index: inorder traversal sequence number
(b) An STB tree of size 6
(c) An STB tree of size 12
(The decimal numbers above and below each node indicate its L-Index and I-Index respectively)

(a) L table

L-Index	STB Code
1	1
2	01
3	11
4	001
5	011
6	101
7	111
8	0001
9	0011
10	0101
11	0111
12	1001
13	1011
14	1101
15	1111
16	00001
17	00011
18	00101

(d) STB table of (b)

I-Index	STB Code
1	001
2	01
3	011
4	1
5	101
6	11

(e) STB table of (c)

I-Index	STB Code
1	0001
2	001
3	0011
4	01
5	0101
6	011
7	0111
8	1
9	1001
10	101
11	11
12	111

Fig. 2. STB encoding of two ranges 6 and 12

Lemma 1. *The left subtree of a node n contains only STB codes lexicographically less than C_n; The right subtree of n contains only STB codes lexicographically greater than C_n.*

Proof. [**Sketch**] Given any STB code n which is a binary string that ends with 1, we denote C_n as "$S1$" where "S" is a binary string or an empty string. It follows that C_{lc}="$S01$" and similarly, $C_{lc.lc}$="$S001$" and $C_{lc.rc}$="$S011$". Now it is easy to see that all the STB codes in the left subtree have "$S0$" as their prefix. Since "$S0$" precedes "$S1$" in lexicographical order, all the STB codes in the left subtree are lexicographically less than C_n. The rest of the lemma follows similarly.

Theorem 2. *An STB tree is a binary search tree based on lexicographical order.*

Proof. Theorem 2 follows directly from Lemma 1.

An **L table** stores the STB codes of an STB tree in order of *level order traversal*. We denote the index of an L table as **L-Index** and use L to denote the set of decimal numbers in L-Index. An important observation about L table is that it can be shared by STB trees of different sizes: the first m rows of the L table represents an STB tree of size m in level order. An **STB table** stores the STB codes of an STB tree in order of *inorder traversal*. We denote the index of an STB table as **I-Index** and use I to denote the set of decimal numbers in I-Index.

Example 5. Consider the STB tree of size 6 in Figure 2 (b). If we order its STB codes according to level order traversal sequence, they match the first 6 rows of the L table in (a). Ordering the codes in order of inorder traversal sequence would produce the STB table in (d). Similar observation can be made for the STB tree in (c).

Algorithms To encode a range m with STB encoding is to realize the mappings represented by an STB table of size m. Intuitively, this can be achieved by traversing the STB tree of size m in inorder.

Formally speaking, STB encoding defines a mapping $f : I \rightarrow B$ where B denotes the set of STB codes. More specifically, f is established through two levels of mappings: $f(i) = h(g(i))$ where $g : I \rightarrow L$ and $h : L \rightarrow B$. Deriving h is straight forward from the L table. Depending on the range to be encoded, the size of L table can be extended dynamically. How g can be established is shown in Algorithm 2 which is based on inorder traversal of a binary tree. First a stack *path* is initialized to store the L-Indices of a root-to-leaf path(line 1). Then we proceed to call Function **PushLeftPath** which pushes the L-Index of the leftmost path (starting from the root) into *path* (line 2). For each $i \in I$, we map i to the top element in *path* (Recall that during an inorder traversal, the leftmost element is always visited first). Then the L-Index of the leftmost path that starts from the right child of the top element is pushed into *path* (line 3 to 6).

Next we show that STB encoding is order preserving and of optimal size.

Theorem 3. *Given a range m and any two numbers j and k such that $1 \leq j < k \leq m$, it follows that $C_j \prec C_k$ where C_j and C_k denote the STB codes transformed from j and k based on STB encoding.*

```
Algorithm 2: ItoLMapping($m$)
  Data: $m$ which is the range to be encoded.
  Result: The mapping from I-Index to L-Index stored in an array $ItoL[1 \ldots m]$.
1 Initialize Stack $path$;
2 PushLeftPath($path$, $1$, $m$);
3 for $i$=$1$ to $m$ do
4     $l$=$path$.Pop();
5     $ItoL[i] = l$;
6     PushLeftPath($path$, $2 \times l + 1$, $m$)        /* $2 \times l + 1 \longrightarrow$ right child */
7 end
```

```
Function PushLeftPath($path$, $l$, $m$)
  while $l \leq m$ do
      $path$.Push($l$);
      $l = 2 \times l$                                 /* $2 \times l \longrightarrow$ left child */
  end
```

Proof. Since an STB tree is a binary search tree (Theorem 2), an inorder traversal of the STB tree visits the STB codes in increasing lexicographical order. In other words, STB encoding is order preserving.

Lemma 2. *Level i of an STB tree has 2^{i-1} STB codes (except possibly the last level) of length i. (Assume the root is of level 1).*

Lemma 2 easily follows from the properties of STB trees.

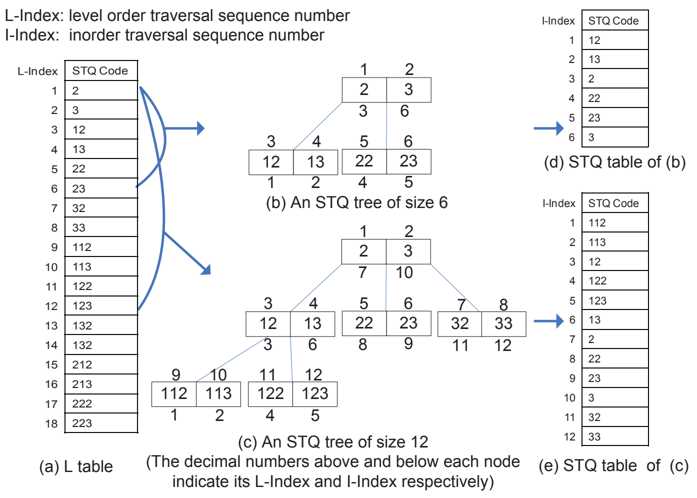
Since an STB code is a binary string that ends with 1, there are 2^{i-1} possible STB codes of length i. From Lemma 2, we can see that an STB tree has all the possible STB codes of length i at level i (except possibly the lowest level). The fact that an STB tree is a complete binary tree implies that STB codes with length i are always used up before STB codes with length i+1 are used. Therefore STB encoding produces labels with optimal size.

3.2 ST-Quaternary (STQ)

We illustrate our STQ encoding scheme using the data structure we call STQ tree. An **STQ tree** is a complete ternary tree. Each node of the STQ tree is associated with two STQ codes: left code (L) and right code (R) where $R = L$ with the last number 2 change to 3. L and R of the root are 2 and 3 respectively.

Given a node n in the STQ tree, the left code of its left child (lc), middle child (mc) and right child (rc) can be derived as follows:

- L_{lc}= L_n with the last number 2 change to 12;
- L_{mc}= $L_n \oplus 2$ ($\oplus$ means concatenation);
- L_{rc}= $R_n \oplus 2$.


L-Index: level order traversal sequence number
I-Index: inorder traversal sequence number
(a) L table
(b) An STQ tree of size 6
(c) An STQ tree of size 12
(The decimal numbers above and below each node indicate its L-Index and I-Index respectively)
(d) STQ table of (b)
(e) STQ table of (c)

L-Index	STQ Code
1	2
2	3
3	12
4	13
5	22
6	23
7	32
8	33
9	112
10	113
11	122
12	123
13	132
14	132
15	212
16	213
17	222
18	223

I-Index	STQ Code
1	12
2	13
3	2
4	22
5	23
6	3

I-Index	STQ Code
1	112
2	113
3	12
4	122
5	123
6	13
7	2
8	22
9	23
10	3
11	32
12	33

Fig. 3. STQ Encoding of two ranges 6 and 12

For every node, we have $R = L$ with the last number 2 change to 3.

Two STQ trees with 6 and 12 codes are shown in Figure 3 (b) and (c).

Lemma 3. *The left subtree of a node n contains only STQ codes lexicographically less than L_n; The middle subtree of n contains only STQ codes lexicographically between L_n and R_n; The right subtree of n contains only STQ codes lexicographically greater than R_n.*

The proof is similar to that of Lemma 1, so we omit it here. Given Lemma 3, an STQ tree can be seen as a search tree if we define the inorder traversal sequence to be in order of: (1) Traverse the left subtree; (2) Visit L of the root; (3) Traverse the middle subtree; (4) Visit R of the root and (5) Traverse the right subtree. In this way, we can define **I-Index**, **L-Index**, **STQ table** and **L table** similar to those of STB tree.

STQ encoding defines the mapping from I-Index to STQ codes which is achieved through two levels of mappings: from I-Index to L-Index and from L-Index to STQ codes. As shown in Figure 3, the mappings from L-Index to STQ codes are stored a single L table (a) which can be shared by multiple ranges. The mappings from I-Index to L-Index can be derived from Algorithm 4 which performs an inorder traversal of the STQ tree.

The correctness of our STQ encoding algorithms follows from the fact that its inorder traversal visits the STQ codes in increasing lexicographical order. The resulting label size is also optimal because our algorithm favors STQ codes with smaller lengths.

Algorithm 4: ItoLMapping(m)

```
Data: $m$ which is range to be encoded.
Result: The mapping from I-Index to L-Index stored in an array $ItoL[1 \ldots m]$.
Initialize Stack $path$;
PushLeftPath($path$, $1$, $m$);
for $i$=$1$ to $m$ do
    $l$=$path$.Pop();
    $ItoL[i] = l$;
    if $l$ mod $2$ =$1$ then                                  /* $l \longrightarrow lcode$ */
        PushLeftPath($path$, $3 \times l + 2$, $m$)    /* $3 \times l + 2 \longrightarrow$ middle child */
    else                                                       /* $l \longrightarrow rcode$ */
        PushLeftPath($path$, $3 \times l + 1$, $m$)     /* $3 \times l + 1 \longrightarrow$ right child */
    end
end
```

Function PushLeftPath($path$, l, m)

```
while $l \leq m$ do
    $path$.Push($l + 1$);
    $path$.Push($l$);
    $l = 3 \times l$                                      /* $3 \times l \longrightarrow$ left child */
end
```

3.3 ST-Vector (STV)

Our STV encoding scheme is based on the data structure we call STV tree. It is a complete binary tree where each node is associated with a vector code: C. The vector codes of the root, its left child and right child are (1,1), (2,1) and (1,2) respectively.

Given a node n and its parent p in the STV tree, the vector codes of its left child (lc) and right child (rc) can be derived as follows: If n is the left child of p, C_{lc}=2 $\times$ C_n - C_p; C_{rc}=C_n + C_p; Else, C_{lc}=C_n + C_p; C_{rc}=2 $\times$ C_n - C_p. An example of STV tree is shown in Figure 4.

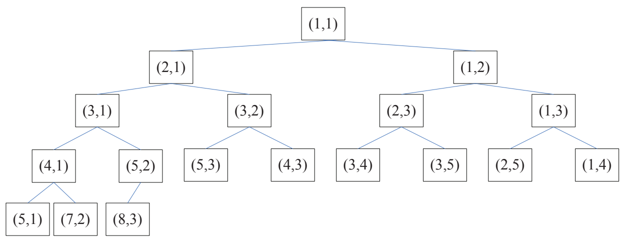


Fig. 4. STV tree

Theorem 4. *An STV tree is a binary search tree based on vector order.*

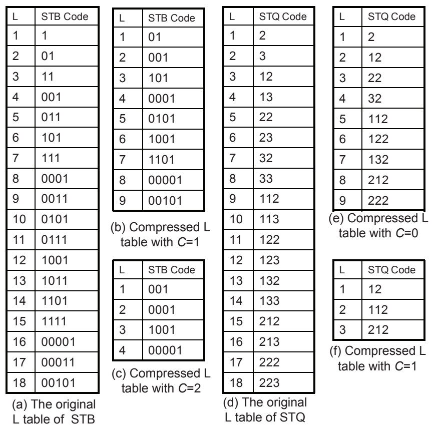
The proof is based on mathematical induction, we omit it here. Given the STV tree, we can define L table similar to that of STB encoding which stores the mapping from L index to Vector codes. Moreover, since STV tree is a binary search, Algorithm 2 can be directly applied to derive the mapping from I to L index. We ignore the details of STV encoding since it is similar to STB encoding.

3.4 Comparison with insertion-based approach

Compared with the insertion-based approach, our design of ST encoding as a two level mapping has the following advantages: (1) Since $h : L \rightarrow STB/STQ/STV code$ remains the same for different ranges, the cost of encoding a new range is only to compute $g : I \rightarrow L$. By sharing h for different ranges, we avoid costly table creation for every range; (2) Compression technique can be conveniently applied to L table to provide high flexibility of memory usage (Section 4). The compression technique is easily incorporable because compressing L table only affects h while h and g are independent of each other; (3) By exploiting the common mappings of different ranges, we can further speed up the encoding of multiple ranges (Section 5).

4 Encoding Table Compression

The L table of STB is shown in Figure 5 (a). Considering its STB codes with indices from 2 onwards, we can see that every STB code at index $2i + 1$ can be deduced from the STB code at index $2i$ by changing the second last number to 1. Therefore we can compress this L table to half by only retaining the rows with even indices ((b)). Thus, the mapping from L-Index to STB codes for becomes:



L	STB Code
1	1
2	01
3	11
4	001
5	011
6	101
7	111
8	0001
9	0011
10	0101
11	0111
12	1001
13	1011
14	1101
15	1111
16	00001
17	00011
18	00101

(a) The original L table of STB

L	STB Code
1	01
2	001
3	101
4	0001
5	0101
6	1001
7	1101
8	00001
9	00101

(b) Compressed L table with C=1

L	STB Code
1	001
2	0001
3	1001
4	00001

(c) Compressed L table with C=2

L	STQ Code
1	2
2	3
3	12
4	13
5	22
6	23
7	32
8	33
9	112
10	113
11	122
12	123
13	132
14	133
15	212
16	213
17	222
18	223

(d) The original L table of STQ

L	STQ Code
1	2
2	12
3	22
4	32
5	112
6	122
7	132
8	212
9	222

(e) Compressed L table with C=0

L	STQ Code
1	12
2	112
3	212

(f) Compressed L table with C=1

Fig. 5. Compress L tables of STB and STQ by factors of 2^C and 2×3^C respectively

$$h(l) \rightarrow \begin{cases} LTable[l/2] & ,when\ l\ mod\ 2 = 0 \\ LTable[\lfloor l/2 \rfloor] with\ the\ sec\text{-} \\ ond\ last\ number\ change\ to\ 1 & ,when\ l\ mod\ 2 = 1 \end{cases} \tag{1}$$

The table in (b) can be further compressed by a factor of 2 if we consider the STB codes with indices from 2 onwards. We exclude the STB codes with odd indices since they can be derived from the STB codes with even indices by changing the *third* last number to 1 ((c)). In this way, we can compress the L table of STB by factors of $2, 4, 8 \ldots 2^C$ and we denote C as the compression factor.

By analyzing the L table of STQ in Figure 5 (d), the straight forward compression is to exclude the STQ codes with even indices since they can be derived from the STQ codes with odd indices by changing the last 2 to 3 ((b)). Therefore the mapping from L-Index to STQ codes becomes:

$$h(l) \rightarrow \begin{cases} LTable[\lceil l/2 \rceil] & ,when\ l\ mod\ 2 = 1 \\ LTable[l/2]\ with\ the \\ last\ number\ change\ to\ 3 & ,when\ l\ mod\ 2 = 0 \end{cases} \tag{2}$$

Consider the table in Figure 5 (e), it can be further compressed by a factor of 3 if we consider the STQ codes from index 2 onwards. The STQ codes at indices $3i$ and $3i + 1$ can be derived from the STQ code at index $3i - 1$ by changing the second last number to 2 and 3. Therefore we exclude the STQ codes at indices $3i$ and $3i + 1$ and the resulting table is shown in (f). In summary the L table of STQ can be compressed by factors of $2, 6, 18 \ldots 2 \times 3^C$.

The L table of STV can be compressed by a factor of 2 based on the bilateral symmetry we observe in the STV tree (Figure 4). Further compression is possible based on the symmetry at lower levels. Overall we can achieve compression factors of 2^C.

5 Tree Partitioning (TP)

We introduce Tree Partitioning (TP) as an optimization to further enhance the performance of ST encoding technique. We use STB tree to illustrate the idea of TP. Our optimization technique can be easily adapted for STQ and STV trees.

STB encoding technique, as we have shown, is a mapping $f(i) = h(g(i))$ where $g : I \rightarrow L$ and $h : L \rightarrow B$. Since h remains the same for different ranges, the cost of encoding a range is dominated by g. The motivation for TP optimization is that, given multiple ranges to be encoded, the computational cost of g can be reduced if we can exploit the common mappings for ranges that are close to some extent.

Suppose there are two STB trees T of size s_1 and T' of size s_2 (without loss of generality, we assume $s_1 < s_2$), we analyze the common mapping of the two trees when they have the same height, say k, i.e. $2^k \leq s_1 < s_2 < 2^{k+1}$.

Our TP algorithm divides T' into three partitions:

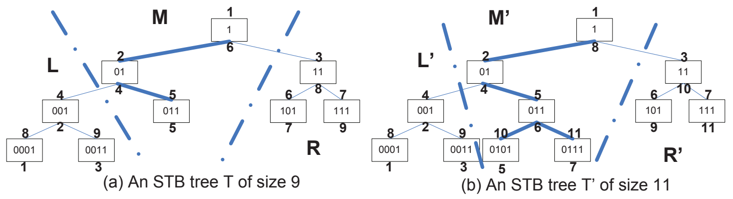


Fig. 6. TP Optimization

L partition All the nodes on the left of the path from the root to the node with L-Index=$s_1 + 1$.

R partition All the nodes on the right of the path from the root to the node with L-Index=s_2

M partition The rest of the nodes in the STB tree

T is also divided into three partitions: L', R' and M'. L' and L partitions have the same L-Index and so do R' and R partitions. And the rest of the nodes fall into M'. g in L and L' partitions are the same as the two partitions overlap and are visited first during inorder traversal. If we increase all the I-Index in R by $s_2 - s_1$, g in R and R' also coincide.

Example 6. Two STB trees T and T' in Figure 6 (a) and (b) are partitioned based on our TP algorithm. In the resulting partitions, g in L and L' are the same. g in region R can be derived from that in R' if we increase the L-Index in R by $11 - 9 = 2$.

Since both M and M' bounded by two root-to-leaf paths, Algorithm 2 can be easily modified to compute the mappings in them (an intermediate state can be calculated based on direct calculation which is available in [10]). By partitioning the range to be encoded, we can re-use some of the previously-computed mappings and avoid re-computing g for the whole range.

6 Experiments and Results

In this section, we experimentally evaluate and compare the various encoding techniques developed in this paper against the insertion-based encoding schemes including CDBS, QED and Vector. The comparison of CDBS, QED and Vector with the previous labeling scheme are beyond the scope of this paper and can be found in [5, 8].

We used data sets from XMark benchmark, Treebank, SwissProt and DBLP datasets for our experiments. The characteristic of these data sets are shown in Table 1. We used JAVA for our implementation and our experiments are performed on Pentium IV 3 GHz with 1G of RAM running on windows XP.

Data set	Max/average fan-out	Max/average depth	No. of nodes
XMark	25500/3242	12/6	179689
Treebank	56384/1623	36/8	1666315
SwissProt	50000/301	5/3	2437666
DBLP	328858/65930	6/3	3332130

Table 1. Test data sets

6.1 Encoding Time

First we evaluate the encoding time of these encoding schemes using containment labels of the XMark data set. We randomly generated 80 XMark documents whose sizes range from 1 MB to 90 MB. In Figure 7, we observe clear time difference between ST encodings and insertion-based encodings: our STB and STV encodings are both approximately 3 times faster than CDBS and Vector encoding; Moreover, our STQ encoding is approximately 7 times faster than QED encoding. The reason is clear from the comparison of algorithms: insertion-based encodings need to create an encoding table for every range, which is significantly slower than our ST encodings that perform index mapping of a single table. The advantages of ST encoding are more significant when we apply TP optimization which exploits common mappings of encoding multiple ranges. Overall ST encodings with TP are by a factor of 5-11 times faster than insertion-based encodings for containment labels. The results confirm that our ST encoding techniques are highly efficient for encoding multiple ranges and substantially surpass the insertion-based encodings.

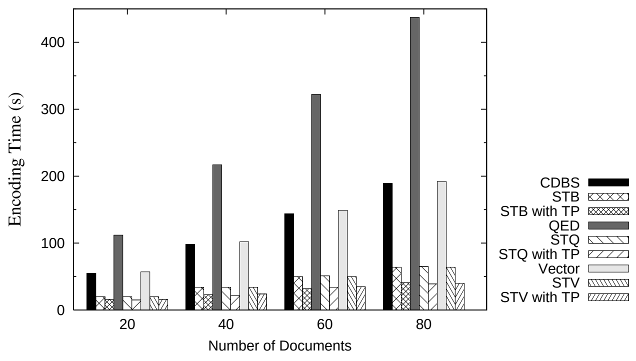
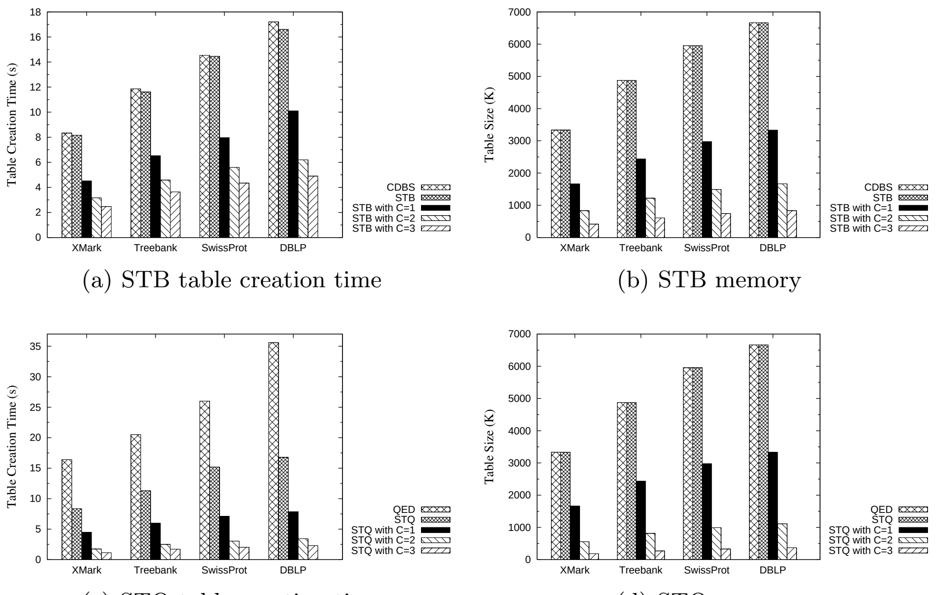


Fig. 7. Encoding containment labels of multiple documents

6.2 Memory Usage and Encoding Table Compression

We compare the memory usage of different algorithms which is dominated by the size of the encoding tables and the results are shown in Figure 8. Without any compression, the table size of STB and CDBS are the same, and so are their table creation times. However, unlike CDBS whose table size is fixed, our STB encoding can adjust its table size by varying the compression factor C. A larger C yields a smaller table size and less table creation time. Similar observation can be made in Figure 8 (c) and (d) for quaternary strings. The table creation time of STQ is less than that of QED due to the complexity of the QED insertion algorithms. By adjusting the compression factor, our ST encoding can process large XML data sets with limited memory available.



(a) STB table creation time

(b) STB memory

(c) STQ table creation time

(d) STQ memory

Fig. 8. Encoding table compression

6.3 Label size and query performance

We empirically evaluate the label size and query performance of different labeling schemes. We have proved that both STB and STQ encodings produce labels of optimal sizes. The labels of vector and STV encoding schemes are stored as UTF8 strings. From our experimental results, their label sizes may differ by a small amount which is overall negligible, so we ignore the diagrams here. Moreover, since the labels produced by ST encoding and its insertion-based counterpart are of the same format, their query performance is also the same. In summary, the labels produced by our ST encoding techniques are of optimal quality.

7 Conclusion

In this paper, we take the initiative to address the problem of efficient label encoding. We propose ST encoding technique which can be applied to range-based labeling schemes to produce dynamic labels. We show that ST encoding technique is highly efficient and has a wide application domain. Compared with insertion-based encodings which are main memory-based and have fixed memory requirements, our ST encoding technique has an adjustable memory usage and is therefore able to process very large XML documents with limited memory available. An interesting future research direction is to explore more dynamic formats and study how the application scope of ST encoding could be extended to these formats.

References

1. T. Amagasa and M. Yoshikawa and S. Uemura. QRS: A Robust Numbering Scheme for XML Documents. In ICDE, 2003.
2. E. Cohen and H. Kaplan and T. Milo. Labeling Dynamic XML Trees. In SPDS, 2002.
3. C. Li and T. W. Ling. QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. In CIKM, 2005.
4. C. Li and T. W. Ling and M. Hu. Efficient Processing of Updates in Dynamic XML Data. In ICDE, 2006.
5. C. Li and T. W. Ling and M. Hu. Efficient Updates in Dynamic XML Data: from Binary String to Quaternary String. In VLDB J., 2008.
6. C Zhang and J. F. Naughton and D. J. DeWitt and Q. Luo and G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In SIGMOD, 2001.
7. I. Tatarinov and S. Viglas and K. S. Beyer and J. Shanmugasundaram and E. J. Shekita and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In SIGMOD, 2002.
8. L. Xu and Z. Bao and T. W. Ling. A Dynamic Labeling Scheme Using Vectors. In DEXA, 2007.
9. L. Xu and T. W. Ling and H. Wu and Z. Bao. DDE: from dewey to a fully dynamic XML labeling scheme. In SIGMOD, 2009.
10. L. Xu and T. W. Ling and Z. Bao. and H. Wu. Efficient Label Encoding for Range-based Dynamic XML Labeling Schemes (Extended) www.comp.nus.edu.sg/∼xuliang/encodingextend.pdf
11. Paul F. Dietz. Maintaining order in a linked list. In Annual ACM Symposium on Theory of Computing, 1982.
12. Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In VLDB, 2001.
13. S. Abiteboul and S. Alstrup and H. Kaplan and T. Milo and T. Rauhe. Compact Labeling Scheme for Ancestor Queries. In SIAM J. Comput, 2006.
14. Patrick O'Neil and Elizabeth O'Neil and Shankar Pal and Istvan Cseri and Gideon Schaller and Nigel Westbury. ORDPATHs: Insert-friendly XML Node Labels. In SIGMOD, 2004.