An Efficient prefix based labeling scheme for Dynamic update of XML Documents

Article *in* International Journal of Advanced Intelligence Paradigms · January 2018

DOI: 10.1504/IJAIP.2018.10008500

2 authors:

Krishna Asawa
Jaypee Institute of Information Technology
52 PUBLICATIONS **289** CITATIONS
SEE PROFILE

Dhanalekshmi Gopinathan
Jaypee Institute of Information Technology
20 PUBLICATIONS **138** CITATIONS
SEE PROFILE

An efficient prefix based labelling scheme for dynamic update of XML documents

G. Dhanalekshmi* and Krishna Asawa

Department of Computer Science,
Jaypee Institute of Information Technology,
Noida Uttar Pradesh, India
Email: dhanalekshmi.g@jiit.ac.in
Email:krishna.asawa@jiit.ac.in
*Corresponding author

Abstract: The increasing volume of XML documents and the real-world requirement to support the updations has motivated the research community to develop dynamic labelling schemes. Each of the dynamic labelling schemes proposed till date differs in characteristics and has its own advantages and limitations. They may differ in terms of the query supported, their update performance, label size, etc. In this paper, a new prefix based labelling scheme is proposed which is compact, dynamic. And it also facilitates the computation of structural relationships which is the core part of query processing. The proposed scheme can handle both static as well as dynamic XML documents. The experimentation is conducted to evaluate the performance of storage requirement, structural relationship computation and update processing. The result is compared with some of the existing labelling mechanisms.

Keywords: labelling scheme; XML; structural relationship; dynamic update; ancestor-descendant; parent-child relationship; prefix-based; XML query processing; tree traversal; labelling time; label size; lexicographic order.

Reference to this paper should be made as follows: Dhanalekshmi, G. and Asawa, K. (2021) 'An efficient prefix based labelling scheme for dynamic update of XML documents', *Int. J. Advanced Intelligence Paradigms*, Vol. 18, No. 4, pp.464–480.

Biographical notes: G. Dhanalekshmi received her Master of Technology degree in Computer Science from the Regional Engineering College, Calicut, Calicut University, India. She is presently working as an Assistant Professor with Jaypee Institute of Information Technology (JIIT), Noida, India. She is currently pursuing her PhD from then JIIT. Her research interests include XML databases, information retrieval, principles of compilers and information systems.

Krishna Asawa presently working with Jaypee Institute of Information Technology (JIIT) deemed to be University, Noida, India in the capacity of Professor. She was awarded Doctor of Philosophy (CSE) in 2002 from the Banasthali Vidyapith, deemed to be University, Banasthali, India. Her area of interest and expertise includes soft computing and its applications, information security, knowledge and data engineering. Before joining to the JIIT, she worked with National Institute of Technology, Jaipur, India and with Banasthali Vidyapith.

1 Introduction

The expressive and extensible nature of XML makes it as a standard format for information representation and exchange over the World Wide Web. The increasing number of XML documents over the internet necessitates the development of mechanisms that helps to retrieve the data from the XML documents efficiently. In general, XML data may be very large and can have deep nested elements. And its ordered tree structured model provides the rich semantic content which is essential for querying the XML data. As the labelling schemes helps to store the order and structural information in a compact labels, it is widely chosen approach for XML query processing. Most of the research to date has focused on the construction of labelling schemes capable of efficient query processing and query optimisation over static XML data. However, as the volume of XML data increases and there is a requirement for a labelling scheme that can support efficient updates.

The labelling schemes assign unique labels to each node which is capable of holding the information about the node and its relationship with other nodes. Most of the existing labelling schemes support static documents only. During updation, the existing labelling schemes either re-label all or some of the existing nodes in the documents, or re-calculate some values. Another important issue in updation is that it has to preserve the document order. The order of the element is important as it has got influence on semantics of the XML. Hence, preserving document order while updating the XML documents is very important. There are many research have been conducted to support the document order (Cohen et al., 2010; Ko and Lee, 2010; O'Neil et al., 2004; Wu et al., 2004) while updating the XML documents. But the update cost for these approaches are still expensive.

In this paper, a labelling scheme is proposed which is compact in size and supports order sensitive update of XML documents.

The rest of the paper is organised as follows: Section 2 provides the related work Section 3 explains the detailed approach of the proposed labelling scheme. Section 4, elaborates the experiments conducted on the proposed system. The paper is concluded in Section 5 with the analysis and future work.

2 Related work

In general, the XML labelling schemes are categorised as range based, prefix based labelling schemes. Range based labelling schemes (Dietz, 1982; Li et al., 2001) are based on pre/post order traversal of the XML tree. The labels are assigned a pair of values start and end that cover the range of values of its descendant nodes. These schemes efficiently determine the A-D relationships, but they are not suitable for dynamic XML documents. During dynamic updating such as inserting a new node or sub tree may lead re-calculations of the labels of the existing nodes. This can be solved by increasing the interval size and reserving some numbers unused for later purpose. But this may lead to increase of space overhead. To avoid this re-labelling problem, (Amagasa et al., 2003), provides a scheme that uses a floating point values for the start and end intervals. However, this scheme also cannot solve the re-labelling problem if the insertions are done in the frequent manner. In general, all the range based labelling scheme has some

fixed interval, whenever the values between the range is used up; the schemes have to re-label or re-use the existing or deleted nodes.

Prefix based labelling schemes (Lu et al., 2005; Dhanalekshmi and Krishna, 2014; Ko and Lee, 2010) stores the information about its ancestors in the label itself. They can efficiently compute the structural relationship by looking at the label itself. DeweyID (Lu et al. 2005), is a prefix labelling scheme bases on integers. Each label specifies path from root to that node with a delimiter '.' indicates a unique path from the root to a specified node. The level of the node is obtained from the number of components in the label. This is static labelling scheme and re-labelling is required during updating. (Cohen et al., 2010) use Binary strings are used to label the nodes. In this scheme the root is labelled with an empty string. And the children nodes in the first level are labelled as 0, 10,110 and so on. For any node u, the children are labelled as L (u).0, L (u).10 and L (u).110 and so on. Here, L (u) label of the node u. O'Neil et al (2004) proposed an ORDPATH labelling scheme which is based on Dewey order. It uses odd numbers for initial labelling and reserves even and negative numbers for later insertions. But if the size of the reserved code overflows then it has to re-label existing nodes. In comparison with range based labelling scheme, the prefix based scheme needs to re-label the sibling nodes after the inserted node. (Ko and Lee, 2010), proposed improved binary string labelling (IBSL). Each label in the scheme uses binary bit strings. This scheme avoids re-labelling during updations. But space overhead and label size are not efficient. (Duong and Zhang, 2005) proposed the LSDX labelling scheme. Each label in this scheme is a combination of letters and digits. The label of the root node is 0a, where the integer 0 represents the level or depth of the node and the alphabet represents the self label of the node. Even though the LSDX is designed to meet the dynamic nature of xml data, it is not a persistent labelling scheme. And there are situations where collisions can occur during updation. Wu et al. (2004) proposed a prime number based labelling scheme. In this scheme prime numbers are used to label each node in the XML tree. Even though it supports order-sensitive updates without re-labelling the existing nodes, some of the SC values has to be recalculated based on the new ordering of nodes. This re-calculation is a very time consuming one.

However, a good labelling scheme should have the following proper-ties:

- Query efficiency: the labels should help to retrieve the data efficiently. Document Order: The document order should be preserved.
- Efficiently determine the structural relationships.
- Compact size: Label size should be compact
- Update efficiency: It should be able handle the updates without re-labelling or re-calculating existing values.
- Update cost should be minimum

A labelling scheme incorporating all the above properties is a challenging task. Earlier labelling scheme were unable to handle the dynamic updates. They may lead to re-label or re-calculate certain labels. Currently there are many dynamic schemes have been proposed but they may suffer from less compact size or update cost. Motivation behind this paper is to design a scheme which need not re-label the existing nodes when a new node is inserted and preserves the document order with less storage size for the labels.

3 Proposed approach

This section elaborates the proposed labelling scheme. It is a prefix based scheme and uses lexicographic order (defined in definition 2) to compare the labels. Each label is in the proposed scheme is a combination of uppercase alphabets and binary digits '0' and '1'. Each node 'n' is assigned a label as prefix (n), self-code (n). The prefix (n) denotes the label of the parent node and self-code (n) denotes the label of the node itself. The syntax of the node label is shown below.

$$Label(n) = prefix(n).\ Self-code(n);\ \text{where } n \text{ denotes the node to be labeled}$$

Definition 1: Longest common prefix (LCP): LCP of two strings, $X = x_1, x_2, ..., x_m$ and $Y = y_1, y_2, \ldots, y_n$, is the largest integer $L <= \min(m, n)$ such that, $x_1 = y_1$, s$x_2 = y_2, \ldots x_L = y_L$.

Definition 2: Lexicographical order ($\prec$): Two character strings 'A' and 'B' are lexicographically equal, if both A and B are exactly same, i.e., ($A = B$) and $A \prec B$ if and only if the following conditions are satisfied.

1 either length (A) = LCP (A, B)

2 length (A) > L, length (B) > L and A [L] $A[L] \prec B[L]$, where L denotes the LCP (A, B).

For example, consider two strings L_1 = 'A001' and L_2 = 'A01'. Here, LCP (L_1, L_2) = 2. Length (L_1) = 4 and length (L_2) = 3 which is greater than LCP (L_1, L_2) are greater than L. And L_1 [2] = '0' and L_2 [2] = '1'. It is known that 0 1. Hence, the string $A00 \prec A01$ as per the condition 2 in definition 2.

Consider another example, let the two strings L_1 = '01A' and L_2 = 'A'. And LCP (L_1, L_2) = 0. Length (L_1) = 3 and length (L_2) = 1 and L_1 [0] = 0 and L_2 [0] = 'A'. And it is known that $0 \prec A$.. Hence, the string $01A \prec A$ as condition 2 in definition 2.

3.1 Formal algorithm

The proposed approach works for both static and dynamic documents. It has two parts. The first part is the static labelling part, where the each node in the tree is assigned a unique label during parsing process. The static labelling part starts by assigning root node as 'A' where 'A' denotes the self code of the root node. And the child of the root node is labelled as 'A.A' (first child), 'A.B' (second child) and so on. When the number of children at one level exceeds 26, the next node will be labelled as 'A.ZA', 'A.ZB' and so on. Figure 1 illustrates the initial labelling part of our scheme.

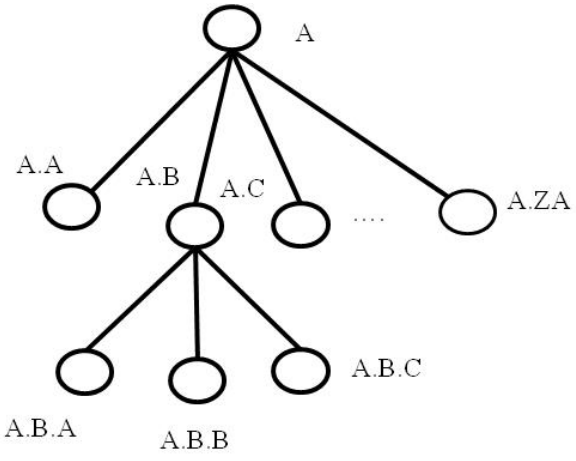
Lemma 1: The set of labels on the initial labels are unique.

Proof: Suppose the initial labels are not unique. Then, there exists two labels $X = x_1, x_2, \ldots, x_m$ and $Y = y_1, y_2, \ldots, y_n$ such that $x_1 = y_1$, $x_2 = y_2 \ldots x_n = y_n$, which means that X and Y are the same. This leads to a contradiction that all initial labels are unique. The fact is that during the initial label assignment, the new nodes are assigned a self code which is incremented value of its previous sibling node's self code.

Algorithm 1 Initial labelling

```
Input :XML document
Output: computed label for each node in the XML element
begin
if n ← root then
        self-code(n) ← "A"
if n ← FirstChild ( Parent (n)) then
        prefix(n) ← label(Parent(n))
        label (n) ← prefix (n) +delimiter + "A" //delimeter denotes "."
else
        //extract last character of the previous sibling
        if lastchar = "Z" then
                //append "A" to the self-code of previousSibling
        else
                self-code(n) self-code(previousSibling)++;
        end if
end if
end
```

Figure 1 Initial labelling of proposed approach



Property 1 (document order) label 'A' precedes 'B' in document order if and only if 'A' 'B' in lexicographic order.

Now, the second part handles dynamic labelling. In this section, the dynamic updation includes three cases such as

1 inserting a node before the leftmost child

2 inserting a node after the rightmost child

3 inserting in between two adjacent siblings.

And the algorithm 2 explains how these three cases are handled by the proposed scheme.

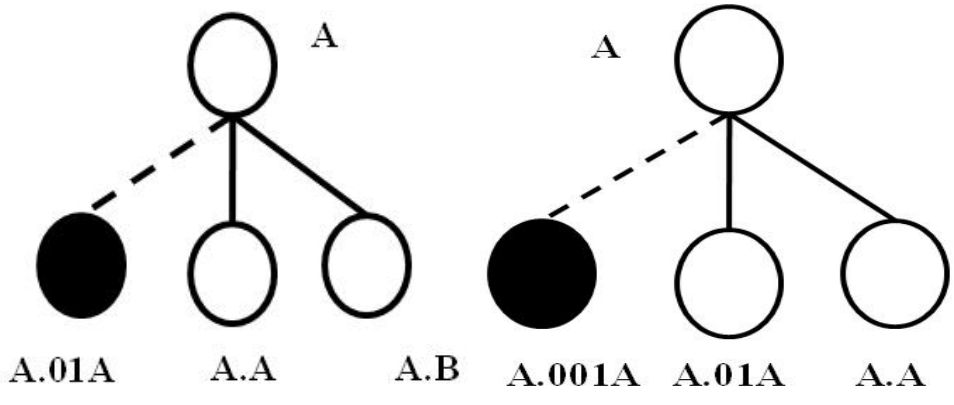
Algorithm 2 Dynamic labelling

```
Input: label_left, label_right,
Output: newNode label_new
Case 1: //Inserting a Node before leftmost node
begin
if (label_left contains binary_string) then
                label_new ← "01" self-code(label_left)
else
                label_new ← "0" self-code(label_left)
end if
end
Case 2: //Inserting a node after the rightmost node
begin
if (label_right contains binary_string) then
                label_new self-code(label_right) + '1';
else
                if (lastChar(label_right == 'Z') then
                        label_new self-code(label_right) 'A';
else
                        label_new increment(lastChar(self-code(label_right)));
                end if
end if
end
Case 3: //Inserting a Node between two adjacent siblings
begin
        if (length(label_left) <= length(label_right) then
                if(label_right contains a binaryString) then
                        Temp extractBinaryString(Label_Right)
                        newTemp ← Change MSB to "0" and prefix a "0" to it
                        label_new ← label_tight-Temp⊕ newTemp;
                else
                        label_new ← "01" label_left
                end if
        else
                label_new ← label_left ⊕ "1";
        end if
end
```

Case 1 of algorithm 2 illustrates how the label is assigned to a node inserted before a leftmost sibling node.

The algorithm checks two conditions. The first condition is, if the label_left has a binary string and the second condition is that if a label without a binary string. In the case of labels without a binary string, the label of inserted node will be a '01' + label of the leftmost sibling as shown in Figure 2(a).

Figure 2 (a) Inserting a node before leftmost node without binary string (b) Inserting a node before leftmost node with a binary string



(a) (b)

For example, suppose a node has to be inserted before the leftmost sibling node with label 'A'. Then, as defined in case 1 of algorithm 2, the new node will be assigned a label as '01A'.

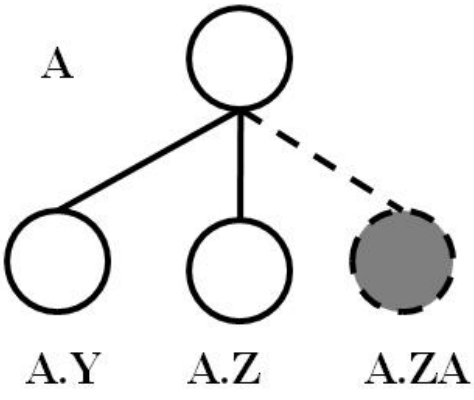
Figure 2(b) illustrates the second condition that, if the leftmost label contains binary string. Suppose a new node has to insert before a node with label '01A'. Here, the label contains a binary string '01' in it. Hence, as the condition 2 defined in case 1 algorithm 2, the new node will get a label by prefixing a '0' to the self-code of the left most nodes. Here, the new node is assigned a self-code as '001A'.

Case 2 of algorithm 2 is describes how to assign a label to new node inserted after the rightmost sibling. To insert a node after a rightmost child with binary string, it simply concatenates a '1' to it. And the second case, to insert a node after a rightmost child without a binary string works as the same way as in LPLX system (Dhanalekshmi and Krishna, 2014). The algorithm checks the last character of the preceding sibling. If the character is 'Z', then the character 'A' is concatenated with the self code of the preceding sibling to obtain the new label. Otherwise the last character of the preceding sibling is incremented to obtain new label.

For example, suppose a node is inserted after a node with label 'A.Z', then the new node will get the label as 'A.ZA' as shown in Figure 3. And the next node after 'A.ZA' will be 'A.ZB', 'A.ZC' and so on.

Next, the case 3 of algorithm 2 describes how a new node is assigned a label when it is inserted between two adjacent siblings. This is handled by taking care of two conditions. The first condition is that, if the size of the left sibling node's label is less than or equal to the size of the right sibling node's label. And the second condition is that, if the left sibling node's label is greater than the size of the right sibling node's label.

Figure 3 Inserting a node after the rightmost child



3.1.1 Size of the left sibling node's label is less than or equal to the size of the right sibling node's label

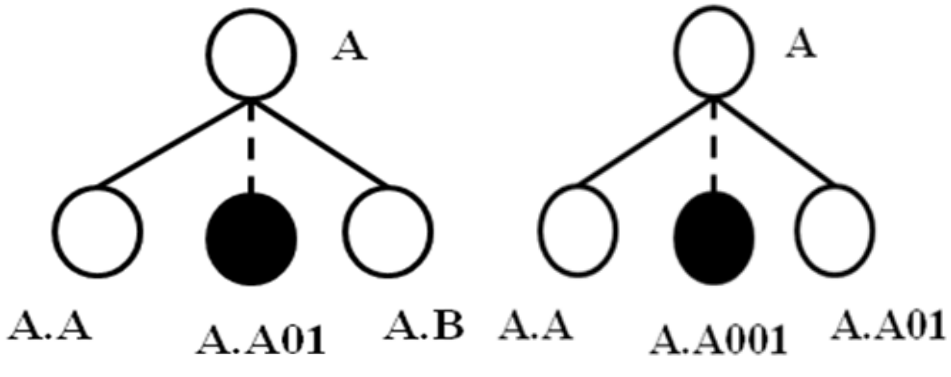
a If the label of the right sibling node has only alphabet characters :In this case, the label of the new node is the label of the left sibling node concatenated with a binary string '01'.

For example, suppose a node has to be inserted between two adjacent nodes with labels 'A.A' and 'A.B'. The newly inserted node is assigned a label 'A.A01' as shown in Figure 4(a). The label satisfies the lexicographic order and document order. ' $A \prec A01 \prec B$.

b If the label of the right sibling node has binary string: In this case, the most significant bit (MSB) of the binary string is changed to '0' and prefix a '0' before the MSB of the binary string.

In Figure 4(b) inserting a new node between two siblings with the label 'A' and 'A01 is shown. The right sibling has the binary substring '01'. The MSB is changed to '0' and a '0' is prefixed to it. Hence the new node gets the label as 'A.A001'.

Figure 4 (a) Inserting a node between adjacent siblings without a binary string (b) Inserting a node between adjacent siblings with a binary string

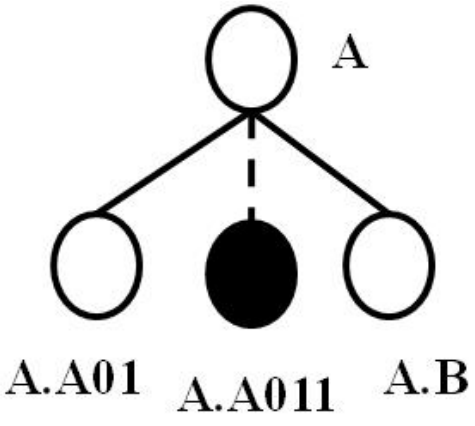


(a) (b)

3.1.2 Size of the label of the left node is larger than the size of the label of the right node

This case is handled by concatenating a '1' to the self-code of left sibling node. Consider an example shown in Figure 5. To insert a new node between the two adjacent sibling nodes with self-codes 'A01' and 'B'. Hence, by the algorithm 2, the newly inserted node is assigned a label 'A011'.

Figure 5 Inserting a node between two nodes with size (left label) > size (right label)



3.2 Correctness lexicographic order of labels

To prove the correctness of the lexicographic order to the labels three cases are considered. The label in this section denotes the self-code of the node.

3.2.1 Inserting a node before a leftmost child

a This case inserts a node to the leftmost sibling without a binary string. For this, consider a label say, L_1 = 'A' which denotes the leftmost child of the root node. Suppose a new node is to be inserted before L_1. Then, the new node, L_{new} is assigned a label 'A01' as specified in case 1 of algorithm 2.

   Now, LCP (L_{new}, L_1) = 0. And L_{new} [0] = '0', L_1 [0] = 'A'. It is known that $0 \prec A$. Hence, as per the condition 2 of definition 2, $L_{new} \prec L_1$.

b The second case is to insert a node to the left most node with a binary string. For this, consider a left most child with a label L_1 = '01A'. Then, according to the case 1 of algorithm 2, the new node L_{new} is assigned a label as '001A'.

   Now, LCP (L_{new}, L_1) = 1. And L_{new} [1] = '0', L_1 [1] = '1'. It is known that $0 \prec 1$. Hence, as per the condition 2 of definition 2, $L_{new} \prec L_1$.

   From (a) and (b), it is clear that inserting a node before the leftmost child follows the lexicographic order.

3.2.2 Inserting a node after the rightmost child

a This case inserts a node after the rightmost sibling without a binary string. For this, consider a label say, L_1 = 'Z' which denotes the right child of the root node. Suppose a new node is to be inserted after L1. Then, the new node, L_{new} is assigned a label 'ZA' as specified in case 2 of algorithm 2.

Now, LCP (L_{new}, L_1) = 1. And length (L_1) = LCP (L_{new}, L_1), hence as per the condition 1 of definition 2, $L_1 \prec L_{new}$.

b The second case is to insert a node after the right most child with a binary string. For this, consider a rightmost child with a label L_1 = 'B01'. Then, according to the case 2 of algorithm 2, the new node L_{new} is assigned a label as 'B011'.

Now, LCP (L_{new}, L_1) = 3. And length (L_1) = LCP (L_{new}, L_1), hence as per the condition 1 of definition 2, $L_1 \prec L_{new}$.

From (a) and (b), it is clear that inserting a node after the rightmost child follows the lexicographic order.

3.2.3 Inserting a node between two adjacent siblings

a size of the left sibling node's label is less than or equal to the size of the right sibling node's label

- If the label of the right sibling node has only alphabet characters. In this case, the label of the new node is the label of the left sibling node concatenated with a binary string '01'. To illustrate this, consider two adjacent labels L_1 = 'B' and L_2 = 'C', then as per the algorithm 2, the new node L_{new} = 'B01'.
- Now, LCP (L_{new}, L_1) = 1. And length (L_1) = LCP (L_{new}, L_1). Hence, as per the condition 1 of definition 2, $L_1 \prec L_{new}$.. Next, the LCP (L_{new}, L_2) = 0. And length (L_{new}) > LCP (L_{new}, L_2). L_2 [0] = 'B' and L_2 [0] = 'C'. It is know than $B \prec C$, thus, $L_{new} \prec L_2$.
- Hence, $L_2 \prec L_{new} \prec L_2$, when the size of the left sibling node's label is less than or equal to the size of the right sibling node's label and right sibling without binary string.
- If right sibling node has binary string, then as per the case 3 of algorithm 3, the new node is assigned a label by changing to '0' and prefixes a '0' before the MSB of the binary string.
- Now, consider two labels, L_1 = 'A' and L_2 = 'A01' and L_{new} = 'A001'.
- Now, LCP (L_{new}, L_1) = 1. And length (L_1) = LCP (L_{new}, L_1). Hence, as per the condition 1 of definition 2, $L_1 \prec L_{new}$. Next, the LCP (L_{new}, L_2) = 2. And length (L_{new}) > LCP (L_{new}, L_2). L_{new} [2] = '0' and L_2 [2] = '1'. It is know than $0 \prec 1$, Thus, $L_{new} \prec L_2$.
- Hence, $L_1 \prec L_{new} \prec L_2$, when the size of the left sibling node's label is less than or equal to the size of the right sibling node's label and rightmost label with binary string.

- Thus, $L_1 \prec L_{new} \prec L_2$, when the size of the left sibling node's label is less than or equal to the size of the right sibling node's label.

b size of the label of the left node is larger than the size of the label of the right node. As per the algorithm 3, the new node in this case is assigned a label by concatenating a '1' to the self-code of left sibling node. Consider for example, two labels L_1 = 'A01' and L_2 = 'B'. Now, L_{new} = 'A011'.

Now, LCP (L_{new}, L_2) = 3. And length (L_1) = LCP (L_{new}, L_1). Hence, as per the condition 1 of definition 2, $L_1 \prec L_{new}$. Next, the LCP (L_{new}, L_2) = 0. And length (L_{new}) > LCP (L_{new}, L_2). L_{new} [0] = 'A' and L_2 [0] = 'B'. It is know than ' $A \prec B$, thus, $L_{new} \prec L_2$.

Hence, $L_1 \prec L_{new} \prec L_2$, when the size of the left sibling node's label is larger than the right sibling node's label.

Thus, from (a) and (b), inserting a node between two adjacent siblings follows lexicographic order.

The next section discusses how the structural relationships are computed for the proposed scheme.

3.3 Structural summary

As structural relationship plays an important role in XML query processing, this section elaborates how the structural relationship between two nodes are computed.

Rule 1 Ancestor-descendant relationship: A-D (A, B): The node A is ancestor of node B, if the label B contains the Label A as a substring in the prefix part of it.

For example consider two node with labels L_1 = 'A' and L_2 = 'A.A.B' respectively. It is noted that, prefix (L_2) = 'A.A' and $L_1$is a substring of prefix (L_2). Hence, L_1 is ancestor of L_2as defined in rule 1.

Rule 2 Parent-child relationship: P-C (A, B): The node A is parent of node B, if the prefix of node B is label A and level of B is one more than level of A.

For example, Let L_1 = 'A.B' and L_2 = 'A.B.B'. Then L_1 is parent of L_2 since L_1 is prefix (L_2) and level (L_2)-level (L_1) = 1.

Rule 3 Sibling relationship: Sibling (A, B): The node A is sibling of B if and only if the prefix of Label A is same as prefix of Label B.

For example, L_1 = 'A.A.B' and L_2 = 'A.A.C'. Prefix (L_1) = prefix (L_2) = 'A.A'. Hence, the nodes L_1 and L2 are siblings to each other.

3.4 Label size analysis

The proposed scheme computes the size of the labels in the same way as described in (Dhanalekshmi and Krishna, 2014) scheme. It calculates the total storage required for the nodes in each level. Let the total number of node at a given level L be N.

In the proposed scheme, labelling starts 'A', 'B', 'C' and so on. Once it reaches 'Z', the next nod, i.e., 27th node is labelled as 'ZA'. Thereafter the consecutive nodes are

labelled as 'ZB', 'ZC' and so on. Hence, if total number of nodes at a given level is less than 27, then the nodes will get a label of one character length. If the number of nodes is greater than 27 and less than 52, then first 26 nodes will get a label of length one character. And the next 26 children will get a self code of two characters and so on.

The total characters required are 120. In general, the total storage requirement for self code at a given level for N number of nodes can be obtained using the formula as follows:

$$\text{size of } (self-code) = \left(\sum_{w=1}^{\frac{N}{26}} (w*26) \right) + \left(N\%26 * \text{floor}\left(\frac{N}{26} \right) + 1 \right)$$

For example, for a level L, if number of nodes N = 66, then total storage required is 1 * 26 + 2 * 26 + 3 *14 = 120. This means that first 26 children gets self code of 1 character length, the next 26 children gets a self code of two characters of length and the remaining 14 nodes gets a self code of three characters of length. And the total number of symbols required will be 120.

4 Experimentation and results

The proposed system has been implemented in JAVA2 JDK5.0. Sun Microsystems SAX Parser is used to parse the XML documents. The performance evaluation is compared with existing labelling schemes Dewey, Binary, IBSL and LSDX. These labelling schemes are chosen because they are prefix based schemes and share same design goals.

There are four set of tests are conducted. The first set is label storage space requirement of labels. The result of this experiment is compared with the existing labelling scheme LSDX and IBSL. And the second set of experiments is conducted for evaluating labels generation time. The third set of experiment is conducted for evaluating the update performance. And the fourth set of experiments is done for the structural relationship computation and the results are analysed with the existing labelling scheme LSDX.

Table 1 Dataset from Xml data repository

Dataset	*#Nodes*	*Max depth*
Mondial	22,423	5
Reed University	10,546	4
Hamlet	6,636	5
Sigmod Record	11,526	6
TPC-H customer	13,501	3
TPC-H partsupp	48,001	3
TPC-H orders	15,001	3
XMark1	167,875	12

Source: Suciu and Miklau (2002)

4.1 Storage requirement analysis

In this section, the storage requirement for the three schemes are tested and compared. To measure the storage requirement, the labels generated are stored in a file and the total size of the labels in the XML documents is calculated from the file. The performance is measured by running the program five times on each data set and average is taken. The storage requirement of proposed schemes is the smallest as compared to the other two.

The graph in Figure 6 shows the result from the computation of storage space.

Figure 6 Storage space requirement

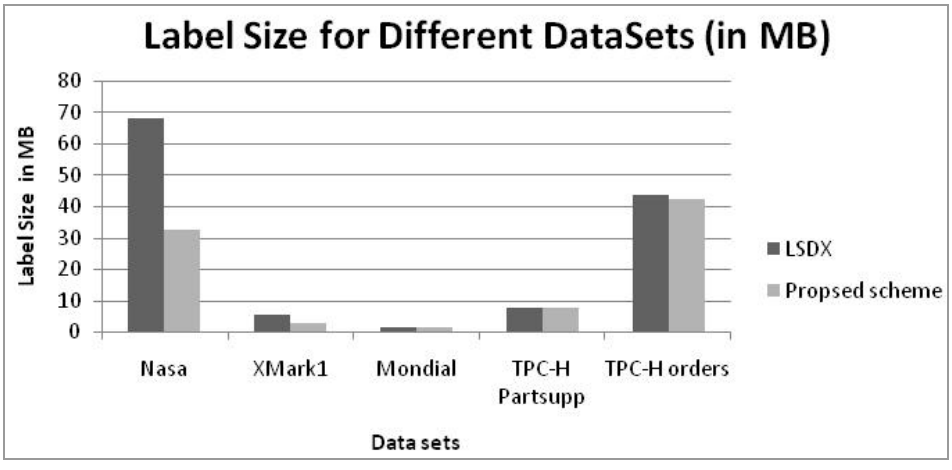
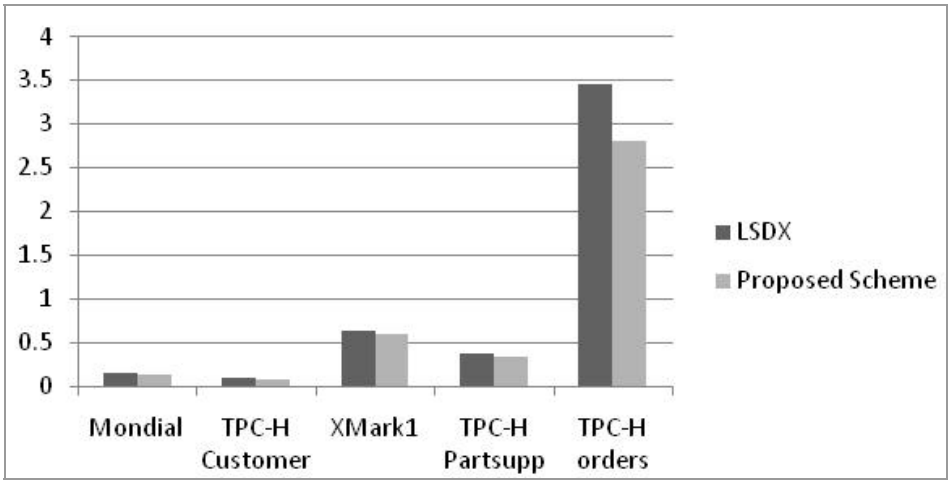


Figure 7 Label generation time



4.2 Label generation time

This set of experiment is conducted to measure the computation time taken to generate the labels. This experiment is conducted by passing the XML documents to the SAX

parser. The Parser generates the label as and when it notifies an element. The time is compared with the existing labelling scheme LSDX. Figure7 label generation time. The experimentation results shows that the proposed labelling scheme takes less time than the existing schemes.

4.3 Update processing

The experiment is conducted on the Shakespeare play on Hamlet file. The file contains a list of five ACT elements. The five different cases considered for the evaluation of update performance are

1 inserting a node before act [1]

2 between act [1] and [2]

3 between act [2] and act [3]

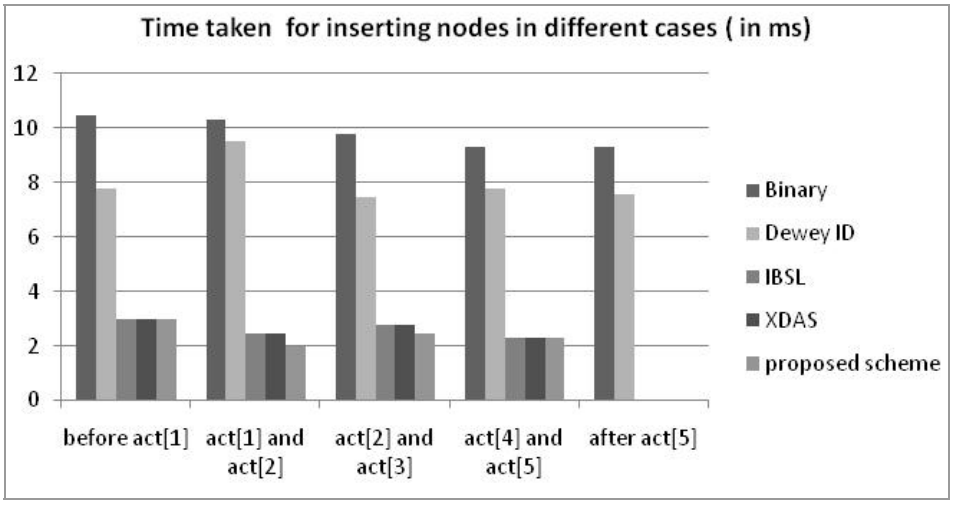
4 between act [3] and act [4]

5 after act [5].

This is compared with existing labelling schemes, Dewey ID, binary and IBSL scheme.

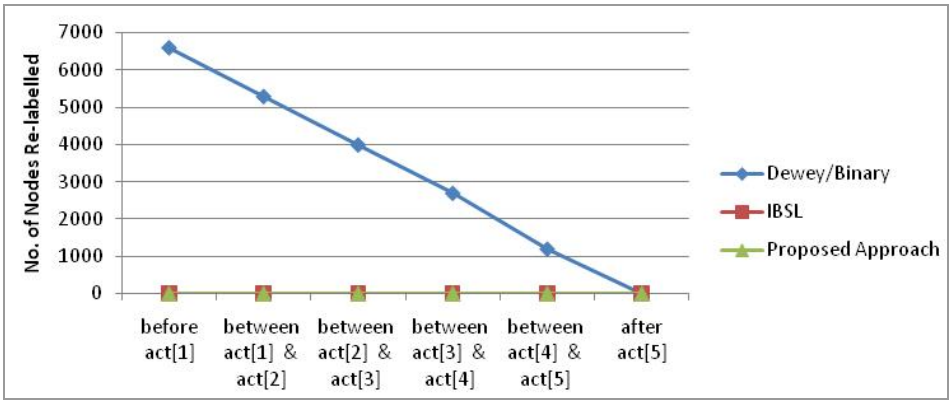
In this section, the time required to perform the insertion operation is measured and the result is shown in Figure 8. From the result, it is clear that IBSL, XDAS and the proposed scheme takes almost same time for the insertion operation whereas DeweyID and binary took more time for the insertion in all cases.

Figure 8 Update processing time



Next, Figure 9 shows the number of nodes re-labelled during the update performance. The Hamlet file has a total of 6,636 nodes. Dewey and binary schemes needs to recalculate the 6,610 nodes when inserting an act before the first child act [1], while IBSL and proposed approach does not need to recalculate it. Figure 9 shows result for the evaluation test.

Figure 9 Number of nodes re-labelled during update processing (see online version for colours)



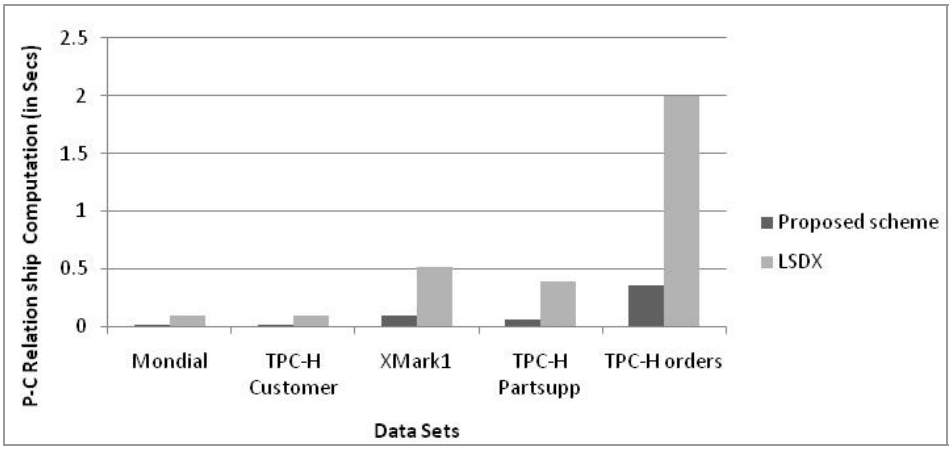
4.4 *Structural relationship computation*

In this section, the proposed system evaluates the performance of time taken to compute the structural relationship computation between the nodes. The results are compared with an existing labelling scheme, LSDX.

4.4.1 *P-C relationship computation*

This set of experiment is conducted to find the parent of a given node. To evaluate this, the labels generated by the proposed scheme are stored in a file. And the method to check the parent as defined in rule 2 is executed on this file. The experiment is performed by executing the program around 15 runs and the average is taken. The proposed scheme takes an average computation time of 0.109 secs to compute the relationship, whereas the existing scheme LSDX takes an average computation time of 0.61 secs to determine the P-C relationship. The proposed system shows an improvement of 82% over the existing compared scheme.

Figure 10 P-C relationship computation time (in secs)



4.4.2 Ancestor-descendant relationship

This set of experiments is conducted to find whether a node is ancestor of another node as defined in rule 1. To evaluation is performed the same way as done in P-C relationship computation. And it is measured that the average computation time for the LSDX for determining A-D relationship is 0.882 secs, whereas the proposed scheme takes 0.6 secs. It is observed that the proposed system shows an improvement of 12.8% over the compared scheme.

Figure 11 A-D relationship computation time (in secs)

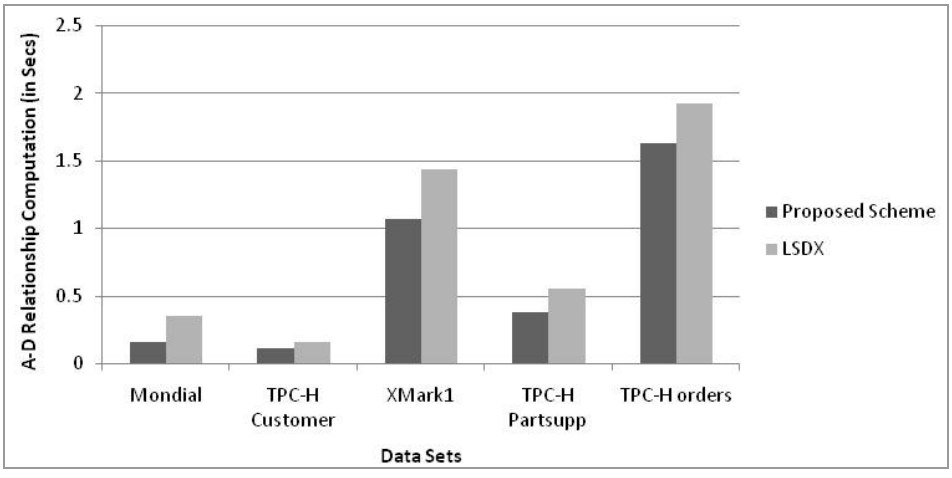
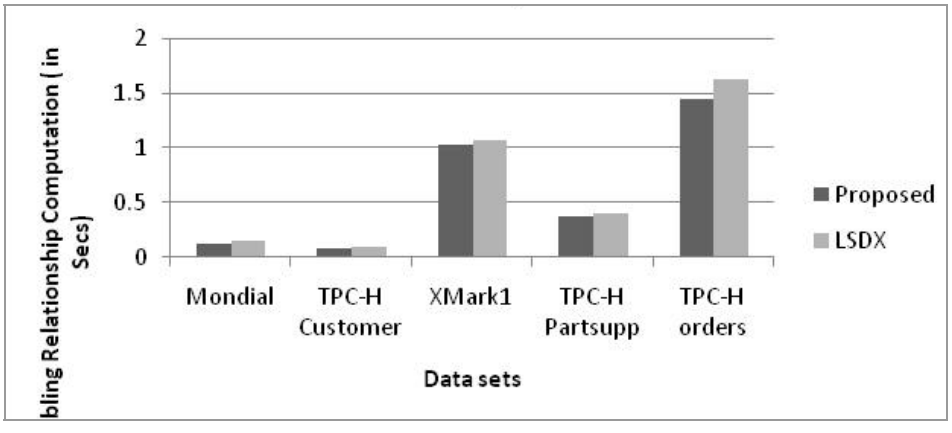


Figure 12 Sibling relationship computation time (in secs)



4.4.3 Sibling relationship computation

This set of experiment is conducted to find the sibling relationship between the two nodes in a XML tree. Two node labels are passed as an argument to this method and returns true if Label1 is sibling of label 2. And the result is compared with LSDX scheme. It takes an average time of 0.6748 sec for computing sibling relationship, whereas the proposed

scheme takes an average time of 0.61 sec for the computation. The proposed system shows an improvement of 9.24% over the compared scheme.

5 Conclusions

Motivated from the need to efficiently support an order sensitive update this paper proposes a new prefix based labelling scheme which follows lexicographic order. It supports dynamic updation without re-labelling existing nodes. The empirical studies are conducted to compute the storage space overhead, generation time, update performance and structural relationship computation. And the results indicate that the proposed scheme is compact in label size compared to existing labelling schemes and update cost is low and determination of structural relationships can be efficiently computed. In the future, we will investigate how to reduce the label size further and evaluate the query performance using the proposed labelling scheme.

References

Amagasa, T., Yoshikawa, M. and Uemura, S. (2003) 'QRS: a robust numbering scheme for XML documents', *Data Engineering, 2003*, *Proceedings, 19th International Conference on*.

Cohen, E., Kaplan, H. and Milo, T. (2010) 'Labeling dynamic XML trees', *SIAM Journal on Computing*, Vol. 39, No. 5, pp.2048–2074.

Dhanalekshmi, G. and Krishna, A. (2014) 'LPLX-lexicographic-based persistent labelling scheme of XML documents for dynamic update', *International Journal of Web Science*, Vol. 2, No. 4, pp.237–257.

Dietz, P.F. (1982) 'Maintaining order in a linked list', *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*.

Duong, M. and Zhang, Y. (2005) 'LSDX: a new labelling scheme for dynamically updating XML data', *Proceedings of the 16th Australasian Database Conference*, Vol. 39.

Ko, H-K. and Lee, S. (2010) '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.

Li, Q., Moon, B. et al. (2001) 'Indexing and querying XML data for regular path expressions', *VLDB*.

Lu, J., Ling, T.W., Chan, C-Y. and Chen, T. (2005) 'From region encoding to extended dewey: on efficient processing of XML twig pattern matching', *Proceedings of the 31st International Conference on Very large Databases*.

O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G. and Westbury, N. (2004) 'ORDPATHs: insert-friendly XML node labels', *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*.

Suciu, D. and Miklau, G. (2002) *XML Data Repository*, Computer Science and Engineering Department, University of Washington November, Vol. 12 [online] http://www.cs.washington.edu/research/xmldatasets/.

Wu, X., Lee, M-L. and Hsu, W. (2004) 'A prime number labeling scheme for dynamic ordered XML trees', *Data Engineering, 2004, Proceedings, 20th International Conference on*, pp.66–78.