A Concise Labeling Scheme for XML Data

Risi Thonangi

Software Engineering and Technology Labs,
Infosys Technologies Limited,
Bangalore, India.
Risi_Thonangi@infosys.com

Abstract

In this paper, we look at the problem of assigning labels to nodes of a dynamic XML tree such that the labels encode all ancestor-descendant relationships between the nodes and the document-order between the nodes. Such labeling facilitates efficient XML query processing. A number of labeling schemes have been designed for this task. These schemes can be broadly classified into (1) Static Labeling Schemes and (2) Dynamic Labeling Schemes. While static schemes generate short labels, their performance degrades in update intensive environments. Dynamic schemes have nice update performance, but their size of labels is high. A good labeling scheme should generate concise labels and should perform better when there are arbitrary updates on the XML tree. In this paper, we present a new labeling scheme called Sector-based Labeling (SL) scheme which labels nodes with sectors. We analyze the proposed SL scheme and show that it generates smaller labels than the static schemes and has update performance as good as the dynamic schemes. We conduct experiments and show that the obtained results substantiate our analyses.

1 Introduction

The growing popularity of XML standard for communication on the Internet has led to ubiquitous existence of XML data. This triggered the need for systems which can store and query XML data efficiently. An XML document consists of a hierarchy of properly nested tagged elements. Query languages such as XPath [5] and XQuery [4] facilitate searching a document by traversing its hierarchy. Using such languages, a user can define path/tree-pattern queries. The document order in which the XML elements appear plays an important role in query formation.

Older XML query processors stored XML data in its native tree form and searched it by traversing its hierarchy. The tree traversal was either top-down or bottom-up [11]. While traversing the tree, the query processor book-keeps all partial matches that were found. When a new node is read by the search procedure, any old partial matches which convert to complete matches are output to the user. These query processors were time consuming and inefficient as they traverse the complete tree for every query.

To overcome this problem, each node in a tree is assigned a label such that these labels capture ancestor-descendant relationships between the nodes[10, 16]. They should also capture the *document-order*[1] among the nodes. Once the nodes are labeled, they become independent entities and can be stored in a back-end relational database without the pointers. This method allows indexing the nodes on *element-name* and other important fields which improves query processing performance.

In this paper, our topic of interest is labeling the nodes so as to preserve relationships between them. A number of labeling schemes have been proposed until now to solve this problem. They can be broadly classified into two categories.

(1) Static Labeling Schemes
These schemes are mainly designed to label XML documents which are static. Cost of inserting a new element into a document labeled by a static scheme is high but the labels generated by these schemes are concise.

(2) Dynamic Labeling Schemes
Schemes in this category are useful to label update intensive XML documents. The size of the labels generated by these schemes is comparatively high, but they can accommodate new insertions efficiently into the XML tree.

Earlier proposed schemes were mainly static labeling schemes and were typically range-based [10, 7, 3]. The Range-Labeling (RL) [10] scheme assigns each node a range on a one-dimensional axis such that a node's range will include all it's children's ranges. The average label-size generated by this scheme is $2 * \log N$ where N is the number of nodes, but labeling new insertions into the XML tree may frequently require relabeling many nodes.

International Conference on Management of Data
COMAD 2006, Delhi, India, December 14–16, 2006
©*Computer Society of India, 2006*

[1]The order in which elements appear in an XML document. This order is the same as the pre-order of a tree.

Newer labeling schemes like [15, 6] are dynamic schemes which design prefix-based labels that increase in length as the node's depth increases. The update performance of these schemes is much better than static schemes but the labels generated are larger in size. The extra storage space required to store these labels affects the query processing performance as more hard-disk reads would be necessary. The labels might also be inefficient to process if they cannot fit into native machine words.

The main contributions of this paper are summarized as follows:

- we propose a new labeling scheme which generates labels smaller than that of static schemes and has update performance as good as the dynamic schemes. The new labeling scheme called as Sector-based Labeling (SL) scheme allocates sectors in a two-dimensional plane as labels.
- We present algorithms to label a given tree with sector-based labels and to insert nodes into it at run time.
- We derive mathematical formulae to quickly find ancestor-descendant and document-order relationships between two labels.
- We give an in-depth label-size analysis of the proposed SL scheme as well as other existing schemes and show that SL scheme is relatively concise and is efficient at handling updates.
- We conduct experiments on standard datasets and compare label-size conciseness and efficiency at handling updates between the proposed SL scheme and existing schemes.

The rest of the paper is organized as follows. Section 2 discusses related work. In section 3, we present the new Sector-based Labeling scheme. In section 4, we analyze the label size complexity and update performance of the existing labeling schemes. In section 5 we give experimental results. Section 6 concludes the paper and discusses some future avenues for research in labeling schemes.

2 Related work

2.1 Static Labeling Schemes

Dietz labeling scheme [7, 16], an early static-labeling scheme, assigns each node two values:"pre" and "post". The *pre* (*post*) value of a node A is the position of A in the pre-order (post-order) traversal of the tree. Node A is an ancestor of B if-and-only-if

$$A.pre \ < \ A.pre \ \wedge \ A.post \ > \ B.post.$$

However, such a labeling scheme does not allow any insertions into the labeled tree without relabeling a large part of the tree. The XISS System [10] employs a labeling scheme which assigns every node two values as its label: "order" and "size". The two values form a range (order, order+size) and hence this scheme is called as $range$-labeling (RL) scheme. The labels are assigned such that a child's interval is contained in all it's parents' intervals. Two nodes A and B form an ancestor-descendant pair if-and-only-if

$$A.order \ < \ B.pre \ < \ A.order + A.size.$$

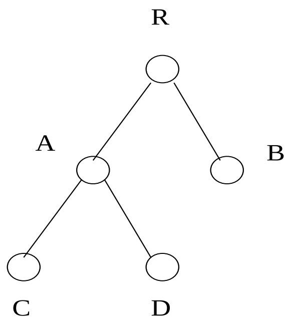
The $size$ of a node determines the number of children it can hold and is inversely proportional to the depth of the node. The $order$ values of the nodes in the tree increase in prefix order. This labeling scheme allows insertions to a node until space in its interval is available. When it runs out of space at a node, a new insertion at it can only be accommodated by "size" of the node. This might require relabeling nodes with pre-order positions larger than the current node. Although Dietz and RL schemes are effective in generating concise labels they cannot handle updates efficiently. It is also difficult, in the case of RL scheme, to determine how much space to allocate to each node. There by, a situation can occur where parts of the XML tree with frequent updates might run out of space while other parts of the tree might be left with free space.

2.2 Dynamic Labeling Schemes

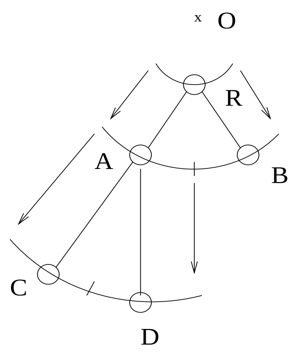
Dynamic labeling schemes like [14, 6, 12] assign a code to each node and the label of a node is the concatenation of all the codes of nodes appearing on its incoming path from the root node. For such labeling schemes, checking for an ancestor-descendant relationship between two nodes is equivalent to determining if one node's label is a prefix of the other. The integer-based prefix labeling scheme [14], called as the Dewey labeling scheme, assigns an integer 'n' (code) to the n'th child of a node. A node's label is the concatenated string of the integers assigned to all it's ancestors. A delimiter is used while concatenation to remove ambiguities. Cohen et al [6] circumvent the problem of delimiters by allocating *prefix-free* binary codes to sibling nodes. They discuss two methods to generate such codes. In section 4, we analyze these two methods.

Wu et al[15] propose a new prefix-based labeling scheme based on the property of prime numbers. In this labeling scheme, each node is assigned a prime-number and the label of a node is formed as the product of all it's ancestors' prime numbers. Checking for an ancestor-descendant relationship between two nodes labeled with the prime number labeling scheme is equivalent to determining if the descendant's label is perfectly divisible by the ancestor's label.

Li et al propose two schemes, QED[8] and CDBS[9], which include novel encoding methods to support code (a quaternary or binary string) insertion into a sequence of existing codes without disturbing the order between them and without requiring to relabel them. An important feature of these approaches is that they compare codes based on the lexicographical order rather than the numerical order which allows a code insertion between two existing codes by increasing the size of the inserted code. For example, a



(a) An example XML tree



(b) XML nodes and their labels/sectors

Figure 1: Assigning sectors as labels to nodes

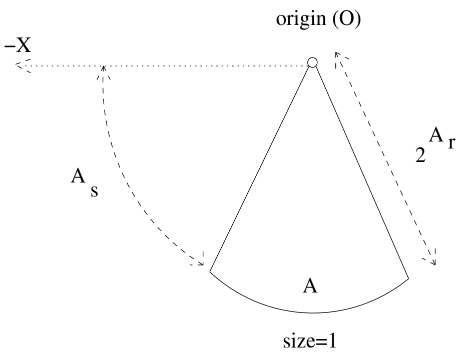


Figure 2: Node 'A' and it's SL label $\langle A_r, A_s \rangle$

new code 1101 can be inserted between two existing codes 11 and 111 without disturbing existing codes. Such an encoding method can be used with static or dynamic labeling schemes.

The above discussed dynamic schemes can handle dynamically changing XML documents easily (see Section 4). However, they are not space efficient. With increase in depth and fan-out of a tree, the size of labels generated by these schemes increase faster than range-based labeling schemes. Apart from such high storage costs, these labels are also inefficient to process on, compared to the range-based labels, especially in cases when they cannot be accommodated into native machine words.

Silberstein et al[13] present two I/O-efficient data structures to maintain labels of large and dynamic XML documents. The two data structures, W-BOX (Weight-balanced B-tree for Ordering XML) and B-BOX (Back-linked B-tree for ordering XML), are B-tree based data structures which organize the labels for efficient updates. The ideas provided here are fairly general and can be incorporated into any labeling scheme.

3 Sector-based Labeling

The Sector-based Labeling (SL) scheme chooses sectors for nodes in such a way that the ancestor-descendant relationship and the document-order between any two nodes can be determined from their labels. In this section, we explain the proposed SL scheme in detail. Section 3.1 proposes an algorithm to efficiently label nodes of an XML tree with sector-based labels and analyzes the complexity of the generated labels. Section 3.2 presents an efficient scheme to handle new insertions.

We use the following notation in the coming sections. Labels are denoted by A, B. Since each node is assigned exactly one unique label, symbols A and B are also used to denote the respective nodes to which these labels were assigned. If a label A has two fields, the first field is denoted by A_1 and the second field is denoted by A_2; A is written as $\langle A_1, A_2 \rangle$. The X,Y co-ordinates of a point P on a two dimensional plane are denoted by $P.x$ and $P.y$ respectively.

3.1 Sector-based Labeling (SL) Scheme

The sectors are allocated to nodes in such a way that the angle formed by a parent's sector at the origin completely encloses that of all its children. Figures 1(a) and 1(b) show an example tree and the sectors assigned to its nodes. See Figure 2 for an example of a node labeled by SL scheme. The horizontal line which originates at the origin and extends towards the negative X direction (labeled as -X) is the reference axis. Angles are measures w.r.t this axis. SL scheme incorporates two optimizations to make the labeling scheme space efficient. The optimizations are given below.

(1) only sectors of unit size and with their center at origin are considered by the labeling scheme. For example in Figure 2 the sector allocated to A is of size 1 and has its center at the origin.

(2) only sectors with radius a power of 2 are considered. In Figure 2, the radius of the sector allocated to A is 2^{A_r}.

With these two optimizations in place, each sector considered for a label can be uniquely identified by two values: (1) the logarithm[2] of its radius, and (2) the offset it makes w.r.t the reference axis. These two values are assigned as a label to the node to which the sector is assigned. For example, in Figure 2, the node A is assigned a label $\langle A_r, A_s \rangle$ where 2^{A_r} is the radius of the sector assigned to A and A_s is the *radial-distance*[3] of the *starting point* of the sector from the reference-axis. Such a representation for sectors ensures that the space required to store a label is considerably smaller as the sector-length is not stored and only the logarithm of the sector's radius is stored. Section 3.1.2, gives a detailed analysis of the size complexity of SL scheme's labels.

The algorithm to label a given XML tree is given in Figure 3. Given a node A, algorithm Label-Tree labels all the descendants of A by traversing the subtree rooted at A in

[2]in this paper, all logarithms are to the base 2.

[3]The radial-distance between two points on a circle is the smallest distance one point has to traverse on the perimeter of the circle to reach the other point.

```
Label-Tree($A$,$\langle A_r, A_s \rangle$)
    // 'A' is the XML tree node with label $\langle A_r, A_s \rangle$
    // This function labels all the descendants of 'A'
    1. Find min $k$ s.t. $2^k > A.num-children$
    2. i = 0
    3. for-each $D$ a child of $A$
    4.     $D_r = A_r + k$
    5.     $D_s = A_s * 2^k + i$
    6.     Label-Tree($D$,$\langle D_r, D_s \rangle$)
    7.     i = i + 1
    8. return
```

Figure 3: Algorithm to label an XML tree with SL labels

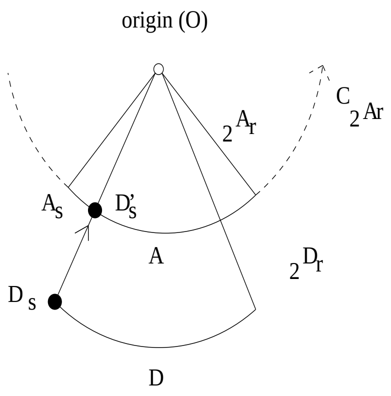


Figure 4: Nodes A and D, where A is an ancestor of D

a depth first fashion. Label-Tree is first invoked with the root node and the root node's label $\langle 0, 0 \rangle$. It initially finds minimum k such that 2^k is atleast the number of children of A (step 1 in Figure 3). Label-Tree ensures that each child of A is assigned a sector that is a part of an *expanded* A's sector (denoted as A' in Figure 5). The radius of A' is 2^k times to that of A and hence it's sector starts at $A_s * 2^k$. Also, the size of sector A' will be 2^k times of A's sector size. As the sector size of A is of unit magnitude, sector size of A' will be 2^k. Steps 2-7 break sector A' into unit length sectors and distribute them to the immediate children of A. Figure 5 shows one descendant D of node A being labeled with the first unit-sector starting from $A_s * 2^k$ on radius 2^{A_r+k}.

3.1.1 Checking for Ancestor-Descendant relationship and Document-Order

The next theorem gives a necessary and sufficient condition the sectors allocated to an ancestor and its descendant will satisfy.

Theorem 1 *Node A is an ancestor of node D,* iff *their allocated labels (sectors), $\langle A_r, A_s \rangle$ and $\langle B_r, B_s \rangle$ respectively, satisfy the following equation:*

$$D_r > A_r \;\wedge\; \lfloor D_s \gg (D_r - A_r) \rfloor = A_s \tag{1}$$

where '$\gg$' stands for the standard right-shift bit-operation.

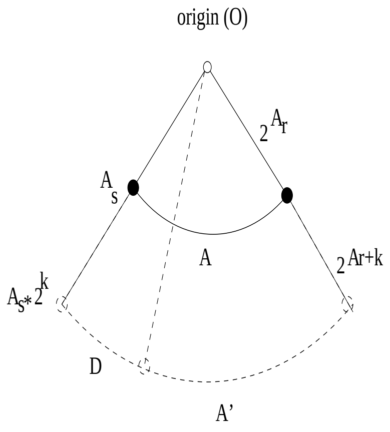


Figure 5: Accommodating A's children

Proof 1 *Let $\mathcal{C}_r$ denote a circle with radius r and center at origin. From figure 4, D will be a descendant of A* iff *the projection of D's sector onto $\mathcal{C}_{2^{A_r}}$ overlaps with A's sector which means the point D'_s should lie on A's sector. The radial distance of D'_s from the reference axis is given by:*

$$D_s * \frac{2^{A_r}}{2^{D_r}}.$$

This projection should lie on A's sector, which means the following should hold true:

$$A_s \;\leq\; D_s * \frac{2^{A_r}}{2^{D_r}} \;<\; A_s + 1$$

$$\Rightarrow \lfloor\, D_s * \frac{2A_r}{2^{D_r}} \,\rfloor = A_s$$

$$\Rightarrow \lfloor\, D_s \gg (D_r - A_r) \,\rfloor = A_s$$

subject to the condition that $D_r > A_r$. Hence, A is an ancestor of D iff

$$D_r > A_r \;\;\wedge\;\; \lfloor D_s \gg (D_r - A_r) \rfloor = A_s$$

Node A is an ancestor of D if $\langle A_r, A_s \rangle$ and $\langle D_r, D_s \rangle$ satisfy Equation 1. Further, A appears before D in the document-order if the projection of D_s on the circle with radius A_r is greater than A_s. When the projection of D_s coincides with A_s, the node with the smaller radius will come first in the document order. The following condition evaluates true when A appears before D:

$$\left(\frac{A_r}{D_r} * D_s > A_s\right) \;\vee\; \left(\frac{A_r}{D_r} * D_s = A_s \;\wedge\; A_r < D_r\right)$$

3.1.2 Label Size Complexity

The size of the label plays an important role because the storage requirement of the labeling scheme has a direct impact on the performance of the XML query processor. The

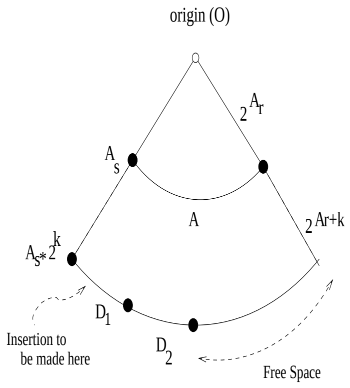


Figure 6: Position at which the new insertion is requested.

number of nodes in the tree is usually huge and hence arbitrarily large labels will attract significant I/O cost at the time of query processing. Further, processing times will be higher for labels which cannot be fit into native machine-words. SL generates the smallest labels when compared to any other labeling scheme and hence improves the query response time. Let f be the fan-out of an XML tree and d be its maximum depth. The below theorem gives the maximum number of bits required to store an SL label.

Theorem 2 *The maximum number of bits an SL generated label would require is*

$$L_{max}(SL) < d*\log f + \log{(d*\log f)}$$

Proof 2 *Since the SL scheme labels every node A in the tree with a unit-length sector, the children of A should be allocated sectors on radius that is f times the radius of A. Radius of A is 2^{A_r}, hence the radius of any of it's children would be $2^{A_r+\log f}$. Thus, if the root node is assigned the label $\langle 0,0\rangle$, a leaf node's radius will be $2^{d*\log f}$. Since we only store the logarithm of the radius, the maximum number of bits required to represent the radius of any node is:*

$$L_{max}(A_r) = \log{(d*\log f)}.$$

The number of leaves in the XML tree are f^d. These leaves are assigned unit-length sectors starting from the reference axis on the circle with radius $2^{d*\log f}$. The last among these leaves is labeled with a sector that starts at f^d-1. Hence:

$$\begin{aligned} L_{max}(A_s) &= \log{(f^d-1)} \\ &< d*\log f \\ L_{max}(SL) &= L_{max}(A_r)+L_{max}(A_s) \\ &< \log{(d*\log f)}+d*\log f \end{aligned}$$

3.2 Handling Insertions

When a new node is inserted the following three cases can occur.

case-1 The position at which the insertion is requested is free. For example in Figure 5, when an insertion is requested at node A's second child (i.e. after D).

case-2 The parent has space, but the position at which the new insertion has to be made is not free. For example, in Figure 6 when an insertion is requested at node A's first child.

case-3 The parent has no space left to accommodate the new child irrespective of the position requested. In Figure 8, node A has 2^k children that have completely occupied the available sector space.

Whenever a new insertion is made, the child node (inserted node) should be allocated a sector such that the ancestor-descendant relationships between the already existing nodes in the XML tree are not disturbed, and new ancestor-descendant relationships are formed between the child node and its ancestor nodes. While inserting a node, we may encounter any of the above three cases. For the first case, the new insertion can be accommodated trivially by labeling the inserted node with the requested sector. For example, in Figure 5 node A has only one descendant D. If an insertion is requested by the user at position-2 among A's children (i.e. after node D), the request can be processed without disturbing any existing labels. The sector at position-2 is available and is allocated to the inserted node.

For the second case, the position at which the insertion is requested is already occupied. In Figure 6, the insertion request is at the position occupied by D_1. To insert the new node, first the subtrees at D_1 and D_2 are moved right by one sector, i.e. D_1 is assigned D_2's sector and D_2 is assigned the sector currently to its right. Such a relocation frees the sector earlier allocated to D_1 which can now be assigned to the new node (see Figure 7). To move a subtree one position to its right, the labels of all the nodes which appear in that subtree have to be changed. Moving the subtrees D_1 and D_2 to their next positions amounts to rotating them by an angle θ in the label space where

$$\theta = \frac{1}{2^{D_{1_r}}}.$$

For every node B that is a descendant of D_1, the radius of it's sector remains unchanged. Hence, if the new label assigned to B is $\langle B_r^{new}, B_s^{new}\rangle$, then $B_r^{new} = B_r^{old}$. Because B's sector is rotated by an angle θ, the offset at which it's new sector starts will be:

$$B_s^{new} = B_s^{old} + 2^{B_r}*\theta = B_s^{old} + 2^{B_r - D_{1_r}}.$$

For the third case, no space is available to accommodate the new insertion irrespective of its requested position and hence the radius of the current children is increased so as to create more sector space. Figure 8 shows a situation where the parent node A has no space to accommodate new insertions. The children $\{D_1, D_2, \ldots, D_{2^k}\}$, which were labeled with sectors of radius 2^{A_r+k} have occupied the available space completely. To make space for the new insertion, they are moved to a larger radius $2^{A_r+k'}$ ($k' > k$) so

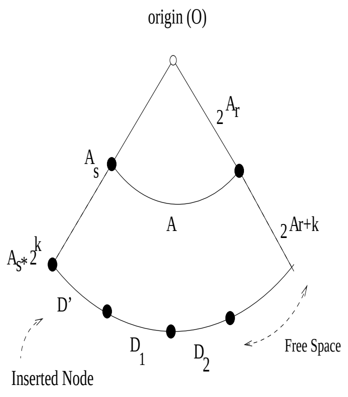


Figure 7: Insertion Accommodated by relocating D_1 and D_2

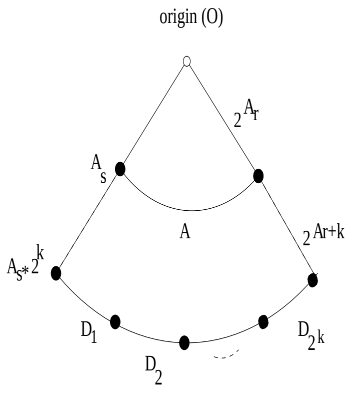


Figure 8: Space not available for the insertion irrespective of its requested position

that extra sector-space can be created to accommodate the insertion (see Figure 3.2). The requested insertion could have been at any position occupied by the children nodes. After relocating all the children to a higher radius, the new node can be inserted as explained in the handling of case-2. Value of k', the exponential increase in the radius of the sectors, can be chosen to be any value greater than k. Notice that after relocation, the parent node A can accommodate $2^{k'-k}$ times more children than before. We choose $k' = k+1$ to ensure that whenever a relocation is triggered, the space available at node A is doubled. In the process of this relocation, the entire subtree of D_i is moved to a new sector D'_i without disturbing the relationships between the nodes. Let the label of this sector be $\langle D_{i_r}^{new}, D_{i_s}^{new} \rangle$. Then:

$$D_{i_r}^{new} = D_{i_r}^{old} + 1 \tag{2}$$

$$\begin{aligned} D_{i_s}^{new} &= A_s * 2^{k'} + (D_{i_s}^{old} - A_s * 2^k) \\ &= D_{i_s}^{old} + A_s * (2^{k+1} - 2^k), \quad \because k' = k+1 \\ &= D_{i_s}^{old} + A_s * 2^k \end{aligned} \tag{3}$$

In the above equation, $A_s * 2^{k'}$ is the new starting point for children of A (see Figure 9) and $(D_{i_s}^{old} - A_s * 2^k)$ computes the position of D_i among its siblings. The new radius of D_i will be 2^{A_r+k+1} which is twice that of it's old radius. But D_{i_r} stores the exponent of D_i's radius. Hence, $D_{i_r}^{new} = D_{i_r}^{old} + 1$. Theorem 3, given below, calculates the new label of any descendant of A.

Theorem 3 *Let B be a descendant of A (the node whose children are to be relocated), and it's label prior to the relocation be $\langle B_r^{old}, B_s^{old} \rangle$. If A's children are moved to sectors with their radius twice larger, then B's label after the relocation can be calculated as:*

$$B_r^{new} = B_r^{old} + 1$$

$$B_s^{new} = B_s^{old} + A_s * 2^{B_r^{old} - A_r}$$

Proof 3 *The set $\{D_1, D_2, \ldots, D_{2^k}\}$ constitutes the child nodes of A, and hence the node B has to be a descendant of one among them. Let that node be D_i. From Equation 3, D_i's new label is $D_{i_r}^{new} = D_{i_r}^{old} + 1$ and $D_{i_s}^{new} = D_{i_s}^{old} + A_s * 2^k$ where $k = D_{i_r}^{old} - A_r$. Since no updates are performed inside the subtree D_i, radii of all descendants of D_i increase by the same amount that D_i has undergone. Hence,*

$$B_r^{new} = B_r^{old} + 1. \tag{4}$$

The descendants of D_i remain in same position relative *to D_i. This means, if $D_{i_s}^{(B_r)}$ denotes the projection of D_{i_s} onto the circle of radius B_r, then the quantity $(D_{i_s}^{(B_r)} - B_s)$ remains unchanged after the relocation of the subtree rooted at D_i. By equaling the new and old values of this quantity:*

$$\frac{2^{B_r^{new}}}{2^{D_{i_r}^{new}}} * D_{i_s}^{new} - B_s^{new} = \frac{2^{B_r^{old}}}{2^{D_{i_r}^{old}}} * D_{i_s}^{old} - B_s^{old}$$

$$2^{B_r^{new} - D_{i_r}^{new}} * D_{i_s}^{new} - B_s^{new} = 2^{B_r^{old} - D_{i_r}^{old}} * D_{i_s}^{old} - B_s^{old}$$

But $B_r^{new} - D_{i_r}^{new} = B_r^{old} - D_{i_r}^{old}$ as both B_r and D_{i_r} have increased by 1 (see Equations 2 and 4). Substituting this result, and also for $D_{i_s}^{new}$ calculated from Equation 3

$$\begin{aligned} B_s^{new} &= 2^{B_r^{old} - D_{i_r}^{old}} * (D_{i_s}^{old} + A_s * 2^k) \\ &\quad + B_s^{old} - 2^{B_r^{old} - D_{i_r}^{old}} * D_{i_s}^{old} \\ &= B_s^{old} + 2^{B_r^{old} - D_{i_r}^{old}} * A_s * 2^k \\ &= B_s^{old} + 2^{B_r^{old} - A_r - k} * A_s * 2^k, \\ &\qquad \because D_{i_r}^{old} = A_r + k \\ &= B_s^{old} + 2^{B_r^{old} - A_r} * A_s \end{aligned}$$

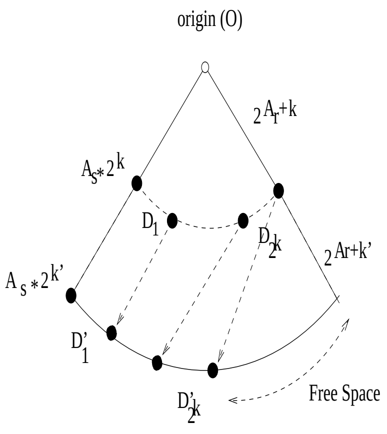


Figure 9: Space created by relocating all children to a larger radius

3.2.1 Cost of Relocation while Updates

As explained at the beginning of this section, three cases arise while inserting a new node into an XML tree. For case-1, no relocation cost is incurred as the requested position for the new node is empty. For case-2, the position is not available and hence sibling subtrees of the inserted node to its right is relabeled (see Figure 10). For case-3, no space is available for the new node irrespective of its requested position. Hence all the sibling subtrees of the inserted node are relabeled (see Figure 11). Note that whenever a subtree is relabeled, the space created at its root is doubled. This ensures that nodes which actively undergo updates are allocated more space than compared to other nodes. This happens dynamically, and allows each node to find the right sector space through the updating environment. As part of section 4, we analyze the update costs of various labeling schemes and compare them against SL scheme.

4 Complexities of other Labeling Schemes

In the previous section (Section 3), we analyzed the label size complexity of SL scheme and the cost of relocation associated with relabeling the XML tree when updates are made. In this section, we analyze and compare the complexities of labeling schemes on two dimensions: (1) Conciseness of the generated labels, and (2) cost of updates. As in Section 3, f and d are defined as the maximum fan-out and maximum depth of an XML tree respectively. N will denote the total number of nodes in the tree which can be calculated as:

$$N \;=\; \sum_{k=0}^{d} f^k \tag{5}$$

4.1 Range-based Labeling Scheme

Range-based labeling (RL) scheme assigns each node two numbers that denote the start and end points of an interval. The maximum value that these numbers can take is the number of nodes in the XML tree. But from Equation 5, the number of nodes N is greater than f^d. Hence, the number of bits required to store one field of a range-based label is greater than $d * \log f$. Therefore:

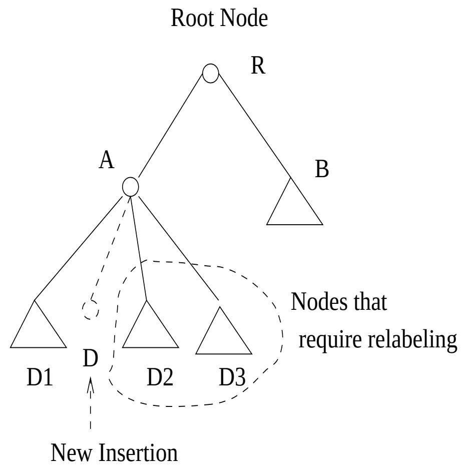


Figure 10: Siblings occurring after the insertion need to be relabeled.

$$L_{max}(RL) \;>\; 2 * d * \log f \tag{6}$$

From Theorem 2, the maximum number of bits an SL generated label can take is:

$$L_{max}(SL) \;<\; d * \log f + \log{(d * \log f)}$$

Notice that $L_{max}(RL)$ is approximately twice of $d * \log f$, while $L_{max}(SL)$ has the term appearing only once. But it contains another term $\log{(d * \log f)}$, which is only logarithmic to that of the earlier term.

Whenever a new node is inserted into the tree, RL scheme relabels all the ancestor nodes of the new node as well as nodes which have a higher pre-order position than the inserted node (see Figure 12). However, the proposed SL scheme requires relabeling only a subset of the descendants of the inserted node's parent; as shown in Figure 10 and occasionally as shown in Figure 11. Notice that the nodes relabeled by SL scheme are a subset of the ones relabeled by RL scheme. Hence, SL not only generates smaller labels than that of RL but also has better update performance.

4.2 Prefix-based Labeling Scheme

The prefix-based labeling schemes assign each node a code and the label of the node is formed by combining it's parent's label and it's code. Kaplan et al[6] propose a labeling scheme which assigns the code "$1^{i-1}0$" to the i'th child of a node. The node's label is formed by concatenating it's binary-string to it's parent's label. For example, if a node's code is "10" and it's parent's label is "0110" then the node's label would be "$0110 \cdot 10 \;=\; 011010$". Notice that this assignment ensures that siblings are always assigned *prefix-free* binary-strings. For this reason, no two

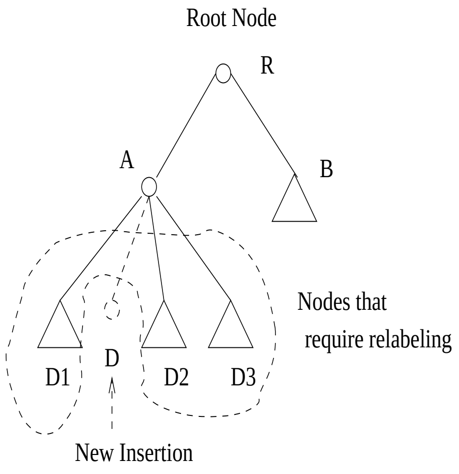


Figure 11: All the children of the ancestor A need to be relabeled.

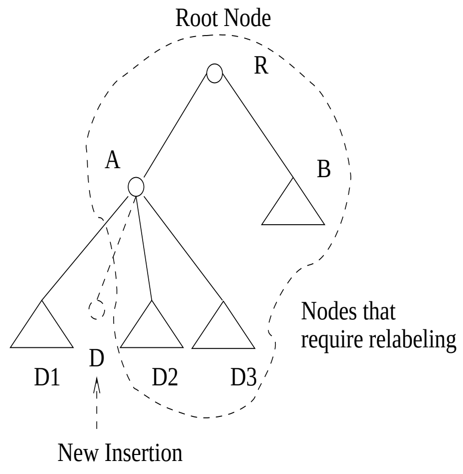


Figure 12: All the nodes with pre-order position greater than the inserted node need to be relabeled.

nodes in a tree are assigned the same label. But the problem with such a labeling scheme is, label-size increases linearly as f increases. Hence, an improved labeling is suggested in which the children of a node are assigned the codes

$$0,\ 10,\ 1100,\ 1101,\ 1110,\ 11110000,\ 11110001, \ldots$$

Namely, the $(i{+}1)$'st string is obtained by incrementing the binary representation of the i'th string and if the string contains all 1's, the string size is doubled by adding 0's at its end. Notice that all the strings generated in such a fashion remain *prefix-free*. The maximum size of the i'th child in the second method is $4 \cdot \log{(i)}$ [6], a marked improvement. We consider only the second form of labeling children in the remaining part of our discussion and refer to the corresponding labeling scheme by *Binary-Prefix Labeling scheme* (BPL). The maximum size for a label in BPL is $(4 \cdot d \cdot \log f)$[6] while an SL scheme's label can take utmost $d * \log f + \log{(d * \log f)}$.

When updates are made on the tree, prefix-based labeling schemes require relabeling parts of the tree when the position at which the insertion is requested is not available (see Figure 10). This is because, prefix-based labeling schemes assign children of a node consecutive binary strings from an infinite prefix-free sequence. Since the assignment is done from the beginning of this sequence, order is maintained among the children.

4.3 Prime Number based Labeling Scheme

Wu et al[15] proposes a prime number based labeling scheme (referred to as PNL scheme from here on) which assigns each node a prime-number and forms the label of a node by multiplying it's prime-number with it's parent's label. For example, if a node's prime-number is 11 and it's parent's label is 6 then the node's label will be 66. We know that, from the characteristics of prime numbers, the k'th prime-number is approximately $k * \log{(k)}$ [15]. Hence, the number of bits required to represent the k'th prime is approximately $k * \log{(k)}$; [15] shows that the error of such an assumption is negligible. Keeping to this assumption and from the condition that $N > f^d$, the number of bits required to store the N'th prime will be greater than $d * \log f$+$\log{(d * \log f)}$[4]. Also, the label assigned to a node by this scheme is not just the code of the node but the product of all the codes of its ancestors. Hence, the maximum size of a PNL scheme generated label could be much larger than that of an SL scheme generated label.

PNL scheme uses the concept of *Chinese Remainder Theorem* to determine the document order using a single value called Simultaneous Congruence (SC). A node's exact document-order position can be calculated as *SC* mod *node's prime* (see [15] for more details). With such a formulation, only a new SC value has to be computed to reflect the change in positions of the nodes whenever a new node is inserted. However, computing this new value requires accessing all the labels of the tree which is computationally heavy.

4.4 QED and CDBS based Labeling Schemes

The primary advantage with QED or CDBS based encoding methods is that they allow arbitrary code insertions into a sequence of existing codes without requiring any relabeling. They order the set of binary-strings (integers) using a lexicographical approach instead of a numerical approach. Approaches like RL scheme or ORDPATHs, which use a numerically ordered domain of integers to build labels, can use these encoding methods to generate components of their labels and benefit from their ordering method at the time of new insertions by not requiring to relabel any of the existing labels. Li et al [9] show that CDBS generates smaller codes than QED and requires the same number of bits as that of integers to represent a given number of codes. Hence, any existing scheme used together

[4]Note that the $d * \log f$+$\log{(d * \log f)}$ bound on size of the N'th prime is a very weak bound considering that the condition $N > f^d$ from Equation 5 is very weak

with the CDBS encoding method will not see its maximum label-size decrease. Thus the SL scheme will remain comparatively label-efficient than the CDBS based schemes.

When a new node is inserted into an already labeled tree, CDBS based labeling schemes do not need to relabel any of the existing nodes. However, SL scheme may require relabeling the descendants of the node at which the new insertion is made. Hence, CDBS based schemes are more update efficient than the SL scheme. But, the labels generated by these schemes for new insertions are much larger since the size of a newly inserted code between two already existing codes is larger than any of them by atleast 1 bit. In the case of SL scheme, whenever a relabeling operation is performed at a node, the increase in the label size will be by 1 bit but the label space at that node doubles. Hence, though CDBS based labeling schemes do not require to perform the relabeling operation dynamic labels generated by them increase in size at a much faster rate.

5 Experimental Results

We implemented in C++, the four labeling schemes: range-based labeling scheme, prefix-based labeling scheme, prime-number labeling scheme and the proposed sector-based labeling scheme. Five XML datasets chosen from the Niagara XML dataset repository [2] were labeled using these schemes. Table 1 shows the characteristics of these datasets. Column 3 in the table gives the number of XML files in the dataset. All the XML files in a dataset are concatenated to form one large file corresponding to the dataset and experiments are conducted on this file. The GNOME XML parser libxml [1] is used to parse the given XML datasets. All the experiments were carried out on an Intel Celeron 2.3GHz machine with 256MB RAM.

Dataset	Name	Files	Nodes	Leaves
D1	NASA	1475	774,602	348,724
D2	Shakespeare	37	327,132	147,448
D3	Sigmod Record	990	98,914	35,633
D4	Movies	490	42,779	16,745
D5	Club	12	5,002	2,073

Table 1: **Characteristics of the Datasets**

For the PNL scheme, [15] mentions two optimizations: (1) use small prime numbers to label the root node and it's children, and (2) allocate $\{2^1, 2^2, 2^3 \ldots, 2^i\}$ as codes to leaves. Further, 2^i is allocated as a code to a leaf only when no primes less than this number are already assigned to other nodes. We included both these optimizations in our implementation of the PNL scheme.

5.1 Label-Size Analysis

Figure 13 shows the results of our experimental study on the space requirements of the four labeling schemes. In all the five datasets, SL performs better than every other labeling scheme. The SL scheme consistently generated labels 25% smaller than that of the RL scheme (the most concise labeling scheme available before) and 45% smaller than that of the BPL scheme. As expected, the average label size of the RL scheme is lesser than the other dynamic labeling schemes because of its tight-labeling of the tree nodes.

5.2 Update-Performance Analysis

While it is important that a labeling scheme generate concise labels, it also should have the ability to efficiently accommodate new insertions onto an already labeled tree. To test the update performance of the labeling schemes we use two experiments that were first given in [13].

Concentrated-Insertions Experiment Initially a two-level XML tree with 2000 elements is chosen and a new two-level subtree with 1000 elements is inserted at the root-node of the base tree amongst the middle of it's children. The insertion is done one element at a time. Specifically, the subtree's root is inserted first at the root node of the base document. Then it's first and last children are inserted, next the second and it's second-to-last children, and so on. The new nodes are constantly inserted at the middle of a parent's children, the most probable position of an insertion. We refer to this experiment as *concentrated-insertions experiment*. The number of nodes updated by the four labeling schemes in the presence of such a sequence of insertions is given in Table-2 [5].

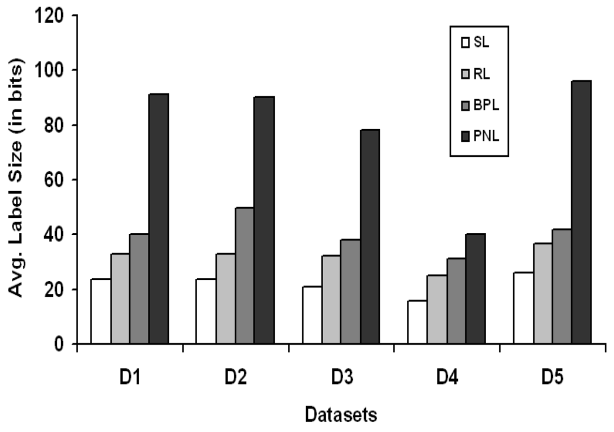


Figure 13: Average label size for each of the labeling schemes

The number of relabellings required by the labeling schemes are divided by 100 for better readability. In this table, see that the RL scheme requires the maximum number of relabellings. This is because, a new node can be accommodated in a RL scheme only by relabeling all the old nodes occurring after the new node in the document order. However, BPL relabels only descendants of the inserted node's right-siblings. Hence, it's update performance in

[5] Results are provided in a table instead of a graph plot to accommodate different ranges of numbers of different labeling schemes.

this experiment is comparatively better than the other labeling schemes. Although SL's update requests are handled in a similar fashion as that of BPL, sometimes it has to relabel descendants of the inserted node's left-siblings as well. This happens when there is no sector space available to accommodate the new insertion, and new space can only be created by increasing the radius of the siblings. But since the sector space doubles after every increase in the radius, such a necessity decreases with the increase in insertions. This is evident from the Table 2 as difference between SL and BPL decreases with the increase in the number of insertions. Note that though PNL's performance was not as good as BPL and SL initially, with the increase in the number of updates it's performance has come very close to BPL. This is because PNL requires relabeling one fifth of the nodes relabeled by RL. But SL and BPL require relabeling a portion of the parent's descendants which have increased due to the concentrated insertions.

Insertions	RL/100	PNL/100	BPL/100	SL/100
100	1786	202	24	25
200	8507	417	98	100
300	25153	642	222	227
400	56725	877	396	401
500	108221	1122	620	625
600	184642	1376	894	904
700	290988	1641	1218	1228
800	432260	1916	1592	1602
900	613456	2201	2016	2026
1000	839577	2496	2490	2500

Table 2: **Number of nodes relabeled by each labeling scheme in a *Concentrated-Insertions Experiment***

Scattered-Insertions Experiment In this experiment, the insertions are scattered over the tree instead of being concentrated at one node. The base-tree is the same as that explained in the first experiment. The maximum number of insertions in this experiment are 10000 instead of 1000 as done in the first experiment. This was done to decrease the effect of scattering the insertions which gave too few insertions per node. Further, the insertions take place only on the non-leaf nodes as insertions on leaves is a no cost operation for SL and BPL. The results of this experiment are given in Table 3. As before, the number of relabellings required by the labeling schemes are divided by 100 for better readability. Notice that the labeling schemes BPL and SL require same relabellings initially which changes as the number of updates increase. This is because, more nodes would come closer to the first powers of 2 with increase in insertions. SL's relabellings remain far lesser than that of PNL and RL.

6 Conclusion

In this paper we proposed a new approach to label nodes in an XML tree such that the labels are concise and allow new insertions to the tree gracefully. We gave a theoretical analysis of the proposed SL scheme as well as other existing schemes and show that our labeling scheme generates labels comparatively smaller than existing labeling schemes. Our experiments on a variety of datasets confirm our theoretical analyses.

Insertions	RL/100	PNL/100	BPL/100	SL/100
1000	990957	198191	555	555
2000	1992089	398417	1140	1140
3000	2978442	595688	1746	1748
4000	3972529	794505	2382	2393
5000	4976614	995322	3044	3070
6000	5988452	1197690	3737	3792
7000	7000148	1400029	4456	4563
8000	8008439	1601687	5199	5368
9000	9006590	1801318	5955	6221
10000	9987864	1997572	6731	7100

Table 3: **Number of nodes relabeled by each labeling scheme in a *Scattered-Insertions Experiment***

References

[1] Libxml: The XML C parser and toolkit of GNOME. http://www.xmlsoft.org/, 2005.

[2] Niagara XML dataset repository, www.cs.wisc.edu/niagara/.

[3] T. Amagasa, M. Yoshikawa, and S. Uemura. Qrs: A robust numbering scheme for xml documents(poster). In *International Conference on Data Engineering*, 2003.

[4] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. XQuery 1.0: An XML query language, W3C working draft. 2001.

[5] J. Clarke and S. DeRose. XML path language (XPath) version 1.0, W3C recommendation. 1999.

[6] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling dynamic XML trees. In *Symposium on Principles of Database Systems*, pages 271–281, 2002.

[7] Paul F. Dietz. Maintaining order in a linked list. In *In proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 122–127, 1982.

[8] Changqing Li and Tok Wang Ling. QED: A novel quaternary encoding to completely avoid re-labeling in xml updates. In *Conference on Information and Knowledge Management (CIKM)*, pages 501–508, 2005.

[9] Changqing Li, Tok Wang Ling, and Min Hu. Efficient processing of updates in dynamic xml data. In *International Conference on Data Engineering*, 2006.

[10] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *The VLDB Journal*, pages 361–370, 2001.

[11] Jason McHugh and Jennifer Widom. Query optimization for XML. In *The VLDB Journal*, pages 315–326, 1999.

[12] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: insert-friendly XML node-labels. In *ACM SIGMOD International Conference on Management of Data*, pages 903–908, 2004.

[13] Adam Silberstein, Hao He, Ke Yi, and Jun Yang. BOXes: Efficient maintenance of order-based labeling for dynamic XML data. In *International Conference on Data Engineering*, pages 285–296, 2005.

[14] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *ACM SIGMOD Conference on Management of Data*, pages 204–215, 2002.

[15] Xiaodong Wu, Mong Li Lee, and Wynne Hsu. A prime number labeling scheme for dynamic ordered XML trees. In *International Conference on Data Engineering*, 2004.

[16] M. Yoshikawa and T. Amagasa. XRel: A path-based approach to storage and retrieval of xml documents using relational databases. In *ACM Transactions on Internet Technologies*, 2001.