CIT. Journal of Computing and Information Technology, Vol. 26, No. 2, June 2018, 99–114
doi: 10.20532/cit.2018.1003955

CBSL – A Compressed Binary String Labeling Scheme for Dynamic Update of XML Documents

Dhanalekshmi Gopinathan and Krishna Asawa

Department of Computer Science, Jaypee Institute of Information Technology, Noida, Uttar Pradesh, India

The growing volume of XML documents over the Web has increased the need for an efficient mechanism to facilitate query processing. Many labeling schemes have been introduced to optimize data retrieval and query processing on XML database documents. As it is known, labels hold information about XML tree nodes such as their position, their relationship with other nodes and their order, which helps in query processing. Most of the existing labeling schemes support query processing over static XML documents. However, they need to re-label during dynamic update. In this paper a compressed binary string labeling scheme is proposed which supports dynamic update of XML documents without re-labeling existing nodes. Analytical results show that the proposed labeling scheme takes less label size in comparison with other labeling schemes. Also, an experiment has been conducted to evaluate the label generation time as well as update processing.

ACM CCS (2012) Classification: Information systems → World Wide Web → Web data description languages → Markup languages → Extensible Markup Language (XML)
Information systems → Data management systems → Query languages → XML query languages

Keywords: compressed binary string, dynamic XML update, label size, XML tree, label generation time, update performance, lexicographic order

1. Introduction

The expressive and extensible nature of XML has become a key feature for its successful adoption as an interchange format between different applications and Web services over the Internet. Moreover, the growing volume of XML documents over the Web, in turn, increases the need for an efficient technique to store and accelerate query processing. XML data comprises nested collections of elements enclosed between start and end tags which describe the semantics of the element.

In general, XML documents are modeled as an ordered tree structure. Hence, the main challenge in XML data management is to provide a storage structure that preserves the tree structure intact. There are various labeling schemes proposed in the literature to provide persistent storage for XML documents by keeping the tree structure. Many of the labeling schemes can efficiently process different queries if the XML documents are static. To include structural modifications of XML data without affecting existing labels is still a hot topic.

In this paper we present a new labeling scheme which follows a lexicographic order of strings. The key contributions of this paper are:

(i) It supports dynamic updating of XML documents. This scheme need not re-label any existing nodes and need not re-calculate any values when inserting an order-sensitive node into the XML tree.

(ii) An analytical study is conducted to show that this labeling scheme requires less symbols to label each node in the XML tree.

The rest of the paper is organized as follows. In Section 2 a review of related work is provided. In Section 3 the proposed approach is elabo-

rated in detail. Section 4 discusses the formal algorithm used in this scheme. In Section 5 an analytical study for the computation of label size is elucidated. Finally, experiment, results and conclusion are discussed in Sections 6 and 7, respectively.

2. Related Work

Existing labeling schemes are mainly classified into two major categories known as interval based (range based) labeling scheme [1], [2] and prefix based labeling scheme [3], [4], [5]. *Range based* labeling scheme generates labels by assigning two values (begin, end) which denote the start and end positions of the element in the document. These schemes can efficiently determine ancestor-descendant relationships. Additionally, the level information is required to find the parent-child relationship between the nodes. However, the major limitation of these labeling schemes is that they cannot determine the sibling relationship by only looking at the labels. To determine the sibling relationship between two nodes, first it is required to search the parent of one node; further, it has to be checked whether the other node is the child of this same parent. This is obviously a time-consuming and expensive process. The second limitation is that they do not support dynamic update. During update, it may be necessary to re-label all the existing or some of the existing labels, which is an expensive operation. This partially limitation is latter resolved by keeping a larger interval size. However, doing this may cause a lot of values unused, which in turn increases the storage size. Moreover, if the insertion operation exceeds the interval size, then a re-labelling may require assigning new values to the inserted nodes and already existing labels.

In the *prefix based* labeling scheme, each node is labeled by concatenating the prefix of the parent label with its self-label. This labeling scheme efficiently determines all the structural relationships between the nodes by merely looking at the labels. In [4] the authors proposed DeweyID which is an integer based prefix scheme. Each node is labeled with an integer concatenated with its parent's label. It is a static labeling scheme which requires re-labeling when new nodes are inserted.

An extension of this approach is proposed in [6] to support the dynamic updating of documents without relabeling. In [3], Cohen proposed the binary strings labeling scheme characterized by labeling the root with an empty string. The children nodes in the first level are labeled as 0, 10, 110 and so on. For any node u, the children are labeled as $L(u).0$, $L(u).10$, $L(u).110$ and so on, where $L(u)$ is label of node u. In [7], O'Neil proposed the ORDPATH labeling scheme based on Dewey order using odd numbers for initial labeling; this scheme reserves even and negative numbers for later insertions. However, if the size of the reserved code overflows, it has to re-label existing nodes. In [8], Duong and Zhang proposed the LSDX labeling scheme where each label is a combination of letters and digits, and the label of the root node is assigned to 0a, where the integer 0 represents the level or depth of the node while the alphabet represents the self-label of the node. Even though LSDX is designed to meet the dynamic nature of XML data, it is not a persistent labeling scheme. There are situations where collisions can occur during updation. Ko and Lee [9] proposed IBSL (Improved Binary String Labeling). Each label in this scheme uses binary bit strings, while the scheme avoids re-labeling during updations. However, space overhead and label size are not efficient. In Dynamic XDAS [10], the labels are generated based on masking technique; the authors use modified approach form [9] to incorporate dynamic updating of XML data.

3. Proposed Approach

The proposed Compressed Binary String Labeling (CBSL) is inspired by the improved prefix based labeling scheme [9]. It uses string encoding to compress the number of symbols in the binary string to label the XML data. The advantage of using the compressed binary string is that it uses less symbols to label each node in the XML tree. It uses lexicographic order to compare its labels. The next section explains the compressed binary string encoding used by the proposed approach.

3.1. Compressed Binary String Encoding

The proposed Compressed Binary String Labeling uses binary strings generated as 10, 110,

1110, and so forth. This labeling compression is defined in Definition 1.

Definition 1. (Compressed Binary String Label). The first binary string, $B_S(1) = 10$, is encoded as "10", which means that only one "1" is followed by a "0". The second binary string, $B_S(2) = 110$ is encoded as "20", meaning that two "1"s are followed by a "0". Further, the third binary string, $B_S(3) = 1110$ is encoded as "30", meaning that three "1" are followed by a zero. Finally, for any k, $B_S(k) = 1^k0$ means that k "1"s are followed by a "0".

Table 1 shows the number of labels with its size. Here the size refers to the number of symbols used to label each node in the XML document. Thus, $10^1 - 10^0$ denotes $(10 - 1) = 9$ labels, $10^2 - 10^1 = (100 - 10) = 90$ labels and so on.

Table 1. Number of labels with its size regarding the number of symbols.

Number of labels	Size (#symbols)
$10^1 - 10^0$	2
$10^2 - 10^1$	3
$10^3 - 10^2$	4
…	…
$10^k - 10^{k-1}$	$(k + 1)$

Definition 2. (Lexicographical order <). If two consecutive binary strings N_{left} and N_{right} are equal ($N_{\text{left}} = N_{\text{right}}$), all bits in N_{left} and N_{right} should be exactly the same. N_{left} is lexicographically smaller than N_{right} ($N_{\text{left}} < N_{\text{right}}$), if one of the following conditions hold:

a) $N_{\text{left}}[k] = 0$ and $N_{\text{right}}[k] = 1$ during bit-by-bit comparison of binary strings from left to right, at any position, say k or,

b) N_{right} has N_{left} as its prefix.

For example, consider two binary strings "110" and "1110": "110" is lexicographically smaller than "1110" as per condition a) in Definition 2. For two binary strings "10" and "100", string "10" is lexicographically smaller than string "100" as per condition b) in Definition 2.

The next section explains the proposed labeling scheme in detail. The example tree shown in Figure 1 is used to illustrate the labeling of each node using the CBSL label.

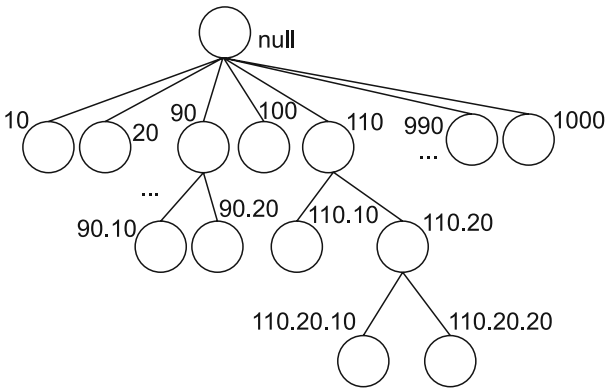


Figure 1. The component resource consumptions.

3.2. Initial Labeling of CBSL

The initial labeling of CBSL starts with labeling the root as NULL. The first level of children is labeled as "10", "20", "30", …, "90" and so forth. According to this approach, the first nine children are labeled with two symbols such as "10", "20", "30", and so forth; the next ninety children are labeled "100", "110", "990" with three symbols and so on.

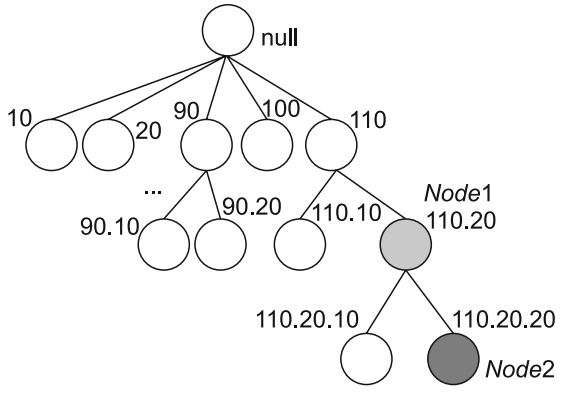
Initial labeling in this scheme follows the lexicographic order, hence it is a unique label. The structural relationships such as parent-child (P-C), ancestor-descendant (A-D) and sibling relationships are computed by using the rules discussed below:

(i) Parent-child relationship (P-C): If the label of *Node*1 is equal to the prefix label of *Node*2, then *Node*1 is the parent of *Node*2.

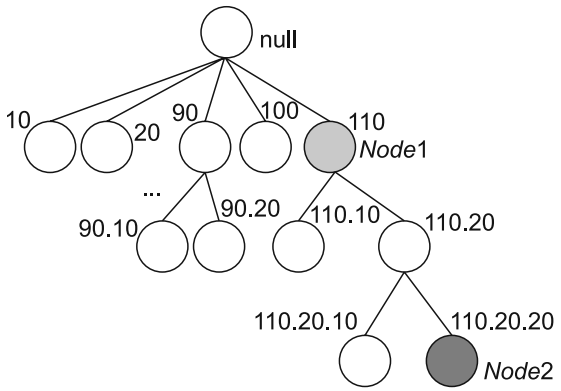
(ii) Ancestor-descendant relationship (A-D): If the label of *Node*1 is a prefix string of the prefix label of *Node*2, then *Node*1 is an ancestor of *Node*2.

(iii) Sibling relationship: If the prefix of *Node*1 and the prefix of *Node*2 are the same, then *Node*1 and *Node*2 are sibling nodes.

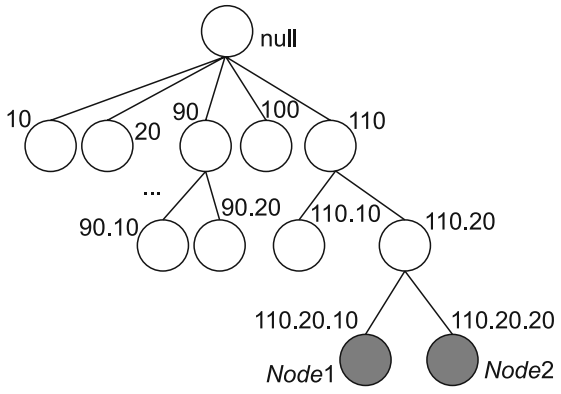
These structural relationships play an important role in query processing. The next section elaborates how CBSL handles the dynamic updates.



(a)



(b)



(c)

Figure 2. (a), *(b)* and *(c)* present the relationships described in rules (i), (ii) and (iii).

3.3. Dynamic Insertions using CBSL

CBSL supports updating XML data dynamically without collision. Inserting a node for various conditions is being elaborated in the form of cases as given below.

Case 1: Inserting a node before the leftmost child:

If the leftmost child has a label with distinctive character "#" (hash), the new node will be assigned a label by changing the last bit "1" in the left label to "01". For instance, if a new node is inserted before the leftmost child "100.10#01", the self-label of the leftmost child is "#01", therefore the new node will get the label "100.10#001", as the last bit of the left node label is changed to 01 as shown in Figure 3.

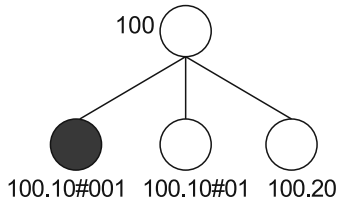


Figure 3. Inserting a node before the leftmost child (Case 1): label contains #.

If the leftmost child doesn't contain the distinctive character "#", then the new node will get a label by concatenating a string "#01" with the leftmost child's label, as depicted in Figure 4.

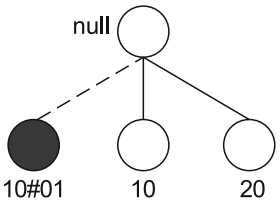


Figure 4. Inserting a node before the leftmost child (Case 1): label contains no #.

Case 2: Inserting a node after the rightmost child:

If the rightmost child contains a distinctive character "#" in its label, the new node will be assigned a label by concatenating a bit "1" to the right label. For instance, a new node inserted after the rightmost child "20#11" will get a label "20#111", as depicted in Figure 5.

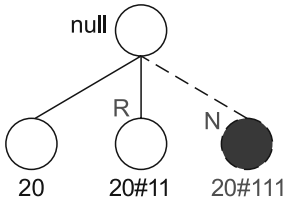


Figure 5. Inserting a node after the rightmost child (Case 2): label contains #.

Next, if the rightmost child doesn't contain distinctive character "#", then the new node will get a label by concatenating a string "#11" with the rightmost child's label, as depicted in Figure 6.

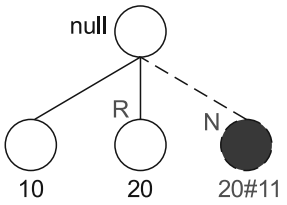


Figure 6. Inserting a node after the rightmost child (Case 2): label contains no #.

Case 3: Inserting a node between two adjacent siblings:

The two conditions considered here are:

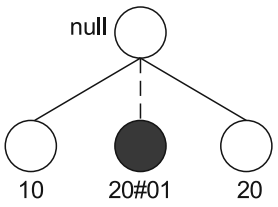
(i) the length of the left child's label is less than or equal to the length of the right child's label, and

(ii) the length of the left child's label is greater than the length of the right child's label.

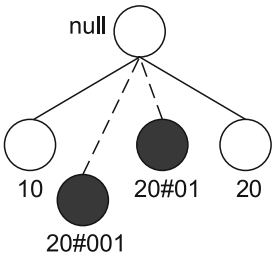
In the first condition, the new node will get a label by suffixing the string "#01" to the right child's label if the distinctive character (#) is not already present in the right node's label, as shown in Figure 7. Furthermore, if a distinctive character is present, the new node will get the label by changing the last bit to "01", as shown in Figure 8.

In the second condition, where the length of the left node's label is greater than the length of the right node's one, the new node will get a label by suffixing the string "1" to the left node's label, as depicted in Figure 8.

In the following, the formal labeling algorithm is elaborated in detail.



(a)



(b)

Figure 7. Inserting a node between two nodes: condition (i).

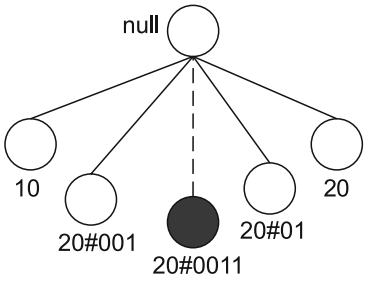


Figure 8. Inserting a node between two nodes: condition (ii).

3.4. Labeling Algorithm

Algorithm 1 elaborates the initial labeling as part of the proposed labeling scheme, while Algorithm 2 defines how dynamic update are handled by the scheme.

3.4.1. Initial Labeling

Algorithm 1 gives the details of CBSL labels for each node in the XML tree initially. A depth-first traversal is performed to assign a label to each node. The algorithm implements Definition 1 of the binary encoding. Here "." is used as the delimiter between the prefix and the self-label part of a label.

Algorithm 1. Initial labeling.

```
Input: XML Document D
Output: Label for each node n in D
begin
  if (n is root) then
    Label (n) = null
  else
    prefix (n) = Label (P)
  end
  if (n is firstChild (P)) then
    selflabel (n) = 10
  else
    templabel = countOnes (nextBstring())
    Label (n) = templabel + 0
  end
    Label (n) = prefix (n) + delimiter + selflabel (n)
end
```

The method nextBstring() generates the next binary string for the node visited, while the method countOnes() counts the number of "1"s in the generated binary string. Dynamic insertions in the proposed scheme are performed as discussed in Subsection 3.3. Dynamic update follow the lexicographic order.

Lemma 1. Inserting a node N_{new} between the two lexicographically ordered strings N_{left} and N_{right} using the CBSL approach follows the lexicographic order, *i.e*. $N_{left} < N_{new} < N_{right}$.

Proof. The above statement is proved by considering two cases. The first one is when the length of the left sibling node is less than or equal to the right sibling node, while the second is when the length of the left sibling node is greater than the right sibling node.

Case a: If length (N_{left}) ≤ length (N_{right}), the N_{new} is assigned a label as shown in steps (b1) of Algorithm 2 which checks two cases:

(i) both labels without a distinctive character "#", and

(ii) labels with a distinctive character "#".

In the proposed labeling, according to Definition 1, the valueOf ((length (N_{left}) − 1) denotes the number of "1"s in the binary string representation of label N_{left}, while valueOf ((length (N_{right}) − 1) indicates the number of "1"s in the label N_{right}. Note that, as per the initial labeling, valueOf ((length (N_{right}) − 1) cannot be less than valueOf ((length (N_{left}) − 1).

(a1) Labels without distinctive character "#" and length (N_{left}) <= length (N_{right}).
Here valueOf (length (N_{left}) − 1) < valueOf (length (N_{right}) − 1) means that the number of 1's in N_{right} is larger than N_{left}, *i.e*. N_{left} is the prefix of N_{right}. Then, as per condition b in Definition 3.2, N_{right} is lexicographically larger than N_{left}. Thus, $N_{left} < N_{new}$, because N_{new} is obtained by concatenating "#01" to N_{right}.

(a2) Length (N_{left}) < length (N_{right}) with distinctive character "#".
In this case, N_{new} is assigned a value by changing the last bit "1" of N_{right} to "01". Since 0 < 1 lexicographically, then, as per the condition a in Definition 3.2, $N_{new} < N_{right}$.

Based on (a1) and (a2), $N_{left} < N_{new} < N_{right}$ lexicographically when length (N_{left}) ≤ length (N_{right}).

Next, to prove the second part, $N_{new} < N_{right}$, we must also consider two cases.

Case b: If length (N_{left}) > length (N_{right}), N_{new} is assigned a label as shown in step (b2) of Algorithm 2., *i.e*., $N_{new} = N_{left}$ + "1".

(b1) From Algorithm 2, N_{new} is assigned a label by concatenating a "1" to N_{left}, thus making N_{left} a prefix of N_{new}. Now, as per condition b in Definition 2, $N_{left} < N_{new}$ lexicographically.

(b2) Since length (N_{left}) > length (N_{right}) and N_{left} is lexicographically smaller than N_{right}, it must satisfy the condition in Definition 2. It means that there exists some position, say k, where N_{left} [k] is "0" and N_{right} [k] is "1". Note that "0" is lexicographically smaller than "1". Therefore, concatenating "1" to N_{left} for assigning a value for N_{new} still makes it smaller than N_{right} lexicographically, *i.e*. $N_{new} < N_{right}$.

Based on (b1) and (b2), $N_{left} < N_{new} < N_{right}$ when length (N_{left}) > length(N_{right}). Therefore, from case (a) and case (b), $N_{left} < N_{new} < N_{right}$.

4. Label Size Analysis

In this section, the term *size* denotes the number of symbols present in a label. As the first part of the size analysis, for a given fan-out (F) value, the maximum number of symbols required to label each string is computed.

Algorithm 2. Insertion (N_{left} , N_{right}).

Input: N_{left}, selflabel of the left sibling, and N_{right}, selflabel of the right sibling
Output: N_{new}, selflabel of the New node
//Case a: Inserting a node before N_{left}
begin
(a1) N_{left} without special character "#"
$N_{new} = 01\# + N_{left}$
(a2) N_{left} with special character "#"
$N_{new} = 0 + N_{left}$
end
//Case b: Inserting a node between two adjacent siblings
begin
(b1) **if** ($lenght(N_{left}) <= lenght(N_{right})$) **then**
if ($!(N_{left}.contains("\#")$ *and* $!(N_{right}.contains("\#"))$ **then**
$N_{new} = N_{left} + \#01$
else
if N_{right} .contains("#") **then**
$N_{new} = N_{right}$ with last bit 1 changed to 01
end
end
end
(b2) **if** ($lenght(N_{left}) > lenght(N_{right})$) **then**
$N_{new} = N_{left} + 1$
end
end
//Case c: Inserting a node after right sibling N_{right}
begin
(c1) N_{right} with special character ("#")
$N_{new} = N_{right} + 1$
(c2) N_{right} without special character ("#")
$N_{new} = N_{right} + 11$
end

Lemma 2. To self-label sibling nodes, for each label string in CBSL, will take at most $\log_{10}(F) + 1$ character symbols.

Proof. In this approach,

(i) 2-symbol character strings can represent the first nine label values, *i.e*. $(10^1 - 10^0)$ values,

(ii) 3-symbol characters string can represent 90 label values, *i.e*. $(10^2 - 10^1)$ values,

(iii) 4-symbol character strings can represent 900 label values, *i.e*. $(10^3 - 10^2)$.

Thus for any k, to represent $(10^k - 10^{k-1})$ labels requires $k + 1$ symbols.

Let

$$F = \left(10^1 - 10^0\right) + \left(10^2 - 10^1\right) + \ldots + \left(10^k - 10^{k-1}\right) = \sum_{i=1}^{k} \left(10^i - 10^{i-1}\right).$$

By mathematical induction,

$$F = \sum_{i=1}^{k} \left(10^i - 10^{i-1}\right) = (k) \cdot 10^k.$$

Hence, the maximum number of character symbols for each label string $K = \log_{10}(F) + 1$.

Example: for $F = 80$, $F = 90$ and $F = 8379$, the symbols required to self-label are

(i) $F = 90$, $K = \log_{10}(90) + 1 = 1.97 + 1 = 3$ symbols.

(ii) $F = 800$, $K = \log_{10}(800) + 1 = 2.90 + 1 = 4$ symbols.

(iii) $F = 8375$, $K = \log_{10}(8375) + 1 = 3.92 + 1 = 5$ symbols.

Next, the computation of total sibling size required for fan-out F nodes is

$$\sum_{i=0,x=2}^{l-2} \left(9 \cdot 10^i\right) \cdot x + \left[F - \sum_{i=0}^{l-2} \left(9 \cdot 10^i\right) \right] \cdot (l+1)$$

Lemma 3. The total sibling self-label size for a fan-out of F nodes in the CBSL will be

$$\sum_{i=0,x=2}^{l-2}\left(9\cdot 10^{i}\right)\cdot x+\left[F-\sum_{i=0}^{l-2}\left(9\cdot 10^{i}\right)\right]\cdot(l+1)$$

symbols. Here, l denotes the length (F).

Proof. The CBSL scheme starts its labeling with the root node as null, hence for the first level of nodes the prefix part will be null. The children at the first level will have only the self-label. Hence, the label starts with 2-character strings such as 10, 20, 30 and so on.

The first step is to compute the size requirement of sibling labels up to $l-2$.

(i) The size requirement of sibling labels is up to $l-2$, where l is the string length (F).

For this, as per the CBSL scheme, labeling falls within the range $(10^1 - 10^0)$, $(10^2 - 10^1)$, $(10^3 - 10^2)$ with a 2-character string, 3-character string, 4-character string and so on.

Hence,

a) $i = 0$ denotes $(10^1 - 10^0) \cdot 10^i = 9 \cdot 10^0 = 9$ sibling labels.

b) $i = 1$ denotes $(10^1 - 10^0) \cdot 10^i = 9 \cdot 10^1 = 90$ sibling labels.

c) $i = 2$ denotes $(10^1 - 10^0) \cdot 10^i = 9 \cdot 10^2 = 900$ sibling labels, and so on.

Now, x denotes the number of symbols used to label these nodes. As per this scheme, "x" starts with 2 and is incremented by 1 for every "i" value up to $l-2$.

Hence, to label sibling nodes, we need $\sum_{i=0,x=2}^{l-2}\left(9\cdot 10^{i}\right)\cdot x$ symbols.

(ii) The size requirement of sibling labels falls in the range $l-1$.

Some cases may have the number of nodes less than $(10^{l-1} - 10^{l-2})$. In such case, to capture the exact number of remaining nodes, the number of nodes which has already been labeled is subtracted from the fan-out value F. Hence, the remaining number of nodes is computed as $F - \sum_{i=0}^{l-2}\left(9\cdot 10^{i}\right)\cdot x$, and the symbols required to label these nodes are obtained by multiplying $(l+1)$, where l denotes the string length of fan-out F.

Therefore, from cases (i) and (ii) it is shown that the total sibling label size is obtained by using formula

$$\sum_{i=0,x=2}^{l-2}\left(9\cdot 10^{i}\right)\cdot x+\left[F-\sum_{i=0}^{l-2}\left(9\cdot 10^{i}\right)\right]\cdot(l+1)$$

Consider an example for fan-out value $F = 8875$. Here, the string length (F) = 4. The first term of the summation series mentioned in Lemma 3 computes the number of symbols required to label the nodes as

a) $i = 0$, $9 \cdot 10^i = 9 \cdot 10^0 = 9$ label values. Symbols required are $9 \cdot x = 9 \cdot 2 = 18$.

b) $i = 1$, $9 \cdot 10^i = 9 \cdot 10^1 = 90$ label values. Symbols required are $90 \cdot x = 90 \cdot 3 = 270$.

c) $i = 2$, $9 \cdot 10^i = 9 \cdot 10^2 = 900$ label values. Symbols required are $900 \cdot x = 900 \cdot 4 = 3600$.

Now, when $i = 3$, which exceeds $(l-2) = 4$, the second term of the formula is used to get the exact remaining number of nodes as $8875 - (9 + 90 + 900) = 7876$ label values. Hence, $7876 * 5 = 39380$ will give the total number of symbols required to label these nodes. Therefore, the total number of symbols required to label a sibling size of 8875 is computed as $(18 + 270 + 3600 + 39380) = 43268$ symbols. Besides, from Property 2 below, the average size of a single label will be $43268/8875 = 4.87$.

Property 1. From Lemma 3, the total sibling label size for a fan-out F is computed as

$$\sum_{i=0,x=2}^{l-2}\left(9\cdot 10^{i}\right)\cdot x+\left[F-\sum_{i=0}^{l-2}\left(9\cdot 10^{i}\right)\right]\cdot(l+1).$$

Hence, storage requirement for CBSL will be

$$\left(\sum_{i=0,x=2}^{l-2}\left(9\cdot 10^{i}\right)\cdot x+\left[F-\sum_{i=0}^{l-2}\left(9\cdot 10^{i}\right)\right]\cdot(l+1)\right)\cdot 8$$

bits. The 8 bits denote the required storage space for 1 symbol.

Property 2. From Lemma 3, the total sibling self-label size for a fan-out F in CBSL is

$$\sum_{i=0,x=2}^{l-2}\left(9\cdot 10^{i}\right)\cdot x+\left[F-\sum_{i=0}^{l-2}\left(9\cdot 10^{i}\right)\right]\cdot(l+1).$$

Hence the average size for a single self-label is

$$\frac{1}{F}\cdot\left(\sum_{i=0,x=2}^{l-2}\left(9\cdot 10^{i}\right)\cdot x+\left[F-\sum_{i=0}^{l-2}\left(9\cdot 10^{i}\right)\right]\cdot(l+1)\right).$$

Property 3. Considering the prefix, the maximum size required to store the complete label (prefix + self-label) is

$$D\cdot\left(\sum_{i=0,x=2}^{l-2}\left(9\cdot 10^{i}\right)\cdot x+\left[F-\sum_{i=0}^{l-2}\left(9\cdot 10^{i}\right)\right]\cdot(l+1)\right),$$

where D denotes the maximum depth of the XML tree.

Property 4. The maximum size required by CBSL for all the nodes in the XML tree with N nodes is

$$N\cdot D\cdot\left(\sum_{i=0,x=2}^{l-2}\left(9\cdot 10^{i}\right)\cdot x+\left[F-\sum_{i=0}^{l-2}\left(9\cdot 10^{i}\right)\right]\cdot(l+1)\right).$$

5. Performance Analysis of Label Size Requirements

In this section, a performance analysis study is conducted to analyze the size required by different labeling schemes such as Dewey, binary, IBSL, binary string and our CBSL scheme. "N", "F" and "D" indicate the number of nodes, maximum fan-out, and max-depth of nodes in an XML document respectively. In conducting this study we consider the benchmark datasets [11], [12] whose characteristics are shown in Table 2.

In Dewey ID [4], all the self-labels use a maximum size of $\log_2(F)$, while the maximum size required to store the complete label including prefix and self-label is "$D \cdot \log_2(F)$". Furthermore, the maximum size required to store all the nodes in the XML tree is computed $N \cdot D \cdot \log_2(F)$. The binary string scheme [3] takes label sizes of 1, 2, 3, ..., F for the first sibling, second sibling and so on. Hence, the total sibling size taken in this scheme is $1 + 2 + ... + F$ which is equal to $F^2/2 + F/2$. And, the average size of the single sibling label is $F/2 + 1/2$. Therefore, the maximum size required to store all the nodes in the XML tree is $N \cdot D \cdot (F/2 + 1/2)$.

In IBSL [13] the total sibling size for the self-label is computed as $F\cdot\log_2(F-1)+2\cdot F-\log(F-1)+1$, and the average size for a single self-label is $\log(F-1)+2-(\log(F-1))/F+1/F$. Therefore, the maximum size required to store all the nodes in the XML tree is $N \cdot D \cdot [\log(F-1)+2-(\log(F-1)/F)+1/F]$. In the IBSL_2010 [9] the total sibling label, the average size for a single self-label, and the maximum size required to store all nodes in an XML tree are computed in the same way in Binary [3]. Table 3 shows the label size analysis for different schemes.

A comparative study of label size computation in various labeling schemes mentioned in Table 3 is performed and the results obtained are depicted in Table 4. It summarizes comparative results obtained on the total sibling size, average size for a single label and maximum size required to store all nodes in an XML tree by different schemes on datasets shown in Table 3. From the table it is clear that our proposed

Table 2. Characteristics of data sets used.

DataSet	Topic	MaxFan-out(F)	MaxDepth(D)	#Nodes
D1	SigmodRecord	6	3	41
D2	NASA	26	6	4834
D3	Shakespeare Play	48	5	6636
D4	Club	13	3	340
D5	Actor	26	4	1110
D6	Department	25	3	2636
D7	Xmark	25500	12	16663000
D8	DBLP	328858	6	3332130
D9	Treebank	56384	36	2437666

Table 2. Characteristics of the data set.

Labeling Scheme	**Total Sibling Label Size**	**Average size for a single self-label**	**Maximal Size required to store all nodes in an XML tree**
Dewey		$\log_2 F$	$N \cdot \mathrm{D} \cdot \log_2 F$
Binary	$F^2 + F/2$	$F/2 + 1/2$	$N \cdot D \cdot (F/2 + 1/2)$
IBSL – Li, Ling (2005)	$F \cdot \log_2(\mathrm{F}-1) + 2 \cdot F - \log(F-1) + 1$	$\log(F-1) + 2 - (\log(F-1))/F + 1/F$	$N \cdot D \cdot [\log(F-1) + 2 - (\log(F-1)/F) + 1/F]$
IBSL – Ko, Lee (2010)	$F^2/2$	$\mathrm{F}/2 + 1/2$	$N \cdot D \cdot (F/2 + 1/2)$
CBSL (proposed scheme)	$\sum_{i=0,x=2}^{l-2}\left(9\cdot 10^i\right)\cdot x+\left[F-\sum_{i=0}^{l-2}\left(9\cdot 10^i\right)\cdot(l+1)\right]$	$\frac{\sum_{i=0,x=2}^{l-2}\left(9\cdot 10^i\right)\cdot x+\left[F-\sum_{i=0,x=2}^{l-2}\left(9\cdot 10^i\right)\cdot(l+1)\right]}{F}$	$\frac{N\cdot D\cdot\sum_{i=0,x=2}^{l-2}\left(9\cdot 10^i\right)\cdot x+\left[F-\sum_{i=0,x=2}^{l-2}\left(9\cdot 10^i\right)\cdot(l+1)\right]}{F}$

Table 4. Label size analysis of different labeling schemes.

	Total sibling label size				Average size for a single self label					Maximal size required to store all nodes in XML tree				
Data set	**Binary**	**IBLS – Li, Ling (2005)**	**IBLS – Ko, Lee (2010)**	**CBSL**	**DeweyID**	**Binary**	**IBLS – Li, Ling (2005)**	**IBLS – Ko, Lee (2010)**	**CBLS**	**DeweyID**	**Binary**	**IBLS – Li, Ling (2005)**	**IBLS – Ko, Lee (2010)**	**CBLS**
D1	21	24.61	21	12	3.5	3.5	4.1	3.5	2	317.95	403.5	317.95	430.5	246
D2	351	169.1	351	108	13.5	13.5	6.5	13.5	4.15	136332	391554	136331.6	391554	120366.6
D3	1176	358.07	1176	144	24.5	24.5	7.46	24.5	3	185309	812910	185309.1	812910	99540
D4	91	70.2	91	39	7	7	5.39	7	3	3774.45	7140	3774.45	7140	3060
D5	351	169.1	351	78	13.5	13.5	6.5	13.5	3	20870	59940	20869.95	59940	13320
D6	325	161.04	325	75	13	13	6.44	13	3	36723.6	102804	36723.61	102804	23724
D7	325137750	424259	325137750	136800	12750.5	12750.5	16.64	12750.5	5.36	2.9E+09	2.55E+12	2.93E+09	2.5E+12	1.07E+09
D8	5.4074E+10	6684712	5.4074E+10	2050006	164429.5	164430	20.33	164430	6.23	3.7E+08	3.29E+12	3.66E+08	3.3E+12	1.25E+08
D9	1.59E+09	1002660	1589605920	322104	28192.5	28192.5	17.78	28192.5	5.71	1.4E+09	2.47E+12	1.39E+09	2.5E+12	5.01E+08

scheme CBSL takes less number of symbols in comparison with the others.

Figure 9 shows the total sibling size taken by CBSL, compared with binary and IBSL schemes. CBSL takes less number of symbols to label XML data sets in comparison with other schemes. For example, the dataset (D3) with a maximum fan-out 48 took 1176 symbols by the Binary [3] and IBSL [9] and IBSL [13] took 354 symbols, whereas CBSL took only 144 symbols to label the same dataset.

The percentage of improvement of CBSL on total sibling label size over binary, and IBSL is shown in Table 5. It can be seen that CBSL achieves a maximum improvement, *i.e.* 100%, for XMark, DBLP, and Treebank while a minimum improvement, 43%, for SigmodRecord when compared to the binary labeling scheme.

Table 5. Improvement of CBSL on total sibling label size.

Dataset	Binary	IBSL – Li, Ling (2005)	IBSL – Ko, Lee (2010)
SigmodRecord	0.43	0.51	0.43
NASA	0.97	0.93	0.97
Shakespeare Play	0.99	0.97	0.99
Club	0.87	0.83	0.87
Actor	0.97	0.93	0.97
Department	0.96	0.93	0.96
XMark	1	1	1
DBLP	1	1	1
Treebank	1	1	1
Average	**0.91**	**0.9**	**0.91**

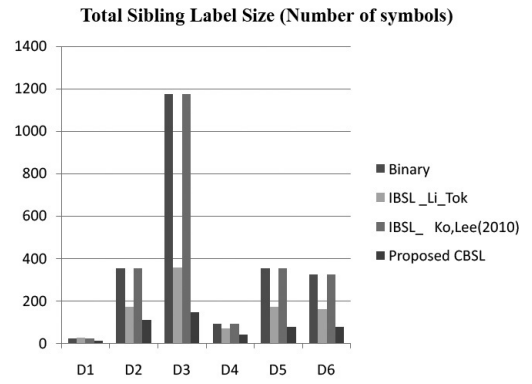


Figure 9. Total sibling label size for the datasets.

Besides, CBSL achieves the overall average improvement of 91% over binary labeling scheme.

The average size of a single label is shown in Table 6 while the improvement percentage on the average size of the single label is shown in Table 7.

The percentage of improvement of CBSL on average size for a single label over Dewey ID, Binary, and ISBSL schemes is shown in Table 8, where it can be seen that the proposed scheme outperforms all the other ones.

For IBSL [13], the minimum improvement of CBSL achieved for SigmodRecord is 0.51, while the maximum improvement is 0.9 for XMark data set.

Next, the maximum label size required to label all nodes in the XML tree for the different datasets is computed and compared with the existing schemes Binary, IBSL – Li, Ling (2005), IBSL – Ko, Lee (2010) and CBSL. The percentage of improvement on maximum label size of CBSL over other schemes on different datasets is shown in Table 8. CBSL achieves a minimum improvement of 23% on maximal label size compared to Dewey ID, and IBSL for the SigmodRecord. Besides, it achieves a maximum improvement of 100% on XMark, DBLP, and Treebank datasets. Additionally, the overall average improvement of the proposed CBSL over other schemes is 90%.

6. Experiments and Results

Two sets of experiments are conducted to evaluate the performance of the proposed labelling scheme. The first set evaluates the time taken to generate the labels, while the second evaluates the update performance of CBLS.

6.1. Label Generation Time

This section explains the time taken to label each node in the XML document by the proposed labeling scheme CBSL. The results are compared with the existing IBSL – Li, Ling (2005), IBSL – Ko, Lee (2010), Dewey ID for the datasets DBLP, XMark, Mondial, and Auction. It is observed that CBSL needs less time to generate the label in comparison with IBSL

Table 6. Average size for a single label.

DataSet	Dewey ID	Binary	IBSL – Li, Ling (2005)	IBSL – Ko, Lee (2010)	Proposed CBSL
D1	3.5	3.5	4.1	3.5	2
D2	13.5	13.5	6.5	13.5	4.15
D3	24.5	24.5	7.46	24.5	3
D4	7	7	5.39	7	3
D5	13.5	13.5	6.5	13.5	3
D6	13	13	6.44	13	3
D7	12750.5	12750.5	16.64	12750.5	5.36
D8	164429.5	164429.5	20.33	164429.5	6.23
D9	28192.5	28192.5	17.78	28192.5	5.71

Table 7. Improvement of CBSL on average label size of single label.

DataSet	Dewey	Binary	IBSL – Li, Ling (2005)	IBSL – Ko, Lee (2010)
SigmodRecord	0.43	0.43	0.51	0.43
NASA	0.85	0.85	0.69	0.85
Shakespeare Play	0.92	0.92	0.73	0.92
Club	0.71	0.71	0.63	0.71
Actor	0.85	0.85	0.69	0.85
Department	0.85	0.85	0.69	0.85
Xmark	1	1	0.88	1
DBLP	1	1	0.9	1
Treebank	1	1	0.89	1

Table 8. Improvement of CBSL on maximum label size.

DataSet	Dewey	Binary	IBSL – Li, Ling (2005)	IBSL – Ko, Lee (2010)
SigmodRecord	0.23	0.43	0.23	0.43
NASA	1	1	1	1
Shakespeare Play	1	1	1	1
Club	0.93	0.97	0.93	0.97
Actor	0.99	1	0.99	1
Department	0.99	1	0.99	1
Xmark	1	1	1	1
DBLP	1	1	1	1
Treebank	1	1	1	1

and Dewey, and provides an 88% improvement over IBSL on labeling the DBLP dataset, 63% improvement on the XMark dataset, and 24% improvement on the Mondial dataset. For the Auction dataset almost all the schemes give approximately the same result.

6.2. Performance Evaluation of Update Processing

Update performance is evaluated using the dataset Shakespeare play Hamlet.xml. As part of the evaluation, the time needed to insert

new nodes and the number of nodes required to re-label are measured. As Hamlet has five acts, the evaluation is tested in four cases such as inserting a node before the act [1], inserting a node between act [2] and act [3], inserting a node between act [4] and act [5] and inserting a node after act [5]. The result is compared with different labeling schemes such as Dewey, Binary, IBSL and XDAS. Figure 12 shows that out of 6636 total nodes of Hamlet file, Dewey and Binary re-label around 6595 nodes, and IBSL, XDAS and the proposed CBSL need not re-label any of the nodes.

Finally, the time required to perform the insertion operation is measured and the result depicted in Figure 13. From the result, it is clear that IBSL, XDAS, and the proposed CBSL take almost the same time for this operation, whereas Dewey ID and Binary took more time for all the cases considered.

7. Conclusion

In this paper a new labeling scheme called CBLS is proposed, which is based on the compressed representation of binary string. This scheme supports dynamic updating of XML

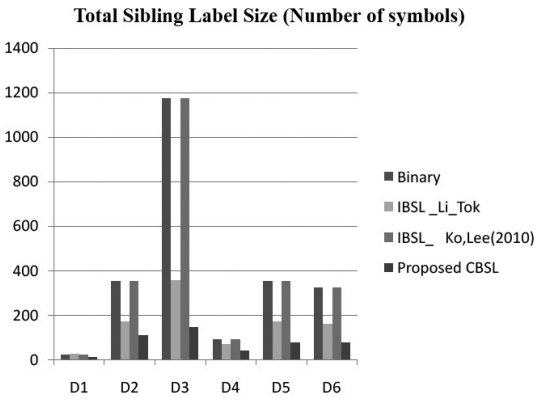


Figure 9. Total sibling label size for the datasets.

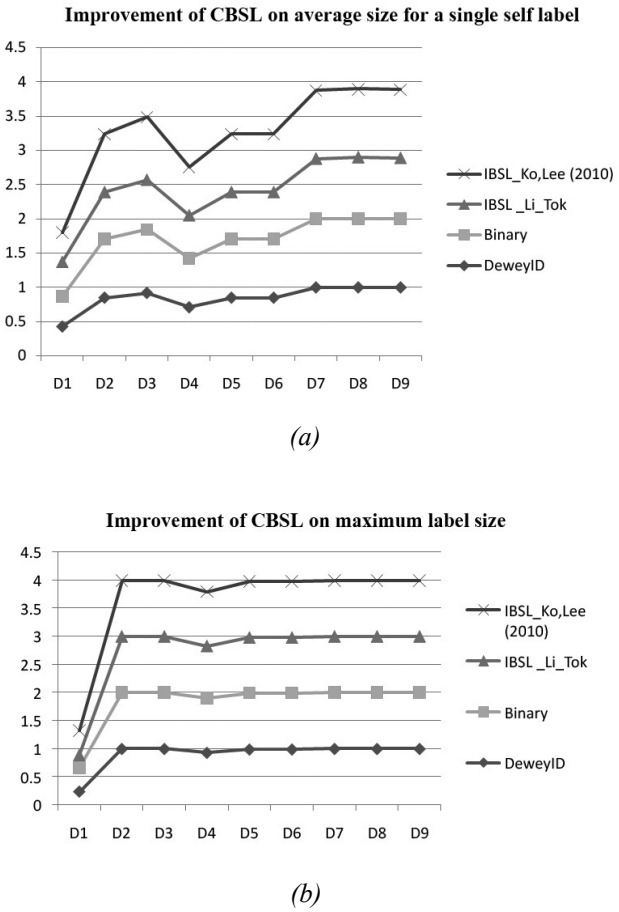


Figure 10. Improvement of CBSL *(a)* on average size of a single label, and *(b)* on maximum label size.

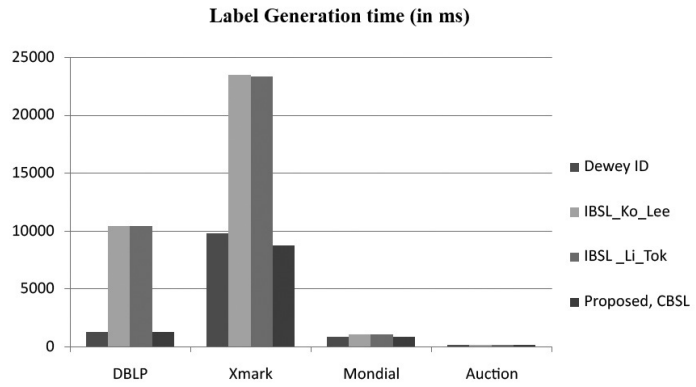


Figure 11. Label generation time in [ms].

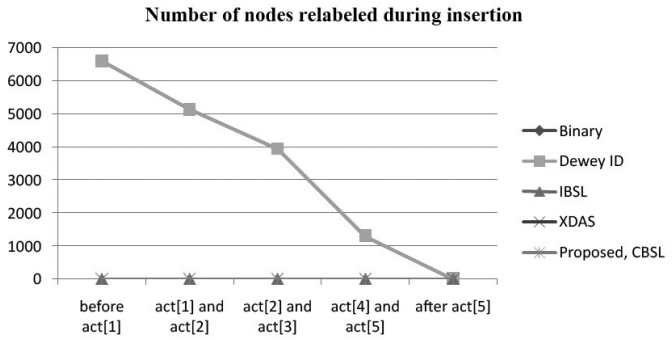


Figure 12. Number of nodes re-labeled during insertion operation.

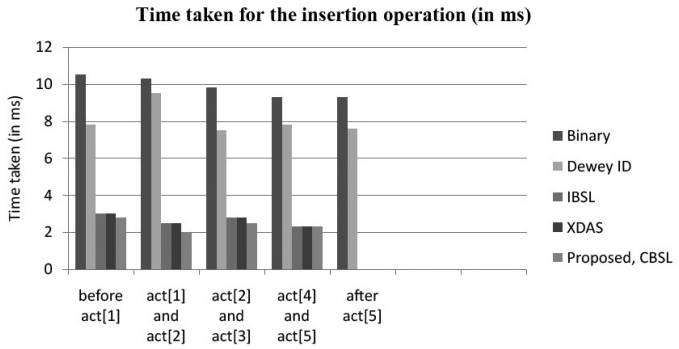


Figure 13. Time taken for each insertion operation in [ms].

documents without re-labeling existing labels, and efficiently recognizes all the structural relationships between the nodes. Additionally, it shows a compact label size. It is noted from the experimental results that CBSL takes less time to generate labels, and the update cost is lower in comparison with other labeling schemes. From the analytical results it is clear that, on different benchmark data sets, this labeling scheme takes less storage space, compared to Dewey ID, Binary, and IBSL.

References

[1] Q. Li *et al*., "Indexing and Querying XML Data for Regular Path Expressions", in *VLDB*, 2001.

[2] C. Zhang *et al*., "On Supporting Containment Queries in Relational Database Management Systems", in *ACM SIGMOD Record*, 2001.

[3] E. Cohen *et al*., "Labeling Dynamic XML Trees", *SIAM Journal on Computing*, vol. 39, no. 5, pp. 2048−2074, 2010.

[4] I. Tatarinov *et al*., "Storing and Querying Ordered XML using a Relational Database System", in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002.

[5] G. Dhanalekshmi and A. Krishna, "LPLX-lexicographic-based Persistent Labelling Scheme of XML Documents for Dynamic Update", *International Journal of Web Science*, vol. 2, no. 4, pp. 237−257, 2014.

[6] L. Xu *et al*., "DDE: from Dewey to a Fully Dynamic XML Labeling Scheme", in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, 2009.

[7] P. O'Neil *et al*., "ORDPATHs: Insert-friendly XML Node Labels," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004.

[8] M. Duong and Y. Zhang, "LSDX: a New Labelling Scheme for Dynamically Updating XML data", in *Proceedings of the 16th Australasian Database Conference*, vol. 39, 2005.

[9] H.-K. Ko and S. Lee, "A Binary String Approach for Updates in Dynamic Ordered XML Data",

IEEE Transactions on Knowledge and Data Engineering, vol. 22, no. 4, pp. 602−607, 2010.

[10] T. A. Ghaleb and S. Mohammed, "A Dynamic Labeling Scheme Based on Logical Operators: A Support for Order-Sensitive XML Updates", *Procedia Computer Science*, vol. 57, pp. 1211−1218, 2015.

[11] G. Miklau and D. Suciu, "XML Data Repository", University of Washington, 2003.

[12] [Online] Available: http://www.cs.wisc.edu/niagara/data.html

[13] C. Li and T. W. Ling, "An Improved Prefix Labeling Scheme: a Binary String Approach for Dynamic Ordered XML", in *International Conference on Database Systems for Advanced Applications*, 2005.

[14] J.-K. Min *et al*., "An Efficient XML Encoding and Labeling Method for Query Processing and Updating on Dynamic XML Data", *Journal of Systems and Software*, vol. 82, no. 3, pp. 503−515, 2009.

[15] J. Lu *et al*., "From Region Encoding to Extended Dewey: On Efficient Processing of XML Twig Pattern Matching", in *Proceedings of the 31st International Conference on Very Large Data Bases*, 2005.

Contact addresses:
Dhanalekshmi Gopinathan
Department of Computer Science
Jaypee Institute of Information Technology
Noida, Uttar Pradesh
India
e-mail: dhanalekshmi.g@jiit.ac.in

Krishna Asawa
Department of Computer Science
Jaypee Institute of Information Technology
Noida, Uttar Pradesh
India
e-mail: krishna.asawa@jiit.ac.in

Dhanalekshmi Gopinathan has received her M.Tech degree from the Department of Computer Science, National Institute of Technology, Calicut, Kerala in 2002. Her research interest covers database management systems, artificial intellegence, theory of computation and compiler design. She is currently doing research in the area of XML databases and query processing. She is pursuing her doctoral work under the supervision of Dr. Krishna Asawa.

Krishna Asawa is working with the Jaypee Institute of Information Technology (JIIT), Noida, India in the capacity of Professor. She was awarded the Doctor of Philosophy (CSE) degree in 2002 from Banasthali Vidyapeeth University, India. Her areas of interest and expertise are soft computing and its applications, information security, knowledge and data engineering. Before joining JIIT she has workedat the National Institute of Technology, Jaipur, India and Banasthali Vidyapith, India.

Received: December 2017
Revised: June 2018
Accepted: July 2018