Insert-friendly XML Containment Labeling Scheme

Canwei Zhuang
Department of Computer Science
Xiamen University
Xiamen 361005, China
cwzhuang0229@163.com

Ziyu Lin[+]
Department of Computer Science
Xiamen University
Xiamen 361005, China
ziyulin@xmu.edu.cn

Shaorong Feng
Department of Computer Science
Xiamen University
Xiamen 361005, China
shaorong@xmu.edu.cn

ABSTRACT
The labeling scheme is designed to label the XML nodes so that both ordered and un-ordered queries can be processed without accessing the original XML file. When XML data become dynamic, it is important to design a labeling scheme that can facilitate updates and support query processing efficiently. In this paper, we propose a novel containment labeling scheme called DXCL (Dynamic XML Containment Labeling) to effectively process updating in dynamic XML data. Compared with the existing dynamic labeling schemes, a distinguishing feature of DXCL is that DXCL is compact and efficient regardless of whether the documents are updated or not. DXCL uses fixed length integer numbers to label initial XML documents and hence yields compact label size and high query performance. When updates take place, DXCL also has high performance on both label updates and query processing especially in the case of skewed insertions. Experimental results conform the benefits of our approach over the previous dynamic schemes.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems – Query processing

General Terms
Algorithms, Performance.

Keywords
Dynamic XML Data, Containment Labeling scheme, Update.

1. INTRODUCTION
XML becomes an important standard for data representation and exchange on the web and elsewhere. Labeling schemes have been wildly adopted to process query over XML data which conform to an ordered tree-structured data model. Labeling schemes facilitate XML query processing by assigning a unique label to each node in the XML tree. In such a way, the structural relationships of the nodes such as ancestor/descendant, parent/child can be efficiently established.

Containment labeling scheme [1] is popular in many XML database management systems. It provides several advantages over prefix labeling scheme [2]. The label size of containment scheme is not affected by the structure of the XML documents; whereas the sizes of prefix labels increase linearly with the depths of XML document, which makes prefix labeling scheme performs poorly if XML documents are deep and complex. In addition, when querying XML data, the prefix based scheme needs a prefix comparison for the determination of the structural relationships, which spends more time than ordering operations.

While containment labeling scheme works well for static XML documents, an insertion of a node incurs re-labeling of large amounts of nodes, which is costly and becomes a bottleneck. The existing approach to design dynamic containment labeling scheme is based on the notion of encoding, which includes CDBS [3] and QED [4]. The encoding approaches transform the original containment labels to some dynamic formats which can avoid re-labeling when updates take places. However, the encoding schemes are no entirely satisfactory. Firstly, transforming labels into dynamic formats incurs extra labeling cost and larger label size. In addition, since encoding approaches generate the codes not sequentially, they all require creating encoding table with size O(N) for labeling N nodes. It may fail to process large-scale XML documents when limited memory is available.

The dynamic labeling schemes and traditional static labeling scheme both have advantages and disadvantages: the dynamic labeling schemes are preferred for XML documents that are frequently updated, in which case the performances of static labeling schemes degrade significantly as large amounts of nodes need relabeling; in contrast, when the XML documents are not or rarely updated, the static documents are more efficiently supported by the static labeling schemes as applying dynamic schemes to documents would result in extra encoding cost and querying inefficiency. For getting better performance, we should choose between the static schemes and the dynamic schemes to label the XML document in accordance with its updating frequency. In practice, however, the line between static and dynamic XML documents is often blurred since the updating frequency of a document varies according to time. Hence making a choice between the static schemes and the dynamic schemes is not an easy thing and in many cases it may turn out to be contrary to one's expectations. It is of great interest to design a labeling scheme tailored for both static and dynamic XML documents.

In this paper, we propose a novel dynamic containment labeling scheme called DXCL which doesn't need transform the original labels to dynamic format but can effectively process updating in dynamic XML data. DXCL labels initial XML document based on integer numbers which are stored with fixed bits, and therefore yields cheap label costs as well as compact label size and high query performance. Moreover, when XML becomes dynamic, DXCL completely avoids relabeling and its label quality is resilient to skewed insertions.

2. RELATED WORK
Due to space constraint, we only focus on XML labeling and encoding techniques related to containment labeling scheme.

[+]Corresponding author.

Containment Labeling Scheme In containment labeling scheme [1], every node is assigned three values: *start*, *end* and *level*, where *start* and *end* denote an interval and *level* refers to the level in the XML document tree. For any two nodes *u* and *v*, *u* is an ancestor of *v iff* the interval of *v* is contained in the interval of *u*. Additionally, with using the *level* of a node, the parent-child relationship can be determined efficiently. Document order can also be deduced well by the comparison of *start* values. However, containment labeling scheme can not support updates efficiently. An insertion of a node incurs relabeling of all its ancestor nodes and all the nodes after this node in document order.

Encoding Schemes are proposed to avoid the re-labeling when XML updating. By applying an encoding scheme to containment labeling scheme, the original labels are transformed to some dynamic codes which can efficiently process updates. QED[4] encoding scheme transforms labels to QED codes. Given three integer numbers 1,2, 3 where each number is stored with 2 bit, i.e. 01, 10 and 11, a QED code is a sequence of these numbers which ends with 2 or 3. QED codes are compared based on lexicographical order and robust enough to allow insertions without re-labeling. For example, "22" can be inserted between "2" and "3" whereas "212" can be inserted between "2" and "22". In additional, QED completely avoids the overflow problem as the number 0 does not appear in QED code itself and can be served as the separator of the different codes. However, the sizes of QED codes increase fast for skewed insertions. For example, suppose there are many codes that are required to be inserted one by one before a QED code "332", then each insertion requires that two more bits should be added for the new inserted code, i.e., the new codes will be "3312", "33112", "33112" etc. The fast increase of code lengths make QED perform poorly. The other encoding scheme CDBS [3] is similar to QED except that its encoding unit is binary bit. Compared with QED, CDBS is compact and its labeling cost is small, but CDBS cannot completely solve the re-labeling problem in frequent updates due to its overflow problem.

P-Containment In [3], a variant of the containment labeling scheme called P-Containment is proposed. Rather than storing the *level* information, P-Containment scheme stores the *start* value of the parent of the node. With the parent information, the parent-child relationship can be determined faster and the sibling relationship can be determined much faster. Furthermore, when dynamic encoding schemes are applied, P-Containment can efficiently process the internal node insertions. Prefix labeling schemes, however, cannot intrinsically avoid re-labeling when an insertion takes place between child and parent nodes.

3. DXCL

Our labeling scheme DXCL is based on P-Containment (see Sec.2), and solves the update sensitive problem.

3.1 DXCL code

We first introduce some correlative conceptions on DXCL code.

Definition 1 (Quaternary String, QS) *Given a set of integer numbers A={1,2,3} where each number is stored with 2 bit, i.e. "01", "10" and "11". A quaternary string is ($q_1q_2...q_t$), where t is the code size; $q_t \in \{2, 3\}$and $q_i \in A$, $1 \le i \le t$-1.*

Definition 2 (DXCL Code, DC) *DXCL Code is a integer number N concatenating a quaternary string, i.e. DC=N $\oplus$ QS= ($N.q_1q_2...q_t$), where N is store with fixed bits and t $\geq$ 0.*

Note that: (1).When *t*=0, the DXCL code is just a integer number and therefore we can apply the integer number to label initial XML document;(2).The delimiter "." in a DXCL code *($N.q_1q_2...q_t$)* is not needed to be stored since the integer number *N* is of fixed length; (3). The same as that of QED [4], number 0 (stored with 2 bits, i.e. "00") does not appear in the quaternary string field because it servers as the separator to identify the different DXCL codes. In such a way, DXCL could never encounter the overflow problem.

Example 1 DXCL uses the number 0 (2 bits) to separate different codes. For example, "9.2209.230" will be separated to "9.22" and "9.23". Importantly, it avoids the overflow problem in this way.

DXCL codes are compared by lexicographical order.

Definition 3 (Lexicographical order $\prec$) *Given two DXCL codes S_L:$M.p_1p_2...p_s$, and S_R:$N.q_1q_2...q_t$, $S_L \prec S_R$ if and only if one of the following three conditions holds:*
C1. M<N;
C2. M=N, s<t and p_1=q_1, p_2=q_2,..., p_s=q_s;
C3. M=N and $\exists\, k \le$ min(s, t), such that p_1=q_1; p_2=q_2;...; p_{k-1}=q_{k-1} and p_k<q_k.

Example 2 We have "9" $\prec$ "10" based on condition *C1*. "9" $\prec$ "9.2" since "9" is a prefix of "9.2", which satisfies *C2*. Based on *C3*, "9.233" $\prec$ "9.3" because the comparison is from left to right.

3.2 Initial labeling

DXCL labels the initial XML tree based on integer numbers which are stored with fixed bits. Suppose the total number of the nodes in an XML tree is *K*. Since we should assign *start* and *end* values to total *K* nodes, we need to generate 2*K* consecutive integer numbers, and the size to store each integer is $\log_2 2K$ bits. Fig.1 shows an example of an XML tree labeled by DXCL. Note that each integer in Fig.1 is stored with 4 bits.

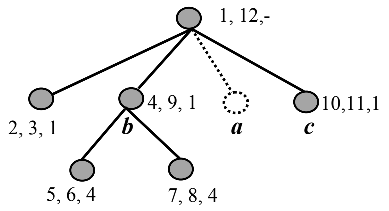


Fig. 1 DXCL scheme ("*a*" is an inserted node)

3.3 Updates processing

We start by introducing two primitive functions to determine a new quaternary string that precedes or follows a given quaternary string *S* in lexicographical order. Suppose a sequence *SEQ* containing all the quaternary strings with size equal to that of *S* is arranged in increasing lexicographical order. We define ***BEF*(*S*)** to get the string before *S* and ***AFT*(*S*)** to get the string after *S* in this sequence. Specially, when *S* is the first string in *SEQ*, i.e. *S*=$1^{size(S)-1} \oplus 2$ (we use "i^k" to represent the *k* "*i*"s in this paper), we define *BEF*(*S*)= $1^{size(S)} \oplus 3^{size(S)}$; when *S* is the last string in *SEQ*, i.e. *S*=$3^{size(S)}$, we define *AFT*(*S*)= $3^{size(S)} \oplus 1^{size(S)-1} \oplus 2$.

Example 3 Let 23 be a quaternary string, then *SEQ* is 12, 13, 22, 23, 32,33. Based on our definition, we have *AFT*(23)=32 and *BEF* (23)=22. Likewise, *AFT*(12)=13, *AFT*(13)=22,...; and *BEF*(33)= 32, *BEF*(32)=23,....Specially, *AFT*(33)=3312 and *BEF* (12)=1133.

It is verifiable that *BEF*(*S*) $\prec S \prec$ *AFT*(*S*) lexicographically. The implements of *AFT* and *BFE* are shown in Algorithm 1 and Algorithm 2.

Algorithm 1 ***AFT*(S)**

```
1  if $S$= "$3^{size(S)}$" then return $3^{size(S)} \oplus 1^{size(S)-1} \oplus 2$;
2  else if the last symbol of $S$ is "2"
3      then return $S$ with the last symbol changed to "3";
4  else if the last symbol of $S$ is "3"
5     denote the position of the lastly encountered "1" or "2" in $S$ as $p$
6     if $substring(S, p, p)$= "1"
7         then return $substring(S, 1, p\text{-}1) \oplus 2 \oplus 1^{size(S)-p-1} \oplus$ "2";
8     else      // the case $substring(S, p, p)$= "2"
9         return $substring(S, 1, p\text{-}1) \oplus 3 \oplus 1^{size(S)-p-1} \oplus$ "2";
```

Algorithm 2 ***BEF*(*S*)**

```
1  if $S$= "$1^{size(S)-1}$" $\oplus$ "2" then return $1^{size(S)} \oplus 3^{size(S)}$;
2  else if the last symbol of $S$ is "3"
3     then return $S$ with the last symbol changed to "2"
4  else if the last symbol of $S$ is "2"
5     $S_T \leftarrow substring(S, 1, size(S)\text{-}1)$;
6     denote the position of the lastly encountered "2" or "3" in $S_T$ as $p$
7     if substring($S, p, p$)= "2"
8        then return $substring(S, 1, p\text{-}1) \oplus 1 \oplus 3^{size(S)-p}$
9     else      // the case $substring(S, p, p)$= "3"
10       return $substring(S, 1, p\text{-}1) \oplus 2 \oplus 3^{size(S)-p}$
```

We then propose the algorithm that can always insert a DXCL code between two ordered DXCL codes, which guarantees that we can update the XML document without re-labeling.

Algorithm 3 ***GetInsertedCode*(S_L, S_R)**

```
Input: DXCL codes $S_L$ and $S_R$
Output: DXCL code $S_M$, such that $S_L \prec S_M \prec S_R$
1  if $size(S_L) = size(S_R)$ then $S_M \leftarrow S_L \oplus$ "2"
2  else if $size(S_L) < size(S_R)$ then
3        $S_T \leftarrow substring(S_R, size(S_L)+1, size(S_R))$
4        $S_M \leftarrow S_L \oplus BEF(S_T)$
5  else  $S_T \leftarrow substring(S_L, size(S_R)+1, size(S_L))$
6        $S_M \leftarrow substring(S_L, 1, size(S_R)) \oplus AFT(S_T)$
                                  //we define $substring(S, 1, 0)=NIL$
7  end
8  return $S_M$
```

Theorem 1 *Given any two DXCL codes S_L and S_R which satisfies $S_L \prec S_R$, we can always find a new DXCL code S_M based on Algorithm 3 such that $S_L \prec S_M \prec S_R$ lexicographically.*

Proof If $size(S_L) = size(S_R)$, then S_M=$S_L \oplus$"2" and $S_L \prec S_M \prec S_R$; If $size(S_L)<size(S_R)$, then $S_L \prec =substring(S_R, 1, size(S_L))$ as $S_L \prec S_R$. Moreover, let S_T=$substring(S_R, size(S_L)+1, size(S_R))$, then S_T is a quaternary string and we can get $BEF(S_T)$ which is smaller than S_T lexicographically. Thus $S_L \prec S_L \oplus BEF(S_T) \prec S_L \oplus S_T \prec =substring(S_R, 1, size(S_L)) \oplus S_T = S_R$. Let $S_M = S_L \oplus BEF(S_T)$, then $S_L \prec S_M \prec S_R$; If $size(S_L)>size(S_R)$, then $substring(S_L, 1, size(S_R)) \prec S_R$ as $S_L \prec S_R$; In additional, let S_T=$substring(S_L, size(S_R)+1, size(S_L))$, then S_T is a quaternary string and we can get $AFT(S_T)$ which is larger than S_T lexicographically. Thus $S_L = substring(S_L, 1, size(S_R)) \oplus S_T \prec substring(S_L, 1, size(S_R)) \oplus AFT(S_T) \prec S_R \oplus AFT(S_T) \prec S_R$. Let S_M= $substring(S_L, 1, size(S_R)) \oplus AFT(S_T)$, then $S_L \prec S_M \prec S_R$. □

Example 4 To insert a code between "9" and "10", since they are of same size, we concatenate one more "2" after "9" to get the inserted code, which is "9.2". To insert a code between "9" and "9.2", since the size of "9" is smaller than that of "9.2", we get S_T = "2" (see line 3 in Algorithm 3) and then concatenate $BEF(S_T)$ = "12" after "9" to get the inserted code, which is "9.12". Similarly, we can get the inserted code "9.3" between "9.2" and "10".

The same as that of QED [4], we require the last symbol of the quaternary string field in a DXCL code to be "2" or "3". We use an example to show the reason.

Example 5 Suppose there are two codes "9.1" and "9.11". We have "9.1" $\prec$ "9.11", but we can not insert anther code S_M such that "9.1" $\prec$ S_M $\prec$ "9.11". Hence we require the quaternary string field in a DXCL code to be ended with "2" or "3".

Based on Algorithm 3 and Theorem 1, DXCL can process XML updating without re-labeling the existing labels.

Example 6 To insert a node "*a*" in Fig. 1, we should insert 2 DXCL codes between the *end* of node *b* "9" and the *start* of the node *c* "10". If we use the original containment scheme, we can not insert a code between "9" and "10" and we must re-label the existing nodes. Based on our *GetInsertedCode* algorithm, we insert a code "9.2" between "9" and "10", and then the *start* value of the inserted node "*a*" is "9.2". The *end* value of node "*a*" is an insertion between the code "9.2" and "10", which is"9.3". We don't need to re-label any node but can keep the labeling scheme working correctly. It is similar for the insertions in other positions.

DXCL has high resilience to skewed insertions. We use an example to analyze the DXCL label size after skewed insertions.

Example 7 Suppose there are 2000 nodes which are required to be inserted repeatedly before the node "*c*" in Fig.1. Then we should generate 4000 DXCL codes one by one between "9" and "10". Based on our algorithm, the first 2 codes are "9.2" and"9.3", which can be stored with 8 bits(equal to 4+2$\times$1+2, where the last 2 is the size of the separator 0);Likewise, the following 2 codes are "9.32" and "9.33" stored with size 10; 6 codes "9.3312", "9.3313", "9.3322", "9.3323", "9.3332" and "9.3333" are stored with size 14; 54 codes "9.33331112",…, "9.33333333" are stored with size 22; and all the rest codes are stored with size 38. It can be seen that the label size of DXCL is little affected by ordered insertion sequence. In contrast, the sizes of CDBS and QED codes increase at 1 or 2 bits per insertion.

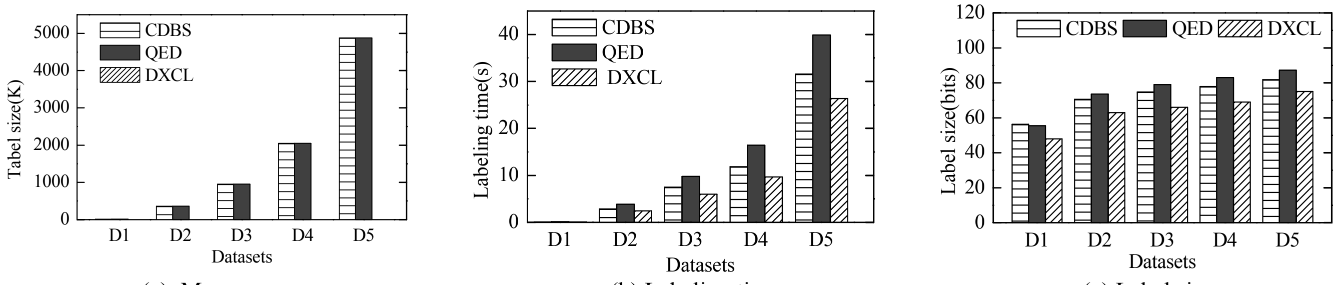
[5] has proved that the label size of any deterministic labeling scheme which does not re-label nodes must increases linearly in the worst case. DXCL cannot escape from this claim also. DXCL needs to assign labels of size $O(n)$ when n nodes are inserted between two consecutive nodes *left* and *right* as follows: the first node *M* is inserted between *left* and *right*; then the second node is inserted between *left* and *M*; the insertion of the i^{th} node($3 \le i \le n$) takes place between the $(i\text{-}2)^{th}$ inserted node and the $(i\text{-}1)^{th}$ inserted node. Clearly, this is normally extremely rare in practice.

4. EXPERIMENT AND RESULTS

We evaluate DXCL against CDBS and QED using P-Containment labels. We don't compare DXCL against Ordpaths(see SIGMOD 2004) and DDE(see SIGMOD 2009) as they are prefix-based schemes. Table 1 shows the test datasets.

Table 1. Test datasets

Dataset	Doc.	No. of nodes	Max/average depth
D1	Hamlet	6,636	6/4.79
D2	Allshakes	179,690	7/5.58
D3	Nasa	476,646	8/3.16
D4	Lineitem	1,022,976	3/2.94
D5	Treebank	2,437,666	36/7.87



(a) Memory usage (b) Labeling time (c) Label size

Fig. 2 Performance study on initial labeling

4.1 Initial Labeling

We test the performances of different schemes labeling on D1-D5.

Memory Usage We compare the memory usage dominated by the size of the encoding table and show the result in Fig.2 (a). We observe clear difference of memory usages between difference schemes. This result conforms to our previous discussions that DXCL need no encoding table whereas both CDBS and QED need a tables of size $O(N)$ where N is the nodes number. DXCL can process large XML documents with limited memory available.

Labeling Time Fig.2 (b) shows the different initial labeling time of different containment schemes. DXCL spends smallest time since it doesn't need to transform labels into dynamic formats as those of CDBS and QED. Additionally, QED needs longest encoding time since QED encodes longer strings and has time-consuming division operation by "3".

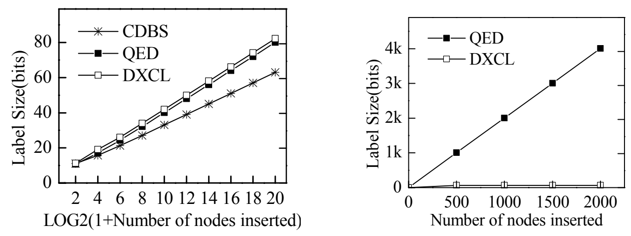
Label Sizes As is shown in Fig.2(c), QED has largest average label size as QED is stored with quaternary string and "0" cannot appear in the QED code itself, which is a waste. Compared with CDBS codes which are stored with variable length, DXCL is more compact.

4.2 Frequent Updates

We discuss the performances of two kinds of frequent updates.

Uniform Insertions We test the cases that 2^n-1(n=1, 2,…,20) nodes are uniformly inserted between two consecutive nodes *left* and *right* where the *end* value of the node *left* and the *start* value of the node *right* both are *NIL*. The 2^n-1 nodes are inserted by n times. At the first time, one node M is inserted between *left* and *right*. At the second time, another 2 new nodes are inserted between every two consecutive nodes which lie between *left* and *right*, i.e., one is inserted between *left* and M and the other between M and *right*. At the k^{th} time (k=1, 2,…, n), 2^k new nodes are inserted similarly. We study the average label size of new inserted nodes after uniform insertions. As is shown in Fig.3 (a), the label size of DXCL shows a little overhead compared with that of QED. However, the difference is negligible. Both DXCL and QED assign labels to updated nodes using quaternary strings and thus their label sizes are approximately the same. Though the label size of CDBS is smallest, it cannot avoid re-labeling in XML frequent updates.

Skewed Insertions are common in practice. CDBS easily encounter overflow problem in the case of skewed insertions as it uses fixed bits to represent its size. Therefore we just compare our DXCL against QED. We test the case that n nodes are inserted one by one before a particular node of which the *start* value is *NIL*. Fig.3 (b) shows the different label sizes after skewed insertions. The size of DXCL labels only increases slightly whereas the size of QED labels has showed a much higher increase. This result illustrates the significant advantage of DXCL.



(a) Uniform insertions (b) Skewed insertions

Fig. 3 Performance study on frequent updates

4.3 Query Performances

Since all the three schemes compare labels based on lexicographical order, the label size is the primary factor in determining the query performances. We ignore the diagrams of the comparison of query performances here as they show the similar trends to that of Fig.2(c), Fig.3 (a) and Fig.3 (b). DXCL has high performance on query processing especially in the case of skewed insertions.

5. CONCLUSION

In this paper, we have presented a novel containment labeling scheme DXCL which not only completely avoids re-labeling when updating, but also has compact size and high query performance. DXCL has controllable size for skewed insertions in which case the existing labeling schemes perform poorly. Experimental results have demonstrated the benefits of our proposed labeling scheme compared to previous approaches.

6. ACKOWLEDGEMENT

This research was supported by the Fundamental Research Funds for the Central Universities under Grant No. 2011121049.

REFERENCES

[1] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In SIGMOD, 2001.

[2] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In SIGMOD, 2002.

[3] C. Li, T. W. Ling, and M. Hu. Efficient updates in dynamic XML data: from binary string to quaternary string. VLDB J, 2008.

[4] C. Li and T. W. Ling. QED: a novel quaternary encoding to completely avoid relabeling in XML updates. In CIKM,2005.

[5] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. In PODS, 2002.