LPLX-lexicographic-based persistent labelling scheme of XML documents for dynamic update

Article *in* International Journal of Web Science · January 2014

DOI: 10.1504/IJWS.2014.070671

2 authors:

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

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

LPLX-lexicographic-based persistent labelling scheme of XML documents for dynamic update

G. Dhanalekshmi* and Asawa Krishna

Department of Computer Science,
Jaypeee 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 number of XML documents over the internet motivated us to develop indexing techniques to retrieve the XML data efficiently. Assigning unique labels to each node and determining the structural relationships is a critical problem in XML query processing. Labelling schemes designed for static XML documents will not support dynamic updates on XML documents. Some dynamic labelling schemes provide dynamic updates but, with a high cost and complexity. In this paper we propose a labelling scheme which supports the dynamic update without relabelling the existing nodes. It also determines the structural relationships efficiently by looking at the labels. A set of performance tests is carried to compute the time required to generate unique labels.

Keywords: XML query processing; dynamic updates; persistent labelling scheme; ancestor-descendant relationship; parent-child relationship; sibling relation; tree traversal; lexicographic order; XPath; labelling time; label size.

Reference to this paper should be made as follows: Dhanalekshmi, G. and Krishna, A. (2014) 'LPLX-lexicographic-based persistent labelling scheme of XML documents for dynamic update', *Int. J. Web Science*, Vol. 2, No. 4, pp.237–257.

Biographical notes: G. Dhanalekshmi received her Engineer degree in Computer Science and Engineering from Regional Engineering College, Calicut University 1995 and MTech in Computer Science from Regional Engineering College, Calicut University, in 2002. She is currently working with Jaypee Institute of Information Technology (JIIT), Noida, India as an Assistant Professor. She is currently pursuing her PhD from JIIT. Her research interests include XML databases and information retrieval, web technology and information systems.

Asawa Krishna is currently working with Jaypee Institute of Information Technology (JIIT), deemed to be university, NOIDA, India in the capacity of Associate Professor. She was awarded Doctor of Philosophy (CSE) in 2002 from 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 Banasthali Vidyapith, India.

1 Introduction

Owing to the increasing popularity of XML (extensible markup language) as a standard for information representation and exchange over the internet, storing and querying XML data becomes more and more important. The self describing nature of XML makes it a good candidate to represent semi-structured data which is large in volumes from different data sources and applications on the web. There are two approaches to store the XML document. The first approach is to store the XML document to the database model such as relational or object oriented. The second approach uses the native XML database in which data is stored and retrieved in their original hierarchical structure. This hierarchical structure is represented as a tree or graph, where nodes represent the elements, attributes, text data, and the edges represent the structural relationships between them. A number of query languages like XQuery, XPath (W3C Recommendation, 1999) exist to query XML database. All these languages use the concept of path expression to specify the path in the semi-structured data. Efficient query processing on XML data requires knowledge of the structural relationships such as ancestor-descendant, parent-child and sibling etc. The traditional way of identifying these relationships is to traverse the tree in depth first order. To avoid the time consuming tree traversal, many labelling schemes have been proposed in the literature. The basic idea in all these labelling schemes is to assign unique labels to the nodes of the XML tree in such a way that it will take less time to determine the relationships by looking at the labels rather than traversing the entire tree of the XML document. The existing labelling schemes are classified as containment labelling scheme (Dietz, 1982; Li and Moon, 2001; Zhang et al., 2001) and prefix-based labelling scheme (Duong and Zhang, 2005; O'Neil et al., 2004; Min et al., 2009; Harder et al., 2007; Cohen et al., 2002). Both the labelling schemes are in use in many applications. Each of the labelling schemes proposed till date has its own characteristics and advantages and limitations. Good labelling scheme should satisfy the following labelling criteria.

- Structural information should be preserved: it deals with how the structural information is encoded in the label. How much time taken to extract this information from the label?
- Update cost should be minimised: update cost should be at minimum and the labels should be persistent. That is, there should not be any collision and relabelling of the existing labels when dynamic updating is being performed.
- Query processing should be efficient: it deals with how the structural information is retrieved during the query processing?
- Size of the label should be minimised: how much storage space is occupied by the labels.

Designing labelling schemes to satisfy all the above criteria is a challenging task. The contributions of this paper are listed as follows:

- proposes new labelling scheme, lexicographic-based persistent labelling (LPLX) scheme which assigns a unique label in lexicographical order to each node in the XML tree with the tuple format (prefix, level, selfcode)

- this LPLX supports dynamic update of XML documents without relabelling the existing nodes
- the structural relationships such as ancestor-descendant, parent-child and siblings have computed at constant time
- experiments were conducted for XML data sets of varying number of nodes and different depth.

The rest of the paper is organised as follows: Section 2 introduces the preliminaries and related works. Section 3 describes the proposed approach. In Section 4 experimentation results on time required to generate label are described. At the end the discussion and conclusion are described in Section 5 and Section 6 respectively.

2 Background and related work

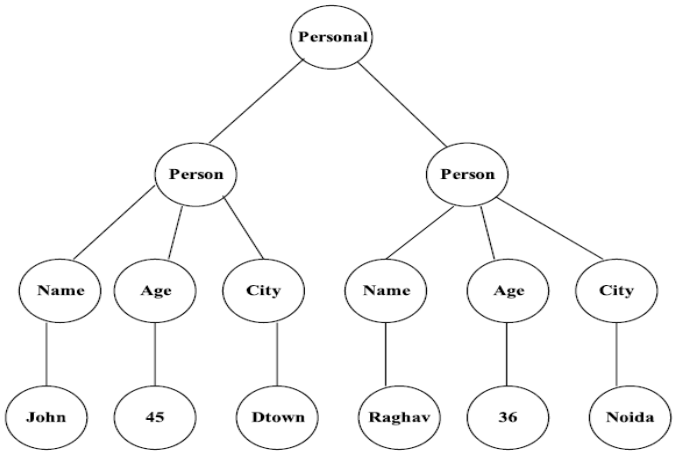
2.1 XML model

W3C specification (W3C Recommendation, 1999) models an XML document as a rooted, ordered labelled tree. Each node in the tree represents an element, attribute or a text value, and the edges represent the structural relationship between the nodes. Efficient processing of query requires the quick determination of structural relationships such as ancestor-descendant, parent-child and sibling between any pair of nodes in the tree. To support the query processing, several labelling mechanisms have been proposed which eliminates time-consuming tree traversal task. Figure 1 represents the sample XML document and Figure 2 represents its corresponding XML Tree.

Figure 1 Sample XML document

```
<Personal>
<person>
    <Name>“John”</Name>
    <Age>45</Age>
    <City>DTown</City>
</person>
<person>
    <Name>“Ragav”</Name>
    <Age>36</Age>
    <City>Noida </City>
</person>
</Personal>
```

Figure 2 XML tree of the XML document in Figure 1



For example in Figure 1, Q1: personal//city, to retrieve the cities of all the persons. The answer for the query Q1 is a set of values {'DTown', 'Noida'}, which represents all *city* elements under the personal element. The matches are retrieved by traversing the tree starting from the context node (defined in Definition 1) *personal* down to its child one by one to check for the *city* element. Once the leaf node is reached, the set of matches are backtracked to the *personal* element node and start search for the next node. This type of search is very expensive in cost and exhaustive in search. It also slows down the query processing performance. As these problems can be overcome by some indexing or labelling mechanism which reduces the search space and improves the query performance

Definition 1: Context node: a context node is the node, the XPath processor is currently looking at. The context node changes as the query evaluation proceeds. The initial context node for a document is the root node.

Definition 2: Tree traversal: is the process of visiting each node in a tree data structure. Such traversals are characterised by the order in which the nodes are visited.

Definition 3: Labelling scheme: it is a method of assigning unique label to each node in the XML document.

The core issue of the query processing is to identify the structural relationships among the nodes in the XML document efficiently. The labelling scheme determines the structural relationships between the nodes such as parent-child, ancestor-descendant, and sibling by comparing their labels. Based on the labels, the query can be processed without accessing the entire XML document. The next section describes existing related work in brief.

2.1.1 Motivation

The node labelling schemes are very helpful in effective management of XML documents. Initial researches on XML were focused on static documents where navigation and retrieval was the limited functionality upon it. But, as and when the number of documents in the WWW is increased, the node labelling schemes helps to

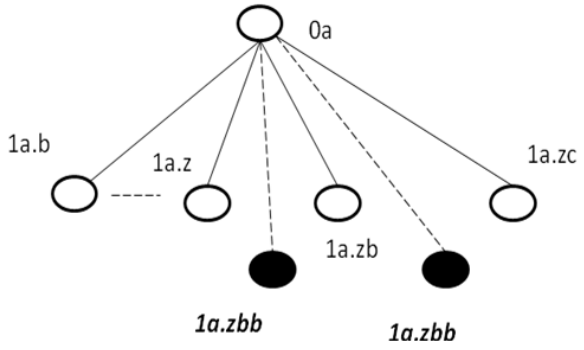
retrieve them in an efficient manner by reducing the search space. Motivated from this we have proposed a new labelling scheme which helps to retrieve the XML node efficiently. It also supports the dynamic updates without relabelling the existing nodes in the XML documents.

2.2 Related work

The labelling scheme as defined in Definition 3 reduces the search space in query processing. The existing labelling schemes are mainly classified as containment labelling schemes and prefix-based labelling schemes. In containment labelling schemes (Dietz, 1982; Li and Moon, 2001; Zhang et al., 2001), each node is labelled with triple (start, end, level), where start and end represent the range of numbers in which all its descendant node labels are included. In the paper, Zhang et al. (2001) propose a labelling scheme which uses a label of the format (start, end, level). For any two nodes A (s_1, e_1, l_1) and B (s_2, e_2, l_2), where s_1, and s_2 denotes the start values for the node A and B. e_1, e_2 represent the end values for node A and B. l_1 and l_2 represent the levels of nodes A and B. Node A is ancestor of Node B if and only if $s_1 < s_2$ and $e_1 > e_2$. Node A is parent of node B if and only if A is ancestor of B and $l_2 - l_1 = 1$. This means that range of B is contained in range of A. This scheme determines the ancestor-descendant and parent-child relationship very efficiently. But it is very inefficient to find the sibling relationship. It needs to find the parent of the node and determine the other nodes is also have the same parent, which needs to determine a number of parent-child relationships recursively and it is very time consuming as well as costly also. Another limitation of this scheme is that it does not support dynamic updates on XML documents. The whole XML tree has to be relabel once a new node is inserted in the tree. This is because the change in the pre-order and post-order tree traversal. Many researches (Su-Cheng et al., 2009; Tatrinov et al., 2002; Xu et al., 2009, 2012; Yu et al., 2005) have performed for handling dynamic update of XML data. In a prefix labelling schemes (Duong and Zhang, 2005; O'Neil et al., 2004; Min et al., 2009; Harder et al., 2007; Cohen et al., 2002), the label of a node encodes its parent information also in its label. Each label consists of three components of the form ($prefix_n$, separator, $selflabel_n$). A *prefix*, which often represents the label of all the ancestors of the node. A separator, which in most cases is the full stop '.'. A self-label, which indicates the position of the node relative to its siblings. The advantage of prefix-based labelling scheme is that it can quickly determine all the structural relationships by looking at the labels only. The main issue with this prefix-based scheme is that it requires large storage if the tree length goes deeper. Simple prefix labelling scheme (Cohen et al., 2002), the root of the tree is labelled with empty string and the first child of the root is labelled as '0', second child is labelled as '10', the third child is labelled as '110', the fourth child is labelled as '1110' and so on. For any node u, the first child of u is labelled as Lab(u).0; second child of u is labelled as Lab(u).10 and so on. Lab(u) denotes the label of the node u. The i^{th} child of node is labelled as $Lab(u).111^{i-1}0$. For any pair of nodes u and v, L(v) is a prefix of L(u) if and only if v is the ancestor of u. This numbering scheme easily identifies the ancestor/descendant relationships. The limitation of this approach is that renumbering is required if a node is added as a sibling other than the right of the last sibling. ORDPATH (O'Neil et al., 2004), proposes an ORDPATH labelling scheme encodes the Parent-Child relationship by including parent's ORDPATH label as a component of child's label. For example 1.3

is parent's node which is included as 1.3.1 for the child's node label, which denotes that 1.3.1 is the first child of the parent with label 1.3. The root is labelled with 1. Initially only positive odd numbers assigned for labelling the nodes. Even numbers and negatives are reserved for future insertions into the tree. The number of division in the label gives the level of the labelled nodes. This technique is efficient to encode the P-C relationship and sibling relationship. This scheme supports dynamic updates without changing the existing labels. The limitation of this approach is that it is not suitable for deep XML trees. The relabeling is unavoidable if the reserved space is exhausted. LSDX (Duong and Zhang, 2005) is a fully dynamic prefix labelling scheme. It generates labels of large sizes and sometimes it undergoes collisions. For example in the Figure 3, suppose we need to add node between the nodes labelled '1a.z' and '1a.zb', according the LSDX labelling scheme the new node gets the new label as '1a.zbb'. Now suppose we need to add one between the two nodes labelled '1a.zb' and '1a.zc', the new node will also obtains the label '1a.zbb' which results in collision.

Figure 3 Example of collision



The next section we discuss our proposed approach for labelling XML document. The characteristics of the proposed system are:

a it efficiently determines the structural relationships between any two nodes in the tree

b It supports the dynamic updates without relabeling the existing nodes in the tree

c It avoids collisions as mentioned in the LSDX (Duong and Zhang, 2005) labelling scheme.

In proposed system, whenever a node is inserted between two nodes, say x1 and x2, the new node is assigned a selfcode as concatenating the selfcode of x1 and x2 which will be always a new string and also satisfies the lexicographic order. This avoids the collision. The proposed system is tested for computing the time for generating the labels and storage requirement.

3 Proposed solution

The proposed labelling scheme is based on the combination of digits and letters. Each label consists of a tuple having three parameters like (prefix, level, selfcode).The first

parameter prefix indicates the label of its parent node. Prefix part contains strings which is the combination of letters and digits. The second parameter Level is an integer which indicates the depth of the node from the root of the XML tree and the third parameter indicates the self label of the node. In the proposed system, the digits (0–9) and uppercase letters (A–Z) are used in combination to label each node. The nodes are labelled in the lexicographic order as defined in Definition 6.

Definition 4: The alphabet $\sum$ is finite set of symbols of the language, and $\sum^*$ is finite sequence of strings over the alphabet $\sum$. Example: $\sum$ = {0, 1, A, B}, then the language consists strings over this alphabet $\sum$ {001A, AB01, 001BA …}.

Defintion 5: Least common prefix: Least common prefix of two strings x = x1 x2 … xm, and Y = y1 y2 … yn, is denoted by lcp(X, Y) is the largest integer l <= min(m, n) such that x1 x2 … xl = y1 y2 … yl.

Defintion 6: Lexicographical order (<): character string *a* is lexicographically equal to character string *b,* if both *a* and *b* are exactly same (*a* = *b*), *a* < *b* if and only if the following conditions are satisfied.

a either *a* is a prefix of *b*

b length (a) > l, length (b) > l and a[l] < b[l], l denotes the least common prefix.

The next section discusses the LPLX labelling scheme that supports the update processing and query processing. The proposed scheme uniquely labels each node in the XML document by traversing the elements in the documents in the depth first traversal manner. It also supports the dynamic update of the tree and assigns new label for the new node without re-computing labels of the existing nodes.

The labelling scheme labels each node as shown

$$\text{LPLX Label}(n, x, y, z) = \text{Prefix}(x).\text{Level}_n(y)\text{Selfcode}_n(z)$$

where

$n \varepsilon \{1, 2, \ldots N\}$ denotes the node to be labelled and N = total number of nodes in a tree. x, $z \varepsilon \{w \mid w \varepsilon \sum^*\}$, where $\sum$ = {0–9 A–Z} and w is strings over $\sum$ as defined in Definition 4. $Y \varepsilon \{I/I \varepsilon N\}$, where N denotes set of natural numbers {0, 1, 2…}

$\text{Prefix}_n(x)$ the label of its parent node P denoted by Label_P,

$\text{Level}_n(y)$ level of the node n;

$\text{Selfcode}_n(z)$ the self label of the node n in the XML Document.

Example 1: LPLX Label L1 = 0A.1B, means, the $\text{Prefix}_n(x)$ = 0A, $\text{Level}_n(y)$ = 1, $\text{Selfcode}_n(z)$ = B.

Example 2: LPLX Label for the root node = '0A', since the prefix part is empty. The level is 0 and selfcode is denoted as A.

The proposed labelling scheme has two parts:

a initial labelling part

b dynamic labelling part.

The initial labelling part discusses how the nodes of XML document are labelled initially. The dynamic labelling part discusses how newly inserted node are labelled without the recomputing the labels of the existing node.

3.1 Initial labelling of proposed scheme

The initial labelling of proposed scheme LPLX is explained in algorithm 1 as shown below. The algorithm starts with a depth traversal of the XML Tree and assigns unique label to every node it visits. It checks whether the input node 'n' is the rootnode of the XML tree. If it is rootnode then a label '0A' is assigned. Next it checks whether the node 'n' is the first child of a parent node 'P'. If it is first child of 'P', then prefix of the current node is the label of the parent node P. And the label can be assigned as prefix(n) followed by the level of the current node and the character 'A'. If the node 'n' is not a first child of parent node 'P'. The algorithm stores the label of the previous sibling. If the last character of the previous sibling's label is 'Z', then the character 'A' is appended to it otherwise lexicographically creates the next character. Figure 4 is an example of the initial labelling scheme.

Algorithm 1 To initial labelling step

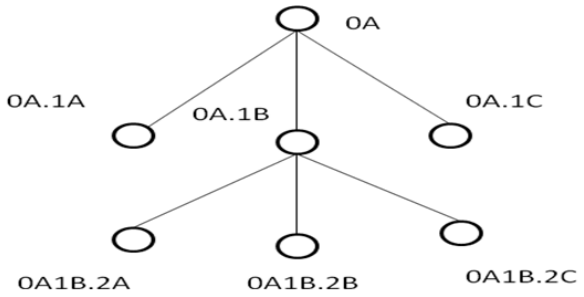
Input: n: n denotes the node to be labelled in the XML tree

Output: a label for node n.

```
1   if(n = rootnode)
        //Assign label "0A" to the root node;
2   if n is firstChild of a Parent Node 'P' then
        //prefix(n) = label(P);
        //label(n) = prefix(n) + "." + level(n) + "A".
3   if n is not the firstChild of a parent node then
        //getPreviousSibling's Selfcode;
        //Check whether selfcode of Previous sibling ends with a character "Z".
        if "yes" then
            //selfcode(n) = selfcode(PreviousSiblling) + "A";
        Else
    //selfcode(n) = selfcode(PreviousSibling)++;
```

In Figure 4, during the initial labelling step, the root node is labelled as '0A' initially, here '0' denotes the level of the root node and 'A' is the label of the root node. The children of root node at level 1 are labelled as 0A.1A, 0A.1B, and so on from left to right. The first child of the any node is assigned a label with three components (prefix, level and selfcode) as (label of its parent node, level of the current node, and selfcode as alphabet 'A'). The nodes other than the first child are labelled by lexicographically incrementing the selfcode of its previous sibling node. If the previous sibling node's selfcode ends with the alphabet 'Z', then the new node will be assigned new selfcode as selfcode of its previous sibling concatenated with letter 'A'. This is the special case in our proposed approach which is defined in Lemma 1 below.

Figure 4 LPLX labelling



Special Case: A parent node 'P' can have unlimited children, but then the selfcodes for children are limited by 26 alphabet characters. These children with the bound of 26 characters are called inbound children (as in Definition 7) of parent node 'P'. The excess children of 'P' are termed as outbound children (as in Definition 8). For outbound child, the labelling scheme works as in Definition 8.

Definition 7: Let P = { $P\alpha_1$, $P\alpha_2$..... $P\alpha_\delta$}, where

- P – denotes the parent node
- $P\alpha_k$ – denotes the child nodes of P and $k \in \{1, 2, \ldots N\}$
- $\alpha = \{i\}$ – where i denotes the inbound child
- $\delta \in N$ – where N denotes natural numbers {0, 1, …}
- inbound child – $\lim_{\delta \to 26} P = \{P_1^{(i)}, P_2^{(i)} \ldots\ldots P_{26}^{(i)}\}$.

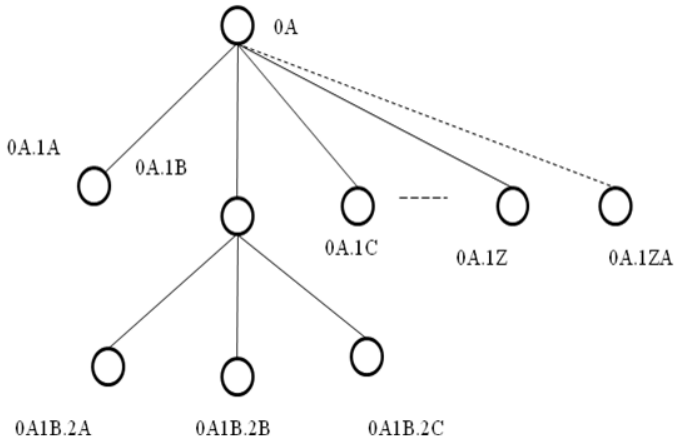
Definition 8: Let P = $\{P\alpha_{(\delta+1)} \ldots P\alpha_N\}$, where

- P – denotes the Parent node
- $P\alpha_k$ – denotes the child nodes of P and $k \in \{1,2,\ldots N\}$
- $\alpha = \{o\}$ – where o denotes the outbound child
- $\delta \in N$ – where N denotes natural numbers {0, 1, …}
- outbound child – $\lim_{\delta \to 27} P = \{P_{27}^{(o)}, P_{28}^{(o)} \ldots\ldots P_N^{(o)}\}$

The first 26 children of a parent node are called inbound children and they are labelled with single alphabet character between [A–Z]. When number of children of a given parent node P exceeds 26, then the first outbound child (defined in Definition 8) is labelled by appending the character 'A' to its previous sibling label and so on.

For example if last child of a parent node 'P' in figure is labelled as '0A.1Z', then the outbound child of 'P' is labelled as '0A.1ZA' and the next child will be labelled as '0A.1ZB' and so on.

Figure 5 LPLX labelling scheme special case label assignment

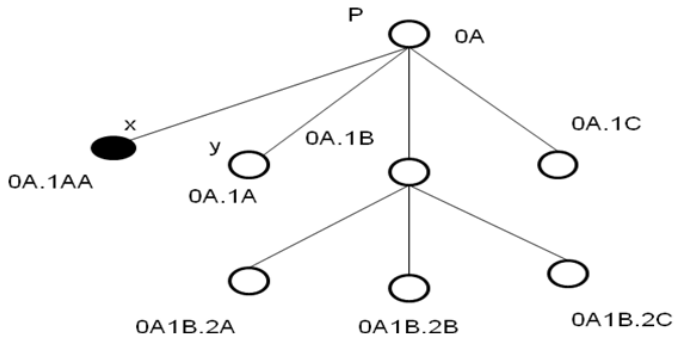


Example 3: Let LPLX label L1 = 0A.1Z, means, the $\text{Prefix}_n(x)$ = 0A, $\text{Level}_n(y)$ = 1, $\text{Selfcode}_n(z)$ = Z. Suppose we want to add a node 'L2' after the node 'L1', L2 is assigned a label as '0A.1ZA'. Suppose a node 'L3' is to be added after the node 'L2' , then 'L3' is assigned a label as '0A.1ZB' and so on.

3.2 Dynamic labelling step

XML documents on the internet are subjected to numerous updates. The updating can change the labels, the order and the structural information of the existing nodes. For example the labelling scheme (Dietz, 1982; Duong and Zhang, 2005) insertion of a new node may cause the re-computation of existing node's label. This is an expensive process. The LPLX scheme overcomes this problem. The dynamic labelling algorithm of the proposed scheme explains how the newly inserted nodes are assigned a unique label without re-computing the labels of the existing nodes. LPLX address three kinds of insertions in an XML tree.

Figure 6 Insert before



Case 1: Insert before: inserting *Before* inserts a new node before a leftmost child of a parent node '*P*'. Suppose node *x* is the node inserted before the node *y* which is the leftmost child of the parent *P*. The label for *x* is computed by concatenating the alphabet

'A' to the label of its following sibling node *y*. Figure 6 shows the computation of the label.

Example 4: Let y = '0A.1A', and we want to add a new node '*x*', left of the node 'y', then, the label for the new node x = '0A.1AA'. Suppose we want to add a new node x1 to the left of the node x, then the new node x_1 will have the label as '0A.1AAA' and so on.

Case 2: Insert between: this inserts a new node between two nodes of the same parent. Suppose a node *x* is to be inserted between two nodes *y* and *z* of the same parent *P*. After insertion y will be the preceding sibling and *z* will be the following- sibling of *x*. The selfcode for *x* will be computed by concatenating the selfcode of the preceding sibling *y* and following-sibling *z*. For example, Figure 7 illustrates this concept.

Figure 7 Insert between

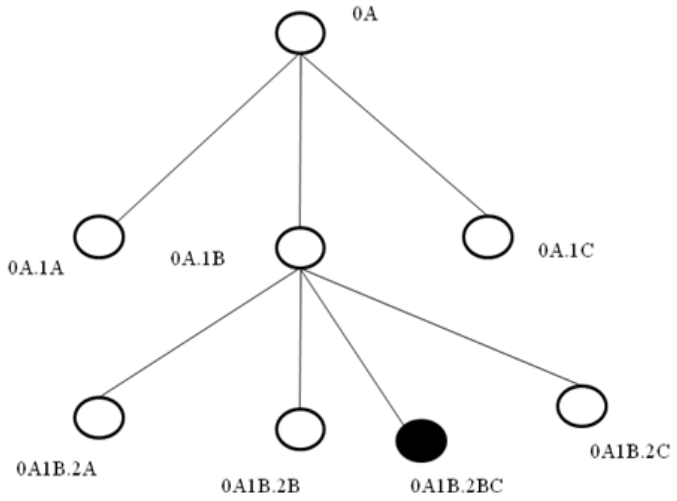
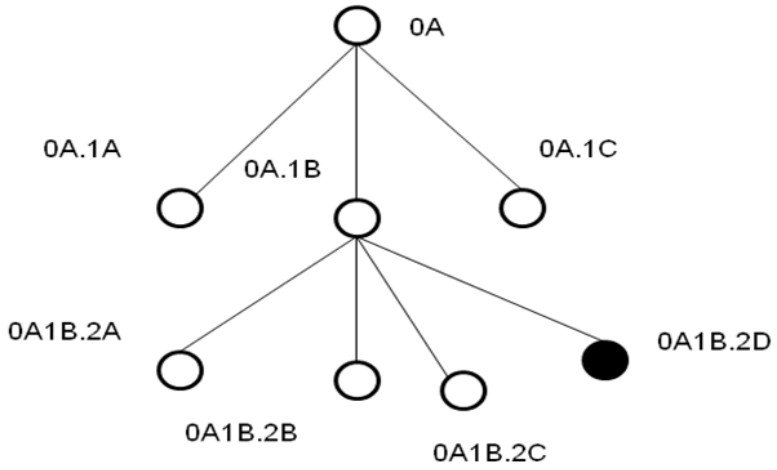


Figure 8 Insert after



Example 5: Let x = '0A1B.2B' and y = '0A1B.2C', and we want to add a new node '*z*', between the node '*x*' and '*y'*, then the label for the new node z = '0A1B.2BC'.

Case 3: Insert after: inserting a new node after the rightmost child of a parent node P. Suppose x is the node to be inserted after the node y of the parent node P as shown in Figure 8. The label for node x is obtained by lexicographically incrementing the selfcode of the preceding sibling y. If the selfcode of y is 'Z', then the alphabet 'A' is concatenated along with the selfcode of y to obtain the selfcode for x.

Example 6: Let x = '0A1B.2C' and, we want to add a new node 'y', after the node 'x', then label for the new node 'y' = '0A1B.2D'.

The pseudo codes for the above three cases are given below. The insertBefore(y) takes parameter 'y'. The parameter denotes the leftmost node of a parent node 'P'. The algorithm will insert the new node as the previous sibling of node y. The selfcode of the new node, say 'x', will be the letter 'A' concatenated with selfcode of node 'y' as shown in Figure 6. The dark circle in the figure denotes the newly inserted node 'x' and the corresponding label of that.

3.2.1 Algorithm for inserting a new node before a left most child of a parent

Algorithm 2 To insert a node x before a leftmost child y

```
Input: x: x denotes leftmost child of a parent node P in the XML Tree.
Output: a label for node x.
Method: This algorithm labels the new node x by concatenating the parent label P with the level
of y and a alphabet "A"
InsertBefore(x, y)
{if (LMchild(y, P)
{
Prefix_x = label (P);
Level_x = level(y);
      Selfcod_x = "A" + selfcode_y;
Label_x = prefix_x + level_x + selfcode_x;
Return labelx;
}
```

The InsertBetweenNodeLabel(y, z) takes two parameters y and z. The first parameter denotes the left sibling of the new node to be inserted and second parameter denotes the following sibling of the newly inserted node as shown in Figure 7. The new node, say 'x', is assigned a selfcode by concatenating the selfcode of node 'y' and 'z'. For example the newly inserted node in the Figure 7 gets the selfcode as 'BC', which is obtained by concatenating the self labels of preceding and following siblings of the node.

Algorithm 3 Inserting a node in between the two nodes of the same parent node P

```
Input: Y, z: 'y' and 'z' denotes the previous and following siblings of newly inserted node 'x' in
the XML tree.
Output: a label for node x.
InsertBetween(x, y)
    {
    Prefix_x = prefix_y;
    Level_x = level(y);
    Selfcode_x = selfcode_y + selfcode_z;
        Label_x = prefix_x + level_x + selfcode_x;
            Return label_x;
    }
```

The *InsertAfterNode(x, y)* takes two parameters. The first parameter denotes the node to be inserted and second parameter denotes the node after which new node x will be inserted. The selfcode of the new node x obtained by lexicographically incrementing the selfcode of the node y as shown in figure. For example, the newly inserted node in Figure 8 is denoted as a dark circle and it is obtaining a label as 'D' which is lexicographically incremented selfcode of its previous sibling selfcode 'C'.

Algorithm 4 Inserting a node after the rightmost child of a parent node

```
Input: y: 'y' denotes the right most child of a parent node 'P' in the XML tree.
Output: a label for node x.
Method: This algorithm labels the new node 'x' by checking the last character of the selfcode of
node 'y'. If the last character is 'Z' then it concatenates a letter 'A' to it and assigns it to the new
node. Otherwise the last character is replaced by the next character in the lexicographic order.
InsertAfter(x, y)
{
if (RMchild(y,P))
    prefix_x = prefix(y);
        level_x = level(y);
    Flag = checkselfLabel(y);
        If (flag) then
            {selfcode_x = selfcode_y + 'A'
        Else
    {ch = extractLastChar(selfcode_y); ch++;
prefix_x = replaceLastChar(selfcode_y, ch);
}
Label_x = prefix_x + level_x + selfcode_x;
Retrun label_x;
}
```

In LPLX system, whenever a node is inserted between two nodes, say x_1 and x_2, the new node is assigned a selfcode as concatenating the selfcode of x_1 and x_2 which will be always a new string and also satisfies the lexicographic order.

Theorem: Let X is the left character string and Y is the right character string which is lexicographically ordered. Then inserting a new string, say X_m, in between x_1 and x_2 by concatenating X and Y is also follow the lexicographic order i.e., $X < X_M < Y$.

Proof: Let string $X = x_1\ x_2\ x_3 \ldots x_m$ is the selfcode of node x_1 and string $Y = y_1\ y_2 \ldots y_n$ is the selfcode of node x_2, then by algorithm defined in Algorithm 3 the new node is assigned a selfcode by concatenating strings X and Y as $X_M = x_1\ x_2 \ldots\ldots x_m\ y_1\ y_2 \ldots\ldots y_n$ which follows the lexicographic order such as $X < X_m < Y$.

Case(a) $X_1 < X_M$

X_1 has a prefix in X_m and length(X_m) > length(x_1). Hence by Definition 6, the character string $X1 < X_m$.

Case(b) $X_M < Y$

Let l = lcp(X_M, Y); Since X_M is the concatenation of two strings X and Y the length of X_M > length of Y. length(X_M) > l and length(Y) > l and XM[l] < Y[l], where l denotes the least common prefix of the strings X_M and Y. then by Definition 6 the character string $X_M < Y$ in lexicographic order.

From case (a) and (b) it is clear that the newly generated string x_m lies in between the two strings X and Y in lexicographic order.

Example 7: Suppose we have two selfcodes for the node X and Y as ‘A’ and ‘B’ respectively. The node to be inserted between X and Y is denoted as X_m. The node Xm is assigned a selfcode as ‘AB’ as defined by algorithm 34. Then by the Theorem 1, we can verify that ‘AB’ lies in between ‘A’ and ‘B’ in lexicographic order. The concatenation always generates the new string and hence it avoids the collision.

The dynamic updating is performed without any collisions. The next section discusses how the structural information is captured by the proposed labelling scheme.

3.3 Structural summary

This section discusses how the proposed system identifies the structural relationship among the nodes in XML tree. The structural relationships between any two nodes play an important role in the query processing. The labelling scheme determines the structural relationships between the nodes such as parent-child, ancestor-descendant, and sibling by comparing their labels. Based on the labels, the query can be processed without accessing the entire XML document which improves the performance of query processing. The proposed system makes use of three rules, Rule 1, 2 and 3, to determine the structural summary between any nodes in the XML tree. It takes constant time to determine these relationships. Rule 1 describes the determination of ancestor-descendant relationship. Rule 2 describes the determination of the parent-child relationships and Rule 3 describes the determination of the sibling relationship.

3.3.1 Method of identifying structural summary

The proposed system label provides the structural information like parent-child, ancestor-descendant and sibling directly by looking at the labels only. The following section discusses the rules to identify the structural summary.

Let L1 and L2 denote the labels for the nodes A and B as $L_1(A,x,y,z) = prefix_A(x)$. $level_A(y)selfcode_A(z)$ and $L_2(B,x,y,z) = prefix_B(x)$. $level_B(y)selfcode_B(z)$.

The following rules identify different structural relationships between the nodes in the XML Tree.

Rule 1 Ancestor-descendant relationship: A-D (A, B)

The node A is ancestor (B) iff $label_A = substr(prefix_B)$.

For example consider the following cases:

Case 1 Let A and B are two nodes with labels LabelA = *0A* and $Label_B$ = *0A1A.2B respectively*. Then node 'A' is ancestor of node 'B' since *0A* is a substr (*0A1A*).

Case 2 Let A and B are two nodes with labels $Label_A$ = 0A and $Label_B$ = 0A1B.2B. Then node 'A' is ancestor of node 'B' since $Label_A$ 0A is a substr(0A1B).

Rule 2 Parent-child relationship: P-C(A, B)

The node A is parent (B) if $prefix_B = label_A$ and $level_B$-$level_A = 1$

For example, Let $Label_A$ = 0A1B and $Label_B$ = 0A1B.2B. Then A is parent (B) since $label_A = prefix_B$ i.e., 0A1B is the prefix of 0A1B.2B and $level_B$-$level_A = 1$.

Rule 3 Sibling relationship: sibling (A, B)

The node A is sibling of B iff $prefix_{A-} = prefix_B$.

For example, Let $Label_A$ = 0A1A.2B and $Label_B$ = 0A1A.2C.
$Prefix_A = prefix_B$ = 0A1A. Hence nodes A and B are siblings to each other.

3.4 Computation of size of the self-code

LPLX 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 node i.e., 27^{th} 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 next 26 children will get a selfcode of two characters and so on. 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 selfcode of one character length, the next 26 children gets a selfcode of two characters of

length and the remaining nodes gets a selfcode of three characters of length. Hence the total character required is 120.

In general, the total storage requirement for selfcode at a given level for N number of nodes can be obtained using the formula as follows:

$$\text{SizeOf(Selfcode(z))}=\sum_{w=1}^{\frac{N}{26}} w*26+N\%*\left(\frac{N}{26}+1\right) \quad (1)$$

In LPLX scheme, the size of the label is increasing as the depth increasing. This will handle by the compaction techniques which will be done as a future work.

3.5 Label generation time and space for varying depths

For a constant number of nodes with varying depths, the labelling time taken by the algorithm is proportional to the number of nodes. For example, assume each visit of node in the tree takes one unit of cost for visiting the node and one unit of cost for each backtracking.

Figure 9 XML tree with 10 nodes and depth 3

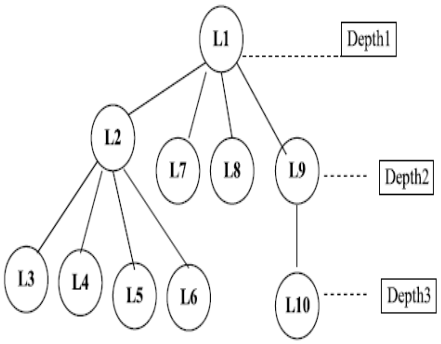


Figure 10 XML tree with 10 nodes and depth 5

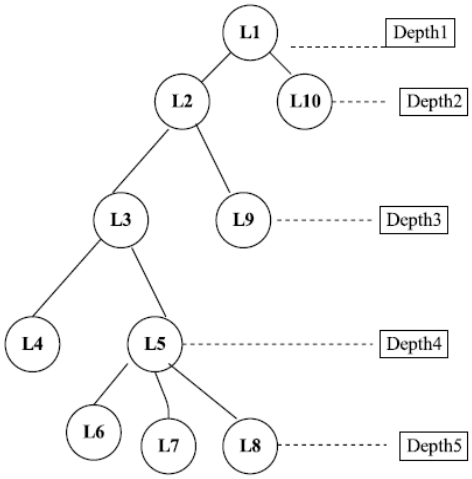


Figure 11 Cost matrix for XML tree ten nodes with (a) depth 3 and (b) depth 5

X \ *Y*	*1*	*2*	*3*
L1	1	0	0
L2	1	0	0
L3	1	0	0
L4	1	1	0
L5	1	1	0
L6	1	1	0
L7	1	1	1
L8	1	1	0
L9	1	1	0
L10	1	0	0

(a)

X \ Y	1	2	3	4	5
L1	1	0	0	0	0
L2	1	0	0	0	0
L3	1	0	0	0	0
L4	1	0	0	0	0
L5	1	1	0	0	0
L6	1	0	0	0	0
L7	1	1	0	0	0
L8	1	1	0	0	0
L9	1	1	1	1	0
L10	1	1	1	0	0

(b)

Figures 11(a) and 11(b) represents the cost table for the above XML tree of ten nodes with varying depths 3 and 5. X denotes the total number of nodes and each node is designated as L1, L2 … L10. Y denotes the depth. The rows represents the cost of visiting (labelling) a node. It is the sum of visiting cost and back tracking cost. We assume that each node takes cost of one unit on visiting that node. The total cost is calculated using the formula defined in equation (2).

The total units of cost consumed for traversing entire tree is formulated as

$$\text{Total Cost} = \sum_{x=1,y=1}^{x=X} \text{Cxy} + \sum_{x=1,y=2}^{x=X,y=Y} \text{Bxy}$$

where C_{xy} denotes the cost of visiting a node where x varies from 1 to X and y varies from 1 to Y which is considered constant time 1 of 1 unit. B_{xy} is the backtracking cost. The figure with number of nodes 10 with varying depths 3 and 5.

In Figure 11(a), cost of visiting (labelling) of node L7 = (1 + 1 + 1) = 3, where first 1 denotes the visiting cost and second and third 1's denotes the backtracking cost. L9 = (1 + 1) = 2, where the first 1 denotes the visiting cost and the second one denotes the backtracking cost. The total cost = 17 units [by applying the formula in equation (2)]. In Figure 11(b), cost of visiting (labelling) of node L7 = (1 + 1) = 2 and L9 = (1 + 1 + 1 + 1) = 4. The total cost = 18 units. From this it is clear that for constant number of nodes in varying depth the labelling time is going to be approximately same. This is also validated by conducting experiments on the dataset specified in Table 1 and the result is shown in Figure 13. This experiment is also conducted with existing labelling scheme LSDX and the result is compared and shown in Figure 13(a) and 13(b).

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. Two set of experiments were conducted on our proposed system. The first set of experiments was conducted to measure the storage required and time required to generate LPLX labels for varying depths. This experimentation is also performed on the existing labelling scheme LSDX and the results are compared.

Experiement I: label generation time and space for varying depths. We have performed this experiment on our own data set with varying number of nodes and depths. We have tested the above experiments for the datasets with varying number of nodes and depths as described in Table 1. The XML test data set consist of documents with 1K nodes (elements) with varying depths like 3, 4, 5, 7, 10, 12, 15 and 20. 2K nodes with varying depths 3, 4, 5, 7, 10, 12, 15 and 20. Similarly 3K, 4K and 5K nodes with varying depths 3, 4, 5, 7, 10, 12, 15 and 20 is created.

Table 1 Characteristics of test dataset

Dataset	*#Nodes (in K)*	*Depth*
D1	1	3, 4, 5, 7, 10, 12, 15, 20
D2	2	3, 4, 5, 7, 10, 12, 15, 20
D3	3	3, 4, 5, 7, 10, 12, 15, 20
D4	4	3, 4, 5, 7, 10, 12, 15, 20
D5	5	3, 4, 5, 7, 10, 12, 15, 20

As a first set of experiment we computed the storage required for the LPLX labels of each dataset specified in Table 1. First experiment for our LPLX labelling system was carried out to find the storage required for the labels generated for the nodes in various depths remains approximately the same. The depth we are considered here is depth 3, 4, 5, 7, 10, 11, 12, 15, 20. We used SAX Parser to parse different depths of XML documents with number of nodes 1 KB, 2 KB, 3 KB, 4 KB, 5 KB. The labels for each node is created and stored in a file. Then the total storage is calculated.

Figure 12 shows the storage required by the LPLX system. For example lowest depth three with varying nodes requires an average storage capacity 0.06 MB. The maximum storage required is 0.11 MB at 5K nodes. The minimum storage required is 0.01 MB at 1K nodes. The highest depth of twenty with varying number of nodes takes an average

storage capacity of 0.06 MB. The maximum storage required is 0.11 MB at 5K nodes. The minimum storage required is 0.019 MB at 1K nodes.

Figure 12 Storage of nodes in varying depths (see online version for colours)

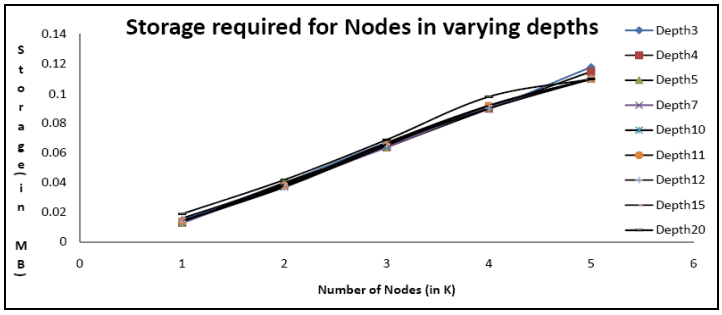
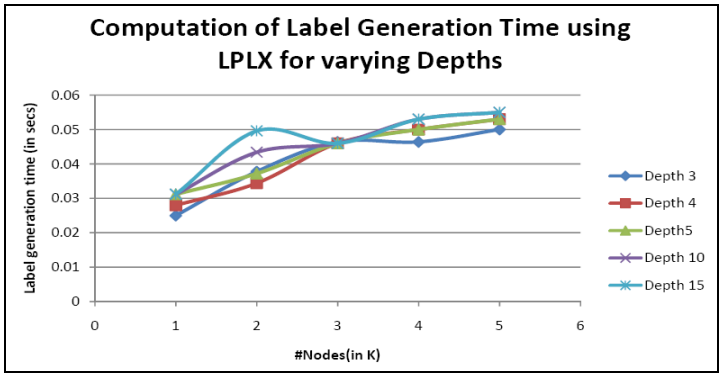
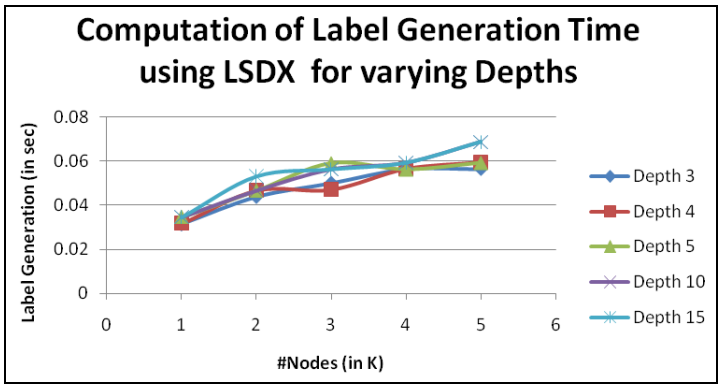


Figure 13 Computation of label generation time for varying depths using (a) LPLX scheme and (b) LSDX scheme (see online version for colours)



(a)



(b)

The second experiment is conducted to test how long it takes to generate labels. We have considered the same dataset as in Experiment 1. The performance is measured by running

the program five times on each data set and average is taken. The analysis done to verify that the time measured for generating labels of constant number of nodes in various depths is proportional to the number of nodes. The reason behind this is, even if the depth is varying the number of visited nodes remains same. This is also validated by conducting experiments on the dataset specified in Table 1 and the result is shown in Figure 13.

To perform this experiment, we have used Java and SAX parser to parse the XML documents and generated labels as and when the node is parsed, measured and notified the time taken for each XML documents in various depths. Figure 13(a) shows the time used to generate labels for varying depths.

We also performed the same experiment on the existing labelling scheme LSDX scheme and compared the results for the same dataset for depths 3, 4, 5, 10 and 15. The results are compared. The proposed scheme gives a better result than the existing one as shown in Figure 13(b).

Figure 13(a) shows the labelling time required with LPLX system with varying depths. Like in the first experiment, we have considered XML documents with nodes 1K, 2K, 3K, 4K and 5K with varying depths like 3, 4, 5, 7, 10, and 15. The time taken to generate labels is measured. For example the lowest depth three with varying nodes takes an average time of 0.036 secs to label the nodes, whereas LSDX scheme takes 0.04 secs for the same. The maximum time taken is 0.047 secs at 5K nodes and 0.05 for the LSDX scheme. The minimum time is taken by 1K nodes with 0.02secs and for the LSDX, the minimum time taken is by 1K nodes is 0.031secs. The highest depth of 15 with varying nodes takes an average labelling time of 0.043secs, whereas the LSDX is taking 0.054secs. The maximum time is taken to label nodes with 5K nodes is 0.05 secs. This data shows clearly that the above statement confirms time measured for generating labels of nodes in various depths remains approximately the same and compared to the existing scheme LSDX, the proposed scheme, LPLX give a better performance.

5 Conclusions and future work

In this paper we proposed a new method called LPLX for labelling nodes in the XML documents. The proposed scheme uniquely labels each node in the XML tree by traversing the document in depth first manner. It also handles the dynamic updates without relabeling the existing nodes. The analysis concludes that the storage required for the labels generated for the nodes in various depths remains almost the same with an overall average of 0.06 MB and the analysis also concludes that the overall average of labelling time is 0.039secs and the time measured for generating labels of nodes in various depths remains almost the same labelling time. This scheme also determines all the structural relationships efficiently by looking at the labels only. We have also measured the time required to generate the labels and storage size required for the nodes at each level. The result shows good amount of improvements in both computation time analysis.

References

Cohen, E., Kaplan, H. and Milo, T. (2002) 'Labeling dynamic XML trees', *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (SPDS)*, pp.271–281.

Dietz, P.F. (1982) 'Maintaing order in a linked list', *Proceedings of 14th Annual Symposium on Theory of Computing*, pp.122–127.

Duong, M. and Zhang, Y. (2005) 'LSDX: a new labelling scheme for dynamically updating XML data', *Proceedings of the 16th Australian Database Conference*, Vol. 39, pp.185–193.

Harder, T., Haustein, M., Mathis, C. and Wagner, M. (2007) 'Node labeling schemes for dynamic XML documents reconsidered', *Data and Knowledge Engineering*, Vol. 60, No. 1, pp.126–149.

Li, Q. and Moon, B. (2001) 'Indexing and querying XML data for regular path expressions', *VLDB 2001: Proceedings of the 27th International Conference on Very Large Databases*, Roma, Italy, pp.361–370.

Min, J-K., Lee, J. and Chung, C-W. (2009) 'An efficient encoding and labelling method of query processing and updating on dynamic XML data', *Journal of Systems and Software*, Vol. 82, No. 3, pp.503–515.

O'Neil, P.E., O'Neil, E.J., Pal, S., Cseri, I., Schaller, G. and Westbury, N. (2004) 'ORDPATHs: insert-friendly XML node labels', *ACM SIGMOD International Conference on Management of Data*, pp.903–908.

Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J. and Zhang, C. (2002) 'Storing and querying ordered XML using a relational database system', Proceedings of the *ACM SIGMOD International Conference on Management of Data*, pp.204–215.

W3C Recommendation (1999) *Extensible Markup Language (XML), XML Path Language (XPath), XQuery 1.0: An XML Query Language*, 16 November [online] http://www.w3.org/TR/ (accessed March 2006).

Xu, L., Ling, T.W. and Wu, H. (2012) 'Labeling dynamic XML documents: an order-centric approach', *IEEE Transactions Knowledge and Data Engineering*, Vol. 24, No. 1, pp.100–113.

Xu, L., Ling, T.W., Wu, H. and Bao, Z. (2009) 'DDE: from Dewey to a fully dynamic XML labeling scheme', *ACM SIGMOD International Conference on Management of Data*, pp.719–730.

Yu, X.J., Luo, D., Meng, X. and Lu, H. (2005) 'Dynamically updating XML data: numbering scheme revisited', *World Wide Web: Internet and Web Information System*, Vol. 8, No. 1, pp.5–26.

Zhang, C., Naughton, J.F., DeWitt, D.J., Luo, Q. and Lohman, G.M. (2001) 'On supporting containment queries in relational database management systems', *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pp.425–436.