ORDER BASED LABELING SCHEME FOR DYNAMIC XML (EXTENSIBLE MARKUP LANGUAGE) QUERY PROCESSING

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Beakal Gizachew ASSEFA

May 2012

İZMİR

We approve the thesis of **Beakal Gizachew ASSEFA**

_____

Assist. Prof. Dr. Belgin ERGENÇ
Supervisor

_____

Assist. Prof. Dr. Yalın BAŞTANLAR
Committee Member

_____

Prof. Dr. Oğuz DİKENELLİ
Committee Member

2 May 2012

_____

Prof. Dr. Sıtkı AYTAÇ
Head of the Department
of Computer Engineering

_____

Prof. Dr. R. Tuğrul SENGER
Dean of the Graduate School
of Engineering and Sciences

ACKNOWLEDGEMENTS

It is with immense gratitude that I acknowledge the support and help of my advisor Asst. Prof. Dr. Belgin ERGENÇ, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject. Working with her has been a great pleasure and honor.

I am indebted to my wife Meba Tadesse. Had it not been for her continuous support, encouragement and taking responsibilities on my behalf, my study in IZTECH would have remained a dream.

I also would like to thank all the professors who gave me courses while my stay in IZTECH. I truly believe the experience I acquired from them has changed me to a great extent. I am also grateful to Dr. Sinem Bezircilioğlu for her patience and time in giving me a valuable comment and suggestions on the format and vocabulary of the thesis.

Finally, I would like to thank my colleagues and fellow academic and administrative staff who had shown a great concern to me during my stay in Turkey and IZTECH in particular.

ABSTRACT

ORDER BASED LABELING SCHEME FOR DYNAMIC XML (EXTENSIBLE MARKUP LANGUAGE) QUERY PROCESSING

Need for robust and high performance XML database systems increased due to growing XML data produced by today's applications. Like indexes in relational databases, XML labeling is the key to XML querying. Assigning unique labels to nodes of a dynamic XML tree in which the labels encode all structural relationships between the nodes is a challenging problem. Early labeling schemes designed for static XML document generate short labels; however, their performance degrades in update intensive environments due to the need for relabeling. On the other hand, dynamic labeling schemes achieve dynamicity at the cost of large label size or complexity which results in poor query performance.

This thesis presents OrderBased labeling scheme which is dynamic, simple and compact yet able to identify structural relationships among nodes. A set of performance tests show promising labeling, querying, update performance and optimum label size.

ÖZET

GENİŞLETİLEBİLİR İŞARETLEME DİLİNDE(XML) SORGU İŞLEMİ İÇİN SEVİYE TABANLI ETİKETLEME YAKLAŞIMI

Günümüz uygulamalarınca üretilen XML verisinin çoğalması, yüksek başarımlı ve sağlam XML veritabanlarına olan gereksinimi arttırmıştır. İlişkisel veritabanlarındaki indeksleme gibi, XML etiketleme de XML sorgulamanın anahtar bileşenidir. XML ağacının düğümlerinin her biri için düğümler arasındaki ilişkileri ifade edebilen ayrı etiketler oluşturmak zorlukları olan problemdir. Önceleri, etiketleme yaklaşımları kısa etiketler üretebildiği halde devingen ortamlarda yeniden etiket oluşturmak gerektirdiğinden düşük başarım göstermekteydiler. Devingen ortamları destekleyen etiketleme yaklaşımları ise uzun etiketler ve üretim karmaşıklığı nedeniyle zayıf başarım gösterebilmektedirler.

Bu tezde, devingen, basit ve kısa olduğu halde tüm düğümler arası ilşkileri ifade eden tek etiketler üretebilen OrderBased etiketleme yaklaşımı sunulmaktadır. Bir dizi başarım değerlendirme testi, bu yaklaşımın etiketleme, sorgulama ve güncelleme de kayda değer sonuçları en iyi etiket uzunluğu ile verebildiğini göstermiştir.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

The fact that XML has become the standard format for structuring, storing, and transmitting information has attracted many researchers in the area of XML query processing. XPath and XQuery are languages for retrieving both structural and full text search queries from XML documents [1 and 2]. XML labeling is the basis for structural query processing where the idea is to assign unique labels to the nodes of an XML document that form a tree structure. Label of each node is formed in a way to convey the position of the node in XML tree and its relationship with neighbor nodes. These relationships are Ancestor-Descendent (AD), Parent-Child (PC), Sibling and Ordering [2].Figure 1.1 shows an example XML document whereas Figure 1.2 shows a tree representation of the XML document in Figure 1.1.

```
<XML>
  <bookstore>
        <book category="COOKING">
                <title lang="en">Everyday Italian</title>

                <author> Giada De Laurentiis</author>
                <year>2005</year>
                <price>30.00</price>
        </book>
        <book category="CHILDREN">
                 <title lang="en">Harry Potter</title>
                 <author>J K. Rowling</author>
                 <year>2005</year>
                 <price>29.99</price>
        </book>
        <book category="WEB">
                <title lang="en">Learning XML</title>
                <author>Erik T. Ray</author>
                 <year>2003</year>
                <price>39.95</price>
        </book>
  </bookstore>
</XML>
```

Figure 1.1. An example XML document

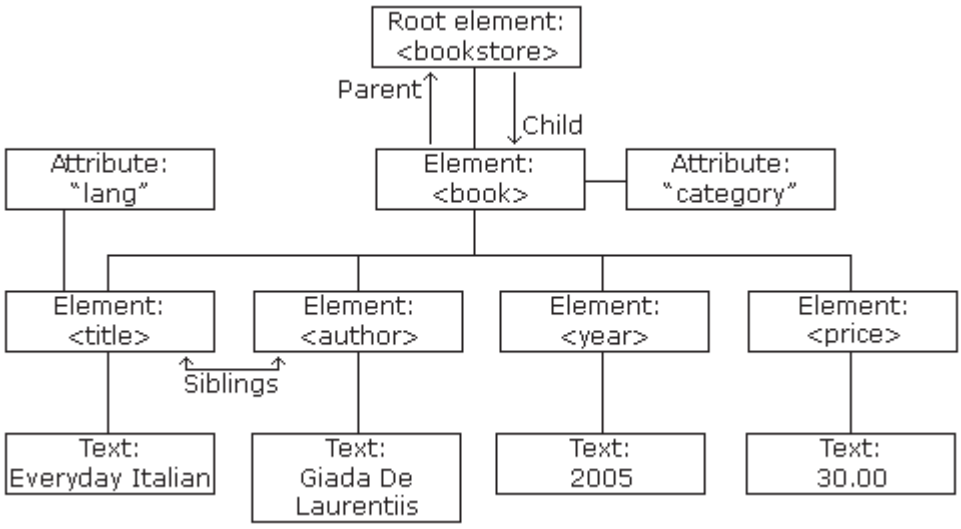


Figure 1.2. A tree representation of the XML document in Figure 1.1

There are basically two approaches to store XML document. The first one is to shred the XML document to some database model. The XML document is mapped to the destination data model example, relational, object oriented, object relational, and hierarchical. The second approach is to use native XML Database (NXD) [39, 40, 41, 42, and 43]. Native XML database (NXD) is described as a database that has an XML document as its fundamental unit of storage and defines a model for an XML document, as opposed to the data in that document (its contents). It represents logical XML document model and stores and manipulates documents according to that model. Although XML labeling is widely used in NXD, it also plays a role in the shredding process.

Labeling schemes can be grouped under four main categories namely; Range based, Prefix based, Multiplication based, and Vector based. Range based labeling schemes label nodes by giving start and end position which indicate the range of labels of nodes in sub trees [3, 4, 5 and 23]. Prefix based labeling schemes concatenate the label- of ancestors in each label using a delimiter [6, 7, 8, 9 and 10]. Multiplication based labeling schemes use multiplication of atomic numbers to label the nodes of an XML document [16 and 19]. Vector based labeling schemes are based on a mathematical concept of vector orders [17, 18 and 24]. Recently, it is common to see a hybrid labeling schemes which combine the advantages of two or more approaches [25 and 26].

A good labeling scheme should be concise in terms of size, efficient with regard to labeling and querying time, persistent in assuring unique labels, dynamic in that it should avoid relabeling of nodes in an update intensive environment, and be able to directly identify all structural relationships. Last but not least, a good labeling scheme should be conceptually easy to understand and simple to implement. Finding a labeling scheme fulfilling those properties is a challenging task. Generally speaking, labeling schemes that generate small size labels either do not provide sufficient information to identify all structural relationships among nodes or they are not dynamic [3, 4 and 5]. On the other hand, labeling schemes that are dynamic need more storage which results in a decrease of query performance [6, 7, 8, 9, 10 and 20] or are not persistent in assuring unique labels [9 and 10].

With the increasing popularity of XML we see commercial software developers engaged in accommodating the demand of efficient XML querying. DB2 supports native format and uses Dewey encoding to assign a unique identifier (NID) that gives the node both a logical and physical addressability that can be used for indexing and query evaluation. NID provides efficient navigation of the XML document, and is also beneficial for evaluating XQuery statements [42 and 43]. ORDPATH is a hierarchical labeling scheme used in the internal implementation of the XML data type in SQL Server 2005 [41]. It's meant to provide optimized representation of hierarchies, simplify insertion of nodes at arbitrary locations in a tree, and also provide document order. In SQL Server 2008, there are additional uses of ORDPATH. There is a new system data type HierarchyID that uses ORDPATH in its implementation. This allows simply hierarchies to be represented as relational column and provides methods that optimize common structural relationship queries [41]. Moreover, many NDXs like TIMBER have been developed for research purposes [40]. TIMBER uses Containment labeling scheme for structural query support.

This thesis presents a novel dynamic labeling scheme based on a combination of letters and numbers called OrderBased. Each label contains level, order of the node in the level and the order of its parent. Keeping the label of the existing nodes unaltered in case of updates and guaranteeing optimized label size are the main strengths of this approach. Label size and dynamicity is achieved without sacrificing simplicity in terms of implementation.

In performance evaluation, OrderBased labeling scheme is compared with LSDX and Com-D [9 and 10]. These labeling schemes are chosen because using combinations of letter and numbers, including the level information of a node in every label, and avoiding relabeling when update occurs are the common features and design goals of the three schemes. Storage requirement, labeling time, querying time, and update performance are measured. Results show that OrderBased labeling scheme is smaller in size and faster in labeling and query processing than LSDX labeling scheme. Although Com-D labeling scheme needs slightly less storage than OrderBased, its labeling, querying, and update performance is the least efficient due to compression and decompression overhead cost.

The thesis is organized as follows: Chapter 2 presents a thorough discussion of related work, Chapter 3 presents OrderBased labeling scheme. Chapter 4 illustrates storage requirements, labeling time, querying, and update performance of OrderBased labeling scheme in comparison with LSDX and Com-D labeling schemes. Finally, Chapter 5 concludes the thesis and gives a glimpse of future works.

CHAPTER 2

RELATED WORK

Labeling schemes can be defined as a systematic way of assigning values or labels to the nodes of an XML tree in order to speed up querying. The problem of finding a labeling scheme that generates concise, persistent labels, supporting updates without the need of relabeling, and ease of understanding and implementation dates back to 1982 [3]. In the pursuit of solving the labeling scheme problem, a number of approaches have been proposed. These labeling approaches can be grouped in four major categories: Range based, Prefix based, Multiplication based and Vector based.

2.1. Range Based Labeling Schemes

Range based labels for a node *X* has a general form of <*start-position*, *end-position*>, where *start-position* and *end-position* are numbers such that for all nodes *Y* in the sub tree of *X*, *start_position(Y)*> *start-position(X)* and *end-position (Y)* < *end-position(X).*

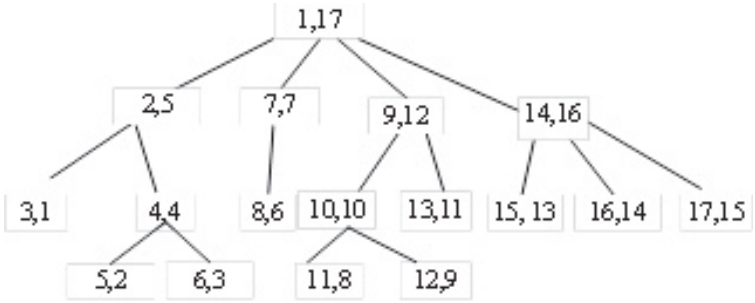


Figure 2.1. Traversal Order Based labeling scheme

The first XML labeling scheme introduced is ***Traversal Order Based*** labeling scheme [3]. This scheme uses numbers and it is based on preorder- postorder traversal notation of a tree. Traversal order based labels has a form of <***pre, post***>. A given node *Y* is a descendant of node *X* Iff *Y.pre >X.pre and Y.post < X.post*.

The two disadvantages of this labeling scheme are first the labels do not contain sufficient information to determine Parent-Child, and Sibling-Order relationships. Secondly, this scheme is not efficient for dynamic XML documents. For example in Figure 2.1 , <9, 12> is the ancestor or <11, 8>, and <12,9>. However, there is not enough information to identify whether <1, 17> is a parent of <7, 7> since the only relationship identified by this labeling scheme is ancestor descendant relationship. Moreover, inserting a node or a sub tree at any point causes global relabeling.

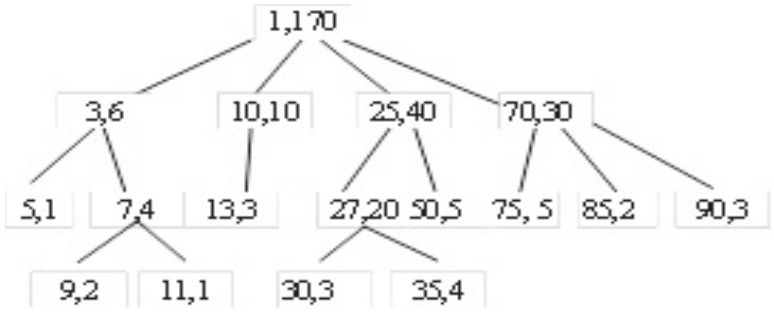


Figure 2.2. Extended Preorder Traversal labeling scheme

Li and Moon proposed an ***Extended Preorder Traversal*** labeling scheme to improve the second drawback of traversal order labeling scheme [4]. This labeling scheme is based on the notation of extended preorder traversal to accommodate future insertions gracefully. Each label is of the form <*preorder, size*>, such that *preorder* is the preorder of the node and *size* is an arbitrary integer greater than the total number of descendants of the node as shown in Figure 2.2. So as to make insertion without relabeling, the size should be reasonably large.

However, assigning a value for size is not straightforward. The approach used is based on an anticipation of the maximum number of nodes to be added and allocating large value to the size parameter. Nonetheless, skewed insertions eventually fill the reserved space. Even if size is large enough, when the reserved spaces are all used up, re-computing becomes inevitable.

In ***Containment*** labeling scheme, every node is assigned three values: “start”, “end” and “level” [29]. For any two nodes *u* and *v*, *u* is an ancestor of *v* iff *u:start* < *v:start* and *v:end* < *u:end*. Node u is the parent of node *v* iff *u* is an ancestor of *v* and *v:level* - *u:level* = 1. Although containment scheme is efficient for determining A-D and P-C relationships, the insertion of a node *n* will lead to re-labeling of all the ancestor

nodes of *n* and all the nodes after *n* in document order. To solve the re-labeling problem, float-point values for the start and end of the intervals was suggested [30]. However, in practice, float-point is represented physically with a fixed number of bits. As a result, after the possible float number is used, relabeling is required. An example of containment labeling scheme is shown in Figure 2.3.

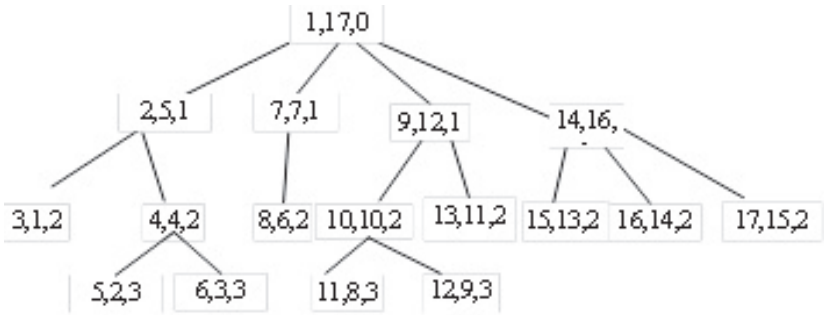


Figure 2.3. Containment labeling scheme

P-Containment labeling scheme is a modified form of Containment labeling scheme [32]. In contrast to the traditional Containment labeling scheme, it stores the start value of the parent node instead of the level information. Hence a P- Containment label is a triple with <start, end, parent_start>. The motivation for including the parent start rather than the level information is to enhance the performance of Parent - Child relationship queries.

Given nodes *u* and *v*, node u is a parent of node v iff the "parent_start" value of node v is equal to the "start" value of node u based on P-Containment. For two different nodes *u* and *v* that are not the root of the XML tree, node u is a sibling of node v iff the "parent_start" value of node u is equal to the "parent_start" value of node v based on P-Containment. Figure 2.4 illustrates an example of a P-Containment encoding.

Figure 2.4. P- Containment labeling scheme

It is reported that determining the parent-child relationship is faster, and determining the sibling relationship is much faster. The Ancestor-Descendant and Sibling, Order relationship determinations based on P-Containment remain the same as the traditional containment encoding [32].

In order to fix the limitations of the Preorder and Extended Preorder Labeling and containment labeling schemes, ***Dynamic Interval Based*** labeling scheme was introduced [5]. It is based on the concept of nested and inverted nested tree. The labeling scheme treats newly inserted nodes as a sub tree and only one number will be used from the reserved numbers.

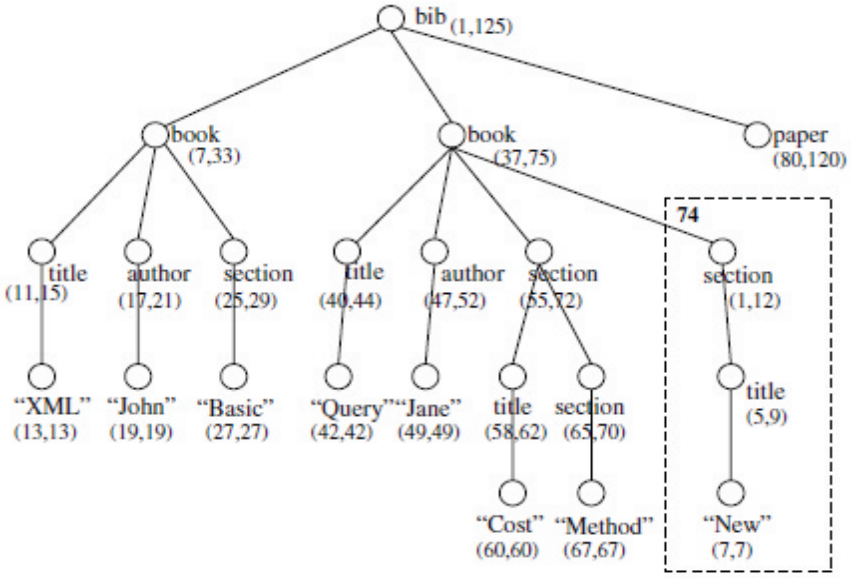


Figure 2.5. Dynamic Interval Based labeling scheme [4]

As illustrated in Figure 2.5, to insert another node after the node with label (55, 72), first it checks if there is free space in the parent node. The label of the parent node is (37, 75). Because numbers 73 and 74 are between 72 and 75, the scheme picks 74 arbitrarily. The new inserted sub tree is labeled starting from 1 as a new tree; however, 74 are regarded as the prefix for all nodes in the inserted sub tree. It was argued that that Dynamic Interval Based labeling scheme is a hybrid labeling scheme with attempt of combining the advantages of prefix and range based approaches [35].

The fact that algorithm uses a single number and treats the inserted sub tree as one tree clearly reduces the usage of the reserved numbers and more nodes can be inserted at a time. The other advantage of this approach is that even if the reserved

spaces are consumed, this approach localizes the relabeling to a great extent by only affecting the parent of the node. However, it does not fully support dynamic XML document since after the reserved space is used up relabeling the whole tree in the worst case becomes unavoidable.

Sector Based labeling scheme, ***SL*** in short is also a range based labeling scheme, however sectors are used instead of intervals and mathematical formulae are presented to determine ancestor-descendant and document-order relationships between label pairs. The two components of a SL labels are the radius of the node from its parent node and its offset value [23]. The idea is that all nodes enclosed in the start and end angle are children of a given node. The radius of the sector is stored on its logarithmic form rather than the radius itself so as to optimize the size of labels. However, it is stated in [28] that in SL, insertion and deletion operations do not adapt gracefully without the need of relabeling.

2.2. Prefix- Based Labeling Schemes

In Prefix based labeling schemes, node X is an ancestor of node Y if the label of node X is the prefix of node Y. The main advantage of prefix based labeling approach is that all structural relationships can be determined by just looking at the labels. The main critics about prefix based labeling schemes is their impractically large storage requirement.

Simple Prefix labeling scheme is an example of a prefix labeling scheme which uses 0's and 1's to label the nodes of an XML tree [6]. The root node is labeled with empty string. The first, second, third , and the n^{th} child of the root node are labeled as '0', '10', '110' ,' (n-1) 1's 0' respectively. For the rest of the nodes of the XML tree, each label concatenates the label of its parent and its self-position using 0's and 1's. Figure 2.6 shows an example of a simple prefix labeling scheme.

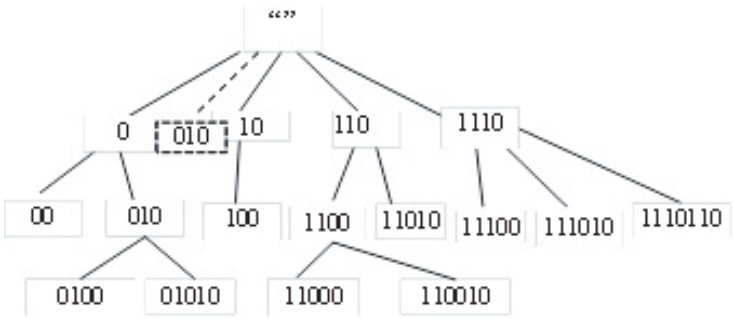


Figure 2.6. Simple Prefix labeling scheme

The advantages of Simple Prefix labeling scheme are the determination of all structural relationships and support of updates without the need of relabeling. For example, "11010" and "1101010" can be generated between "110" and "1110". As shown in Figure 2.6, to insert a node between the nodes with labels '0' and '10', it generates a new label with '010'. However, '010' had already been given to another node. Hence, Simple Prefix labeling scheme does not guarantee uniqueness after insertions.

The authors of the Simple Prefix labeling scheme proposed two approaches for assigning the bits [6]. The first approach has a label growth rate of one-bit such that the positional identifier of the first child of a given node is 0, of the second child is 10, of the third child is 110 and of the n^{th} child is (n-1) ones with a 0 concatenated at the end. The second approach has a double-bit label growth rate. Though the author proposed the use of a clue to minimize the size, the clues are too strict to be satisfied. As a result, both approaches tend to produce significantly large label sizes.

Despite the support of updates without the need of relabeling and the capacity of determining all structural relationships, Simple Prefix labeling scheme generates labels with impractically large size. Moreover, it is not persistent since insertion may result in collision.

Dewey ID is a prefix labeling scheme adapted from the Dewey decimal classification system [31] for the organization of library collections [7]. In Dewey ID, the positional identifier of the n^{th} child is assigned the integer n and this is concatenated to the parent's label and a delimiter.

The main advantage of Dewey ID is that all structural relationships can be identified by just looking at the labels. Insertions and deletions at the right most nodes of a sub tree are accommodated gracefully without relabeling. However, update in the middle or on the left side of a tree /sub tree triggers following siblings relabeling, this

relabeling propagates to their descendants also. The two limitations of Dewey ID are its huge size and inability to fully adapt all kinds of updates gracefully regardless of the position of update. An example of Dewey ID labeling scheme is illustrated on Figure 2.7.

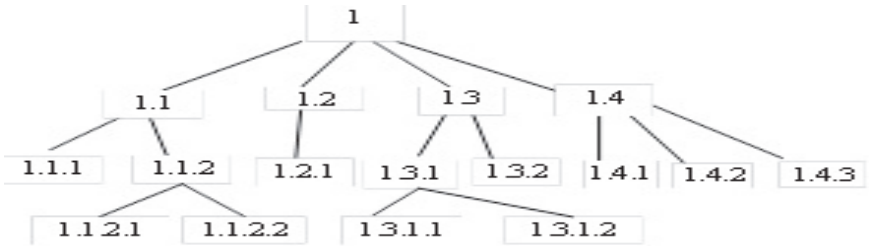


Figure 2.7. Dewey ID

ORDPATH labeling scheme is a prefix based labeling scheme with similar concept with Dewey ID [8]. It has all the advantages of Dewey ID and also allows the usage of negative integers and escapes even numbers to accommodate future insertions. As it can be seen from Figure 2.8, this approach partially supports dynamic XML document. However, bulk insertion at a given position consumes the reserved places hence re-computing of labels becomes inevitable.

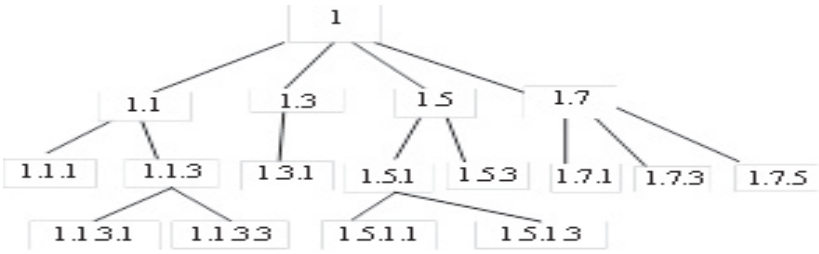


Figure 2.8. ORDPATH labeling scheme

LSDX – Labeling Scheme for Dynamic XML documents is also a prefix based labeling scheme that employs both integers and letters in the construction of a node's label [9]. The root node of the tree is labeled as 0a, where the integer component 0 represents the level or depth of the node and the alphabetic component represents the positional identifier. All structural relationships can be identified by looking at the labels.

LSDX is designed to meet the dynamic nature of xml data. The algorithm for generating labels for a node can be summarized as follows. Given a node *v* with *n* child

nodes: u_1, u_2, u_3 ... u_n, a unique code for u_1 is a combination of its *level + code of its parent node + "." + "b"*. The unique code for u_2 is its *level + code of its parent node + "." + "c"*. The labeling continues for the rest of child nodes in alphabetical order.

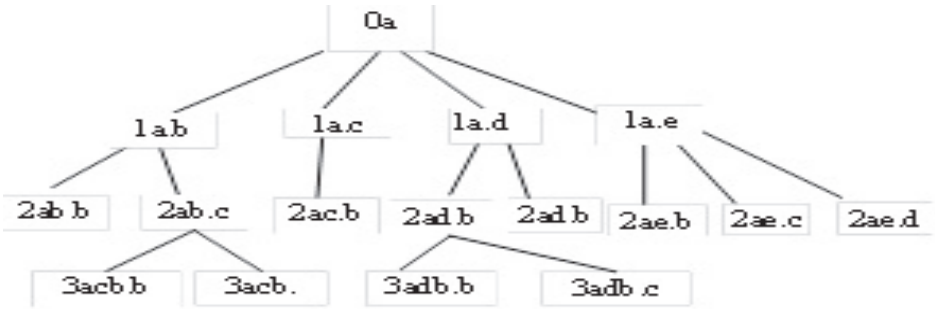


Figure 2.9. LSDX

For instance, in Figure 2.9, *3abc.b* is the descendant of *2ab.c,1a.b*, and *0a*, because it concatenates the labels *'abc','ab'*, and '*a*' as its prefix. One of the drawbacks of LSDX is its huge label size. The total size of labels depends upon the fan-outs and the depth of the tree.

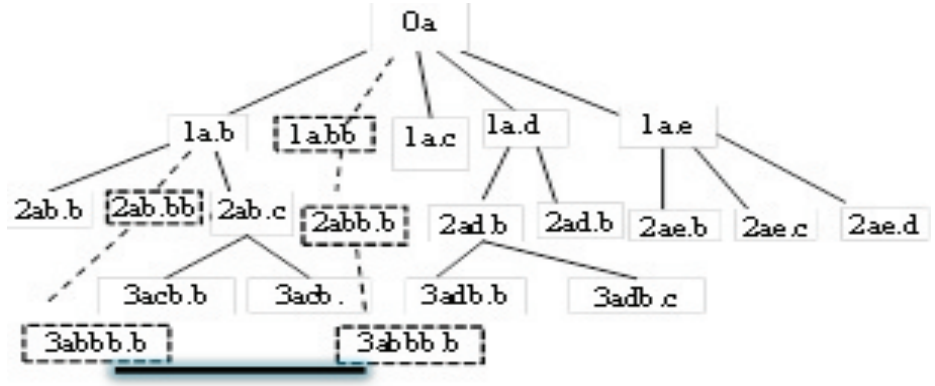


Figure 2.10. Collision in LSDX and Com-D

The other pitfall of LSDX is that it is not persistent. It had been reported that LSDX does not guarantee uniqueness at the time of labeling [20, 28]. In addition to the type of collisions identified above, Figure 2.10 shows that LSDX labeling scheme also does not guarantee uniqueness after insertions.

Com-D which stands for Compact Labeling Scheme for Dynamic XML documents [10] is an attempt to overcome the large label size drawback of LSDX. The idea behind this improved version of LSDX is to check repetitive letters, if any letter appears more than once, it shall be accumulated and replaced by number of its

occurrence + the letter itself. For example, the Com-D equivalents for LSDX labels "*1bdddccccxxx"*, and "*2dfffffgyyyyrrrrr*" are "*1b3d4c4x"* and "*2d5fg4y5r"* respectively. Com-D reduces the size of the labels significantly. It has all the advantages of LSDX; however, it is not persistent and has compression and decompression overhead while labeling and querying respectively.

ImprovedBinary labeling scheme uses bit strings in conjunction with a recursive algorithm to assign unique labels to each node in the XML tree. Figure 2.11 illustrates an ImprovedBinary labelled XML tree; the grey nodes indicating newly inserted nodes in an existing tree. When the XML tree is initially constructed, the root node is assigned the empty string. Initially the leftmost child of the root node is assigned the positional identifier 01 and the rightmost child of the root node is assigned the positional identifier011. From this point onwards, the Labeling algorithm is a recursive function that takes three inputs; an array of nodes (corresponding to all sibling children of a given node), the label of the leftmost sibling node and the label of the rightmost sibling node.

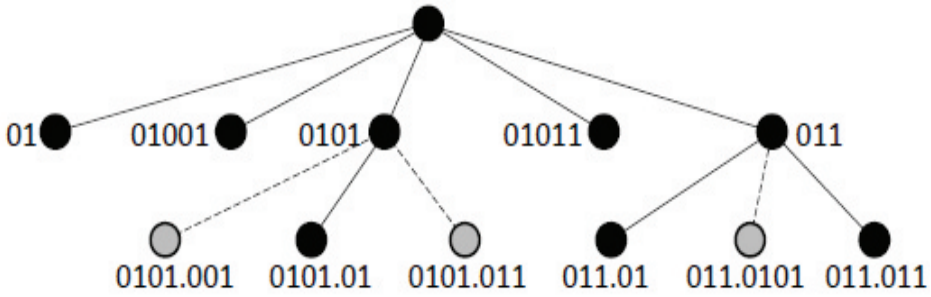


Figure 2.11. ImprovedBinary labled XML tree scheme [36]

An *AssignMiddleSelfLabel* function is invoked to compute a binary string (positional identifier) for the middle node residing between the leftmost and rightmost sibling nodes (e.g.: node 0101 in Figure 2.11). The middle node is determined using the simple calculation ((1 + n) / 2)) where n is the number of sibling nodes passed to the Labeling algorithm. The *AssignMiddleSelfLabel* function takes both values of the leftmost and rightmost nodes into account as well as their lengths to compute a binary string identifier that is minimal in length while ordered lexicographically between the leftmost and rightmost node labels. This is always possible due to a useful property of the algorithm that ensures the computed binary string always to end with 1. Finally, the

labeling algorithm uses the new left and right node labels to recursively call itself until each node in the XML tree has been labeled.

There are three possible types of node insertions. To insert a new node before the first sibling node, the positional identifier of the inserted node is assigned the identifier of the first sibling node with the last 1 changed to 01 (e.g.: node 0101.001 in Figure 2.11). To insert a new node after the last sibling node, the positional identifier of the inserted node is assigned the identifier of the last sibling node with an extra 1 concatenated (e.g.: node 0101.011 in Figure 2.11). To insert a node between any two nodes, the *AssignMiddleSelfLabel* function is used to compute the new positional identifier of the node (e.g.: node 011.0101 in Figure 2.11). The ImprovedBinary labeling scheme ensures that the positional identifiers and node prefixes are lexicographically ordered and consequently node labels are lexicographically ordered when performing component by component comparisons. This labeling scheme permits the evaluation of ancestor-descendant, parent-child and sibling-based relationships [36].

However, it is reported that the label sizes can grow quite rapidly. In particular, a skewed insertion before the first sibling node and after the last sibling node has a bit-growth rate of 1 for each insertion. Also, the ImprovedBinarylabelling scheme cannot completely avoid the relabeling of existing nodes due to the overflow problem. In other words, when the size of labels run out of memory, there will be a need for global relabeling of nodes [36].

Quaternary Encoding labeling scheme was proposed by the authors of the ImprovedBinary labeling scheme with the main motivation of completely avoiding the relabeling of nodes in the presence of updates [34]. The QED labeling scheme is conceptually similar to the approach taken by the ImprovedBinary scheme. However, instead of using a binary string, a quaternary code is employed consisting of four numbers 0, 1, 2, 3 and each number is stored with two bits, that is 00, 01, 10, 11. The number 0 is reserved for use as a separator and only 1, 2, and 3 are used in the QED code itself.

The Labeling algorithm is also a recursive function and operates in a similar manner to its counterpart in the ImprovedBinary scheme. The distinction arises from the fact that the ImprovedBinary scheme is based on the one half ($\frac{1}{2}$) node position whereas the QED scheme is based on one third ($\frac{1}{3}$) and two third ($\frac{2}{3}$) node positions. The AssignMiddleSelfLabel function is replaced with the GetOneThirdAndTwoThirdCode

function. Thus, rather than computing a QED code for the middle node, two QED codes (positional identifiers) are computed, one each for the ($\frac{1}{3}$)th and ($\frac{2}{3}$)th nodes that reside between the leftmost and rightmost sibling nodes.

The GetOneThirdAndTwoThirdCode function takes the values of the leftmost and rightmost sibling nodes into account as well as their lengths to compute two QED codes that always have the following lexicographic order properties: Left node < ($\frac{1}{3}$)th node < ($\frac{2}{3}$)th node < Right node. The Labeling algorithm recursively calls itself until all nodes in the XML tree have been labeled.

The key mechanism employed to reduce the label size is the use of the separator 0 (2 bits) to separate the different codes instead of explicitly storing the size of each variable code. The QED codes may vary in size but the size of the separator 0 remains constant. Each number in the QED code will always be represented by two bits and due to the properties of the labeling scheme, the numbers will never have the 2-bit value 00, which has been reserved as the separator. The QED codes are lexicographically and not numerical ordered. Furthermore, the properties of the QED labeling scheme ensure that an infinite number of QED codes may be inserted between any two consecutive labels without the need to re-label existing nodes and document order will be maintained [34].

A more compact version of QED is presented in [27] called the Compact Dynamic Quaternary String (CDQS) labeling scheme, which can completely avoid relabeling existing nodes in the presence of node insertions.

2.3. Multiplication Based Labeling Schemes

Multiplication based labeling schemes use atomic numbers to identify nodes. Relationships between nodes can be computed based on some arithmetic properties of the node labels. The main limitations of this approach are its very expensive computation and large size. Hence, it is unsuitable for labeling a large-scale XML document.

Unique Identifier labeling scheme is an example of a multiplication based labeling schemes [16]. This technique enumerates nodes using a k fan-out of nodes. Here, each internal node is supposed to have the same number of fan-out k. Thus, virtual nodes are created to balance the number of fan from each level. Each node is

assigned a label starting with integer 1 from top to bottom and from left to right as shown in Figure 2.12 .The UID technique has an interesting property for the parent node to be determined, based on the identifier of the child node. Given a node having the identifier i, the parent ID can be computed as

$$\text{Parent(i)}=Ceil(\frac{(i-2)}{K}) + 1 \quad (2.1)$$

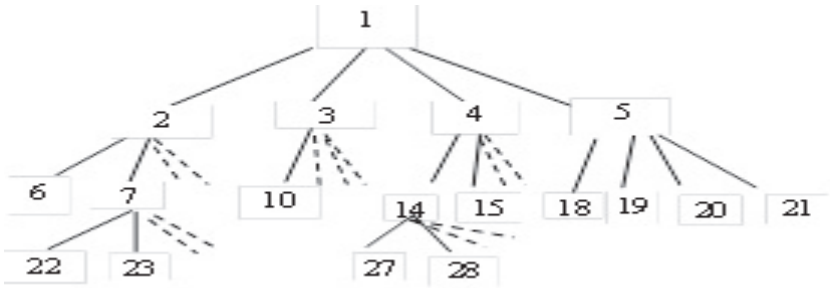


Figure 2.12. Unique Identifier labeling scheme with k=4

In UID the value for the maximum fan-out K should be determined prior to labeling. However, choosing K is an arbitrary selection based on anticipation. If the anticipated value for K is reasonably large, a valuable memory space may be wasted by the virtual nodes. On the other, hand if the value of K is not large enough to accommodate all children of a given node, relabeling the whole tree along with setting a new value for K becomes necessary. Especially, in a dynamic environment where insertions are common, resting K and relabeling the whole tree will be a costly operation. On the contrary, deletion intensive environments tend to waste memory spaces due to virtual nodes.

Prime Number labeling scheme makes use of prime numbers as the building blocks of labels [19]. There are two methods that can be used in prime number labeling scheme namely, bottom-up and top-down labeling schemes.

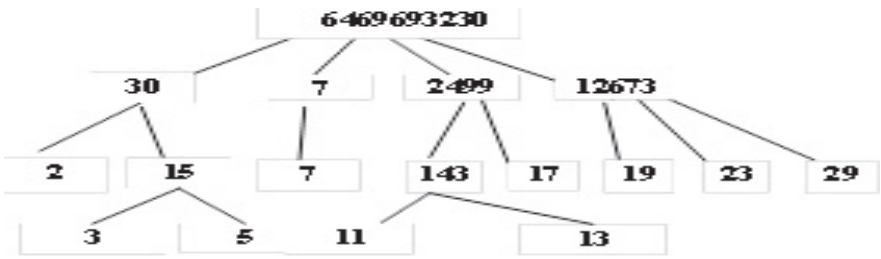


Figure 2.13. Bottom up Prime Number labeling scheme

For bottom-up approach, leaf nodes will be assigned a unique prime number which represents the self-label of the node itself. The parent node will be the product of the child nodes. For an instance, if the labels for two leaf nodes are 3 and 5 respectively, the label of the parent node will be 15 (3 x 5). Parent-Child relationship can be determined easily by calculating the factor for the number assigned to the parent node. Ancestor-Descendant relationship can be calculated by calculating the modulus of the ancestor and descendant node. If the result is 0 then Ancestor-Descendant relationship between the two nodes exists. On Figure 3.13, if the self-label of the ancestor node is 143 and child node = 13, 143 mod 13 = 0; thus, node with the label 143 is the ancestor of node with the label 13. The main problem with this approach is that nodes at the top level would be assigned relatively large numbers.

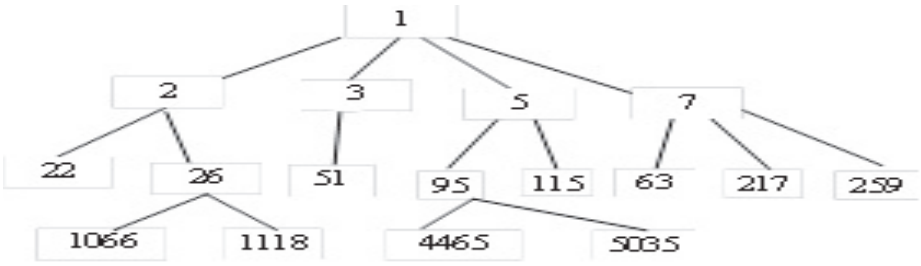


Figure 2.14. Top down Prime Number labeling scheme

On the other hand, top-down approach calculates the label of a node by multiplying parent label and self-label which is a unique prime number. Figure 2.14 demonstrates an example of a top down prime number labeling scheme. For instance, if the parent label is 2 and the self-label is 11 (prime number), the label assigned for this node is 22 (2x11). Parent-child relationship can be determined easily by dividing the child label and parent label. If these numbers are divisible, then parent-child relationship exists between these nodes. Ancestor–Descendant relationship can be ascertained using the same method as in bottom-up approach. Though this approach supports dynamic update, prime number used in this approach may grow larger which produces huge value for the self-label of a node. Since prime number that is assigned to a node can only be used once, larger amount of prime numbers are required for complex XML document.

2.4. Vector Based Labeling Schemes

The other groups of labeling scheme that are seen in literature are based on vector order. A vector code is a binary tuple of the form (x, y) where x > 0. Given two vector codes A: (x_1, y_1) and B: (x_2, y_2), vector a precedes vector B in vector order if and only if $\frac{x1}{y1} \leq \frac{x2}{y2}$. If we want to add a new vector C between vector A and B, the vector code of C is computed as x_1+x_2 , y_1+y_2).The vector order of A<B<C because $\frac{x1}{y1} \leq \frac{x1+x2}{y1+y2} \leq \frac{x2}{y2}$ holds true [17,18].

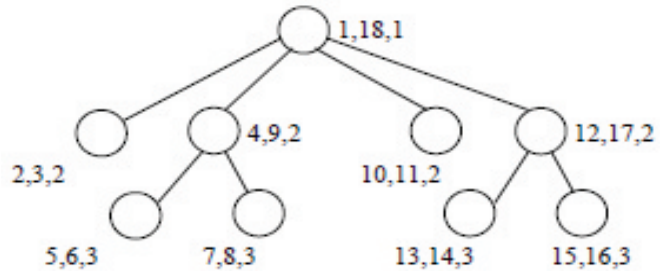
It is demonstrated that the vector based approach can be applied to both range based and prefix based labeling schemes [18]. DDE and CDDE are application of vector order approach to Dewey ID [18, 24] whereas V-containment is its application to containment labeling scheme (range based) [17]. Vector based labeling schemes avoid relabeling in update intensive environment and can be applied to any other labeling schemes, however, there is always a computation overhead to determine relationship among nodes.

V-Containment labeling scheme is an application of vector order to Containment labeling scheme [17]. The order of vector encodings intern is based on the numerical ordering of the Gradients of the vectors.

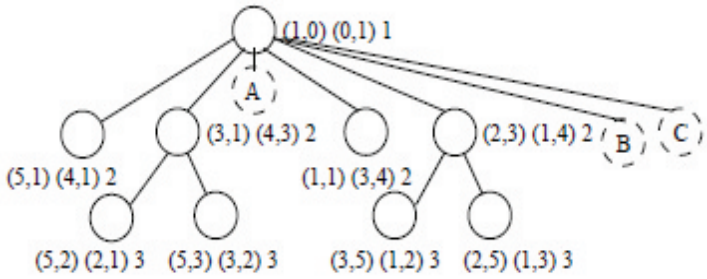
Given that the range of integers is from 1 to 18, we assign vector (1,0) (of *Gradient* 0) to the start position in the range which is 1; and (0,1) (of *Gradient +1*) to the end position in the range which is 18, i.e. v(1)=(1,0) and v(18) = (0,1). The middle position in the range [1, 18] can be found by: *middle* = *d*(1+18)/2 = 10. Hence v(middle) = v(10) = v(1) + v(18) = (1,0) + (0,1) = (1,1). Now that the range [1,18] is divided into two ranges: [1, 10] and [10, 18]. The middle position of [1, 10] is *d*(1 + 10)/2 = 6; and the middle position of [10, 18] is *d*(10 + 18)/2= 14. Therefore, v(6) = (1,0) + (1,1) = (2,1) and v(14) = (1,1) + (0,1) = (1,2). Likewise, this operation continues till all nodes in the tree are labeled.

Figure 2.15 shows an example of applying vector encoding to containment scheme. The *start* and *end* value of the original containment labels are replaced by their corresponding vector codes. The resulting *VContainment* labels are of the form (*startV; endV; level*) where *startV* , *endV* are two vectors. It is easy to verify that the property of

containment scheme holds. For example, Node((2,3),(1,4),2) is the parent of node((3,5),(1,2),3) as G(2,3)=1.5 < G(3,5)=1.67 < G(1,2)=2 < G(1,4) =4and 2+1=3.



(a) An example of Containment labeling scheme



(b) The V-Containment of Figure 2.15 (a)

Figure 2.15. Containment and V-Containment labeling schemes [17]

In V-Containment scheme to handle updated, the concept of the Granularity Sum of a vector V = (x, y) (denoted by GS (v)) is defined as x+y is used. To find a vector between two vectors in vector order, its Granularity Sum needs to be as small as possible so that the resulting label size is small. In Figure 2.15 (b), when inserting node A having both left sibling and right sibling, its *startV* and *endV* are bounded by *endV* of its closest left sibling and *startV* of its closest right sibling, i.e. (4,3) and (1,1). Moreover, GS(1,1)=2<7=GS(4,3). Therefore, the *startV* of A is $v1 + v2 = (5; 4)$ whereas *endV* is $v1+2*v2 = (6; 5)$. When inserting node B which has only left sibling, its *startV* and *endV* are bounded by the *endV* of its closest left sibling the *endV* of its parent, i.e. (1,4) and (0,1). Therefore, the *startV* of B is $v1 + v2 = (1; 5)$ whereas *endV* is $v1+2*v2 = (1; 6)$. Similarly, when we continue to insert C as the last child of the root, its *startV* is $v1+v2 = (1; 7)$ and *endV* is $v1+2*v2 = (1; 8)$.

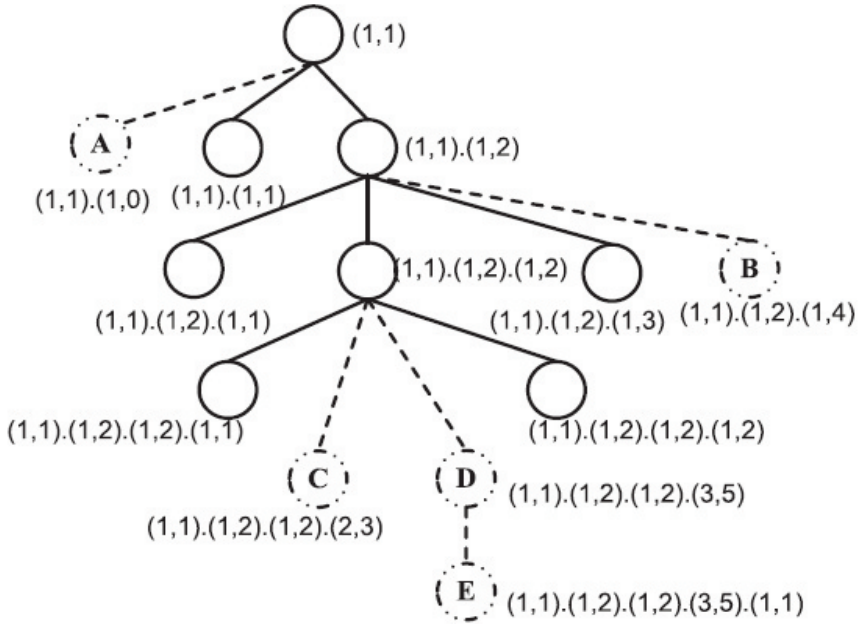


Figure 2.16. V-Prefix labeling scheme [18]

V-Prefix labeling scheme is also an application of a vector order to a prefix labeling scheme [18]. Figure 2.16 demonstrates an example of V-prefix labeling scheme. The idea of prefix labeling scheme is that given a V-Prefix label of the form (x1; y1):(x2; y2) … (xm; ym), we denote it as: v1:v2 … vm where v1=(x1; y1), v2 = (x2; y2) . . . vm = (xm; ym). Thus, V-Prefix label can be seen as a generalized Dewey label where every component is a vector code.

First, we consider the leftmost insertion of element node A in Figure 2.16. A takes the label of its parent as its parent label and a local order less than (1, 1). Thus, we get the new label of A by concatenating its parent's label (1:1) to (1; 0). Since B is inserted at the rightmost position after (1.1).(1.2).(1.3), we derive its local order to be (1; 4). C is inserted between two consecutive siblings. Its parent label is the same as its parent's label whereas its local order should fall between the local orders of its two siblings. That is, (2; 3) = (1; 1) + (1; 2). The local order of D is similarly computed: (3; 5) = (2; 3) + (1; 2). We process the insertion of a leaf node (E) by concatenating its parent label with an additional component, example, (1; 1) as shown in figure 2.17. V-Prefix labeling scheme handles updates with no need of relabeling. However, its label size is nearly double of its equivalent Dewey Id labels.

Dynamic Dewey Encoding (DDE) labeling scheme is an improved and optimized version of V-Prefix labeling scheme [24]. Their difference lies in the fact that

DDE's labels are not vector codes. An example of DDE labeling scheme is depicted in Figure 2.17.

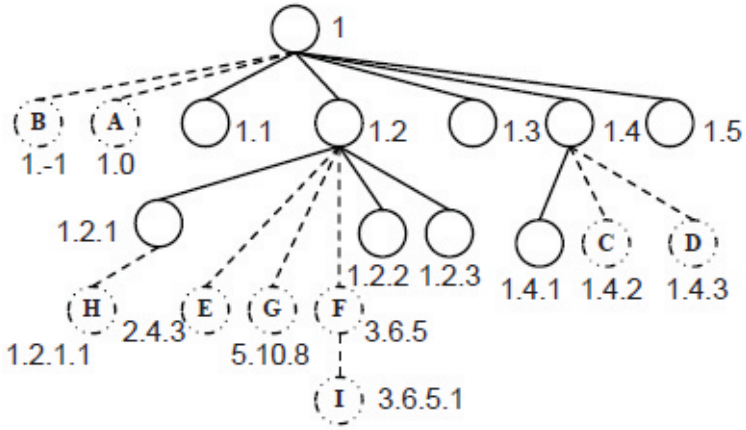


Figure 2.17. Dynamic Dewey Encoding (DDE) labeling scheme [18]

In Figure 2.17, node A is inserted before the first child of the root, we get its label 1.0 by decreasing the local order of 1.1 by 1. Node B is then inserted before A and its label is therefore 1.-1. Node C is inserted after the node with label 1.4.1; in order to get its label 1.4.2 by adding 1 to the local order of 1.4.1. Similarly, the label of node D is 1.4.3. Node E is inserted between two nodes with labels 1.2.1 and 1.2.2 and its label is 2.4.3 which equals to 1.2.2+1.2.3. Likewise, the labels of node F and G are 3.6.5 (2.4.3+1.2.2) and 5.10.8 (2.4.3+3.6.5) respectively. Next, node H is inserted as the child of leaf node 1.2.1, its label is 1.2.1.1 which is concatenation of its parent's label and 1. Node I is also inserted below a leaf node 3.6.5 and its label is therefore 3.6.5.1.

Compact Dynamic Dewey Encoding (CDDE) labeling scheme is also an application of vector order to Dewey ID labeling scheme. The initial labeling of CDDE is the same as DDE. However, in contrast to DDE it employs an optimized way towards reducing the labels size [24]. An example of CDDE is demonstrated in Figure 2.18.

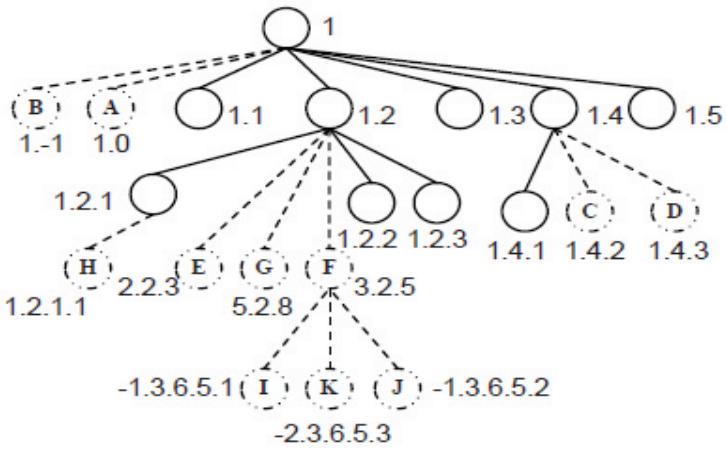


Figure 2.18. CDDE labeling scheme [24]

As illustrated in Figure 2.18, leftmost insertions (node A and B) and rightmost insertions (node C and D) are processed in the same way as DDE labels. Addition of two CCDE labels is handled as follow: *Let* $A : a_1:a_2 :a_3 : : : a_{m-1}:a_m$ *and* $A' = a'_1.a'_2.a'_3 \ldots a_{m-1}:a'_m$ *be two CDDE labels with sibling relationship, addition of them is defined as:*

$$A +_c A' = (a_1 + a'_1):a_2.a_3 \ldots a_{m-1}.(a_m + a'_m) \quad (2.2)$$

Accordingly, when inserting between node 1.2.1 and 1.2.2, the new label for node E is 2.2.3 (1:2:1 $+_c$ 1:2:2). Likewise, the labels for node F and G are 3.2.5 (2:2:3 $+_c$ 1:2:2) and 5.2.8 (2:2:3 $+_c$ 3:2:5).

In order to handle insertion below leaf nodes, CDDE introduces a concept called *The extension operation of a CDDE label A:* $a_1:a_2:a_3 : : : a_{m-1}:am$ *is* defined as*:*

$$EXT(A) \rightarrow \begin{cases} -1.a_1.(a_1 \times a_2).(a_1 \times a_3) \ldots (a_1 \times a_{m-1}).a_m.1 & when\ a_1 > 1 \\ a_1.a_2.a_3 \ldots a_{m-1}.a_m.1 & when\ a_1 = 1 \\ -1.(|a_1| \times a_2).(|a_1| \times a_3) \ldots (|a_1| \times a_{m-1}).a_m.1 & when\ a_1 < 0 \end{cases} \quad (2.3)$$

For insertion between nodes and below leaf nodes, CDDE has a technique Consider the insertion of H in Figure 2.18, given that the parent of H has label 1.2.1, the label of H is EXT(1:2:1) = 1:2:1:1. Similarly, the label of I is EXT(F) =EXT(3:2:5) = -1:3:6:5:1. Inserting node J is just processed as a rightmost insertion and the new label is

-1:3:6:5:2. To insert K between I and J, the new label is derived by adding the labels of I and J: -2:3:6:5:3 (-1:3:6:5:1 +c -1:3:6:5:2).

Finally, in recent years, it is common to see hybrid labeling schemes which balances the weakness of one approach with the strength of another approach [25 and 26]. There are also labeling schemes that capitalize on the characteristics of data structures. The two data structures, W-BOX (Weight-balanced B-tree for Ordering XML) and B-BOX (Back-linked B-tree for ordering XML), are B-tree based data structures which organize the labels for efficient updates [22]. The ideas provided here are fairly general and can be incorporated into any labeling scheme. The work presented in [21] uses XML type information and DTD to enhance query performance. A through survey and trends in XML labeling scheme can be referred from [35, 36, and 37].

2.5. Summary

The chapter is aimed at discussing related work in the area of XML labeling scheme. XML labeling schemes can be grouped into four major categories: Namely, Range based, Prefix based, Multiplication based and Vector based. A summary of the related work is presented on Table 2.1. It is demonstrated that the range based labeling schemes are faster in executing parent A-D (Ancestor Descendant) relationships. Prefix labeling schemes are generally provide adequate information to determine all kinds of structural relations; however they produce large label size. Large label size is the common feature of multiplicative labeling schemes though they guarantee uniqueness. Although most of the vector based labeling schemes are efficient in dynamic environment, generating a new label always needs computation.

The chapter presented a detailed analysis of labeling schemes. Label size, dynamicity, efficient structural relationship determination, uniqueness of labels and complexity are major criteria when choosing a labeling scheme. A desirable labeling scheme hence should be compact, dynamic, persistent, able to identify all structural relationships, and simple.

Labeling schemes that generate small labels, generally, do not provide sufficient information to determine all structural relationships. Perhaps they are not dynamic. On the contrary, labeling schemes supporting full structural relationships tend to generate large label sizes. Moreover, though dynamic labeling schemes avoid relabeling of

nodes, they achieve this at the cost of poor query and labeling time. It is also shown in this chapter that in-persistence and complexity had been the main challenges of dynamic labeling schemes.

Table 2.1. Summary of labeling schemes

	Advantage	Disadvantage	Structural Relationships	Dynamic	Persi-stent
Range based labeling schemes					
Traversal Order Based	Small size, Fast A-D Query	P-C, Sibling, Order – Recursive ,Not Dynamic	A-D	N	Y
Extended Pre order	Small size, Fast A-D Query	P-C, Sibling, Order – Recursive ,Not Dynamic	A-D	N	Y
Containment	Small size, Fast A-D Query	P-C, Sibling, Order – Recursive ,Not Dynamic	A-D	N	Y
P-Containment	Fast P-C Query	A-D– Recursive ,Not Dynamic	P-C	N	Y
Dynamic Interval Based	Small size, Fast A-D Query	P-C, Sibling, Order – Recursive	A-D	Y	Y
Prefix based labeling schemes					
Simple Prefix	All Relations	Large Size, Collision	All	N	N
Dewey ID	All Relations	Large Size	All	N	Y
ORDPATHS	All Relations	Large Size	All	N	Y
LSDX	Dynamic	Large Size	All	Y	N
Com-D	Dynamic, Small size	Collision	All	Y	N
Improved Binary	All relationships	Large size	All	Y	Y
Quaternary Encoding	All relationships	Large size	All	Y	Y
Multiplication based labeling schemes					
Unique Identifier	Fast for Order Queries	Computation cost, Virtual nodes	P-C	N	Y
Bottom up Prime Number	Unique	Large size		N	Y
Top down Prime Number	Dynamic	Large size	A-D,P-C	Y	Y
Vector based labeling schemes					
V-Containment	Fast A-D Query	Computation cost, large size	A-D	Y	Y
V-Prefix	Skewed insertion	Computation cost, impractical size	All	Y	Y
DDE	Skewed insertion	Computation cost, large size	All	Y	Y
CDDE	Skewed insertion	Computation cost, large size	All	Y	Y

CHAPTER 3

ORDERBASED LABELING SCHEME

OrderBased labeling scheme presented in this thesis is based on combination of letters and numbers. Each label contains level, order of the node in the level and the order of its parent. First part of the label is numeric and indicates the level information of a given node. The second part gives alphabetical order of the node relative to the left most node of the level. The last part is the order of the parent node. The order and the level information guarantee unique labels. The usage of characters enables it to generate a completely new order before and after the position of a given node, and also between two nodes without affecting existing order in case of insertions. For instance given two orders O_1, and O_2 where O_1="*abc*" and O_2= *"bd*", we can generate as many strings as we need which are between O_1 and O_2 in alphabetic order ("*abcb*", "*abcd*", *abce*"..).

In OrderBased labeling scheme each label is a triple <*level, order, parentorder*>, where *level* is an integer that represents the distance of the node from the root node, *order* is a character that represents the level based horizontal distance of the node from the left most node at each level, *parentorder* is the parent's order of a given node .The level of the root node is 0, and the level of the children of the root node is 1. Likewise, the levels of other nodes can be computed as the distance of the node from the root node as seen in Figure 3.1.

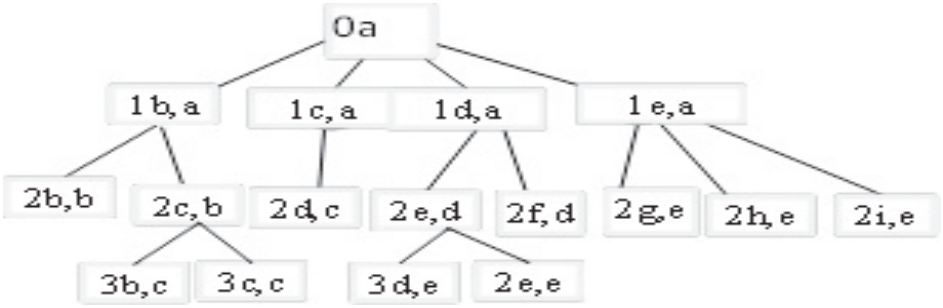


Figure 3.1. OrderBased labeling scheme

An OrderBased label provides the information of the parent-child, and siblings-following/previous in a direct way, and ancestor-descendant relationships in recursive manner. For example in Figure 3.1, the node with label "1e, a" is the parent of the nodes

with labels “2g, e”, “2h,e”, and “2i,e”. This parent to child relationship is provided because the parent order of the three nodes is “e”, and presumably their level is 1+1=2. Moreover, nodes that have the same level information and with the same parent order are siblings. However, to find the ancestors /descendants of a given node, first there is a need to move to the parent/children, and then the parent of the parent/children recursively till the intended level is reached.

3.1. Optimizing Label Size

To address the problem of large storage size, OrderBased labeling scheme has a routine which optimizes the label size of every level. Small label sizes enhance query, update and labeling performances. Before labeling or making any insertions, the OrderBased labeling scheme computes the optimal number of characters needed to label the nodes at every level. To illustrate the need of optimizing the size, we will give a brief description of the size requirement in terms of number of characters.

Assume the total number of nodes at a given level is ***M***. If we start labeling order of the first node in the level by ‘b’, the labeling continues with ‘c’, accordingly the orders of the 25th and 26th nodes will be ‘z’ and ‘zb’ respectively. Since there is a need of concatenating extra ‘b’ after reaching the letter ‘z’ in ever 26th node, the size of the order increases dramatically. If the total number of nodes at a given level ***M*** is not greater than 25, we can generate M unique one character length orders using alphabets from b to z. If M is between 26 and 50 inclusive, we use 25 single character alphabets and (M-25) double character length. For example If M= 10, 40, 66, and 90, then size requirement is then 1(10) =10, 1(25) + 2(40-25) =55, 1(25) + 2(25) + 3(66-50) = 123, and 1(25) +2(25) + 3(25) + 4(90-75) = 210 number of characters respectively.

The total size requirement for orders at a given level with a total number of nodes M can be generalized as,

$$25 * \sum_{i=1}^{w} i \; + \text{M mod } 25*(\text{w+1}) \quad \Longleftrightarrow$$

$$25 * \frac{(w+1)w}{2} + \; M \; mod \; 25 * (w+1) \qquad (3.1)$$

where w=floor (M/25)

In order to have an optimal size of orders, the OrderBased labeling first calculates the number of characters needed to label M number of nodes.

$$25^x = M$$
$$\log 25^x = \log M$$
$$X = Ceil(\frac{\log M}{\log 25})$$

The function *Ceil* returns the smallest integer that is greater than or equal to the given expression. For example, *Ceil* (1.45) =2, *Ceil* (9.8) =10, and *Ceil* (11) = 11.

By this approach the first child is labeled with x number of b's. For example if M is 625, X computed to be 2 , the order of the 1st ,2nd , 26th, 624th , and 625th is 'bb','bc','cb','zy', and 'zz' respectively. By this approach, the total size of orders for all nodes of a given level is

$$M * Ceil\left(\frac{log\, M}{log\, 25}\right) \quad (3.2)$$

Table 3.1. Analytical storage requirement

M	Optimized	Un- optimized
24	24	24
50	100	75
75	150	75
100	200	250
1000	3000	20500
2000	6000	81000
1000000	5000000	20000500000

Table 3.1 shows a comparison of the total number of characters needed to label the order of nodes using optimized and un-optimized approaches. For M<=25 both approaches need same storage requirement, while the number of nodes M is from 26 to 99, storage requirement for the un-optimized approaches is slightly smaller. Generally, for the number of nodes M>100, the storage requirement for the optimized approach is always smaller than the storage requirement of the un-optimized approach. The difference of the storage requirements for the two approaches considerably increases as the number of nodes M increases. This makes the optimized approach to be preferred to the un-optimized approach.

In OrderBased labeling scheme, optimizing the size is a prior operation before labeling and inserting a sub tree. The *Determine-size* routine seen in Figure 2, takes the XML tree to be labeled or inserted as input computes the number of nodes at every level, then returns a string array.

```
Determine-size (XML tree)
{
        String array Y[height of tree]
        Integer array X[height of tree]
        Determine the total number of nodes per each  level
        Put them into an integer array X
        for ( i=0 to height of tree)
        {
                X[i] = Ceil(log X[i] / log 25)
                Y[i]=concatenate X[i] number of 'b'
        }
        Return Y
}
```

Figure 3.2. Determine-size routine

For example , if a given XML document has 500, 3000, 9000 , 1000000, and 2000000 number of nodes at 1st, 2nd ,3rd, 4th and 5th level respectively, the above routine returns *Y, where Y[1]='bb', Y[2]='bbb', Y[3]='bbb', Y[4]='bbbbb', and Y[5]=' bbbbb'.*

3.2. Generating the Order of a Node

Rule 1

Label the order nodes of a given level starting by the concatenation of b's returned by the ***Determine-size*** *routine. For the second, third, and forth node, increment the last character to 'c','d', and 'e' respectively. Accordingly for the rest of the nodes, increment the orders alphabetically.*

For example if 'bbb' is the string returned for a given level, the order of the 1st, 2^{nd} , 25^{th} , 26^{th} , an 15625^{th} node are labeled as '*bbb'*, '*bbc'*,' *bbz'*, '*bcb'*,and 'zzz' respectively.

3.3. Generating Orders for Newly Inserted Nodes

Rule 2

To insert a node before the first node of a given level, get the order of the node then count down to the preceding alphabet, if all characters are "b", insert "a" before the last "b".

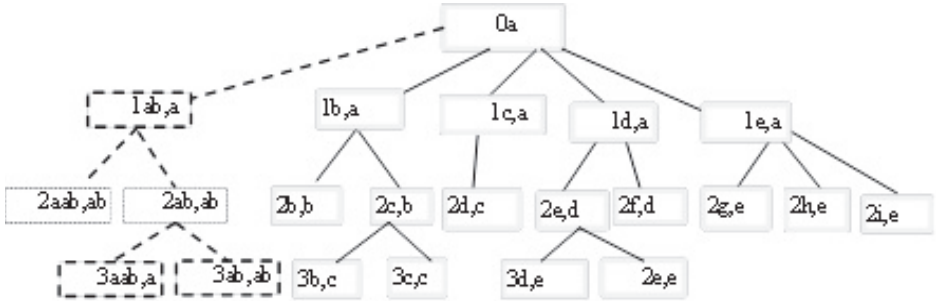


Figure 3.3. Insert a sub tree before the first node of a given level

Figure 3.3 shows how insertion before the first node of a given level is handled by OrderBased labeling scheme. Here Rule 2 is applied to insert a node before "1b,a". Because there is no node before it we add 'a' before 'b' then we will have "1ab,a". At the second level, there are two nodes to be inserted before "2b,b". Thus, applying Rule 2, the labels of the inserted nodes will be "2,ab,ab", and "2aab,ab". Similarly, the labels of the two nodes at level 3 will be "3ab,ab" and "3aab,ab". Insertions before the first node of a given level can be handled by applying Rule 2 without the need of relabeling.

Rule 3

To insert a node between two nodes, keep counting from the code standing before it so that the code for the new node will be greater than the code of its previous sibling and less than the code of its next sibling.

It can be seen from Figure 3.4 that, insertion between two nodes can be made without affecting the order of the existing nodes. Applying Rule 3 at the first level a unique label "1bb,a" is generated between "1b,a" and "1c,a". Likewise at level 3 and level "2cb,bb" , "2cc.bb", and "3cb,cc", "3cd,cc" respectively are unique labels generated between two nodes without the need or relabeling.

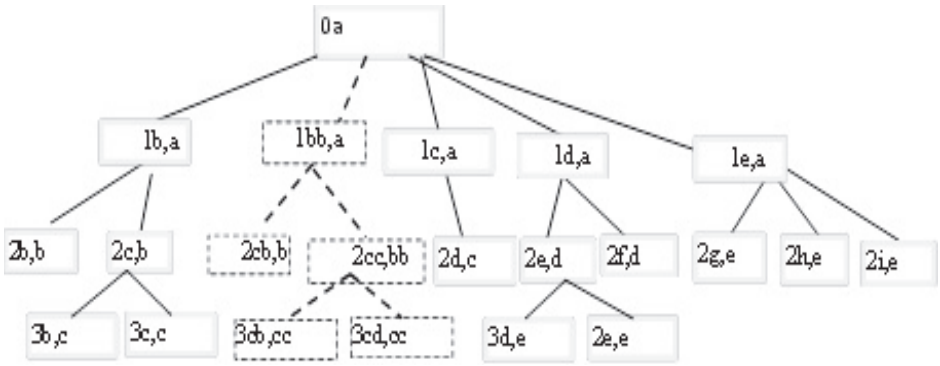


Figure 3.4. Insert a sub tree between two nodes

Rule 4

To insert a node after the last node of a level, increment the order of the last order alphabetically.

Figure 3.5 shows how insertion after the left most node of a tree is handled. Rule 4 states that insertion after the last node of a given node is handled by incrementing the order of the last node alphabetically. That is after "1e,a" is "1f,a", likewise, "2j,f" , "2k,f" and "3f,k","3g,k' are after "2,i,h" and "3e,e" respectively.

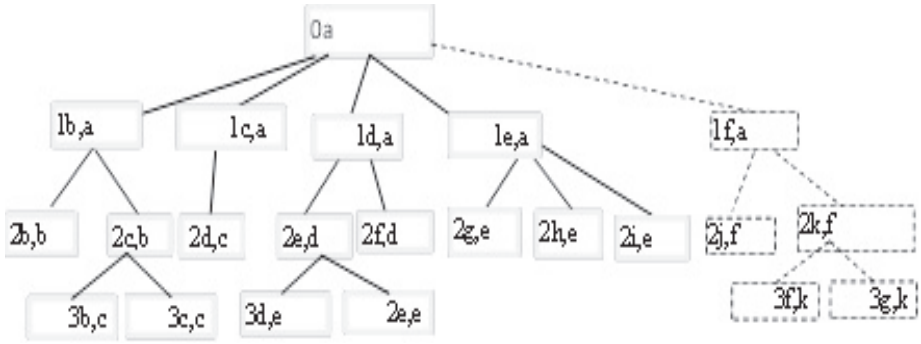


Figure 3.5. Insert a sub tree after the last node of a given level

Figure 3.3, Figure 3.4 and Figure 3.5 demonstrate that inserting a sub tree at any arbitrary position does not need any relabeling of nodes. Rules 2, 3 and 4 guarantee unique labels are given to the newly inserted nodes or sub tree with regardless of the point of insertion. OrderBased labeling scheme is persistent in that it insures a uniqueness of labels in a dynamic environment.

CHAPTER 4

PERFORMANCE EVALUATION

In this performance evaluation part of the study, OrderBased labeling scheme is compared with the LSDX and Com-D (Compressed LSDX) labeling schemes. These labeling schemes are chosen because they share main feature and design goals. Using combinations of letter and numbers, including the level information of a node in every label, and avoiding relabeling when update occurs are the common feature and design goals of the three schemes. Moreover, because three of them contain the information about the label of the parent node, they can be grouped under prefix based labeling scheme.

There are four sets of tests in this performance evaluation: the first set compares the storage requirement of three schemes. The second set analyzes labeling time. The third set examines the query performance and the last set investigates update performance.

4.1. Experimental Setting

The performance evaluation is conducted on an Intel(R) Core™2Duo CPU E8400 @3GHz 2.7 GHz and 2.00 GB of RAM Windows 7 Professional computer. All schemes are implemented using Visual Basic .net 2010. So as to avoid discrepancy, each querying and labeling time performance test is run 5 times and the average is taken.

A B+ tree is used to store the labels. In the non-leaf nodes of the B+ tree, only labels are stored. In addition to labels, the leaf nodes contain the name of nodes of the XML tree or attributes with their corresponding values [15 and 22]. The full implementation source code is presented in the Appendix. The program can be run Visual Basic 2008 or later version.

4.2. Characteristics of the Datasets

The datasets used in this performance evaluation are generated using xmlgen of the XMark: Benchmark Standard for XML Database Management [11]. The xmlgen produces XML documents modeling an auction website, a typical e-commerce application. It generates a well-formed, valid and meaningful XML data. Xmlgen is well known for its efficient and scalable generation of XML documents of several GBs.

Number and type of elements are chosen according to a template and parameterized with certain probability distributions. The words for text paragraphs are taken from Shakespeare's plays. The generator is deliberately designed to have only a single parameter: factor. The factor parameter determines the size of the document generated. It accepts float number from 0 to any number. Zero value for the factor generates the minimum document.

By giving values from 0.5 to 1.0 to the factor parameter of the xmlgen, six datasets with size of 56.2 to 113 MB, with number of nodes ranging from 832,911 to 1666315 and maximum fan-out starting 12750 to 25,500 are generated. The characteristics of the datasets are seen in Table 4.1.

Table 4.1.Characteristics of datasets

Dataset	Factor	Size(MB)	No of Nodes	Max Fan-out
D05	0.5	56.2	832911	12750
D06	0.6	68.2	1003441	15300
D07	0.7	79.7	1172640	17850
D08	0.8	90.7	1337383	20400
D09	0.9	102	1504685	22950
D10	1.0	113	1666315	25500

4.3. Storage Requirement

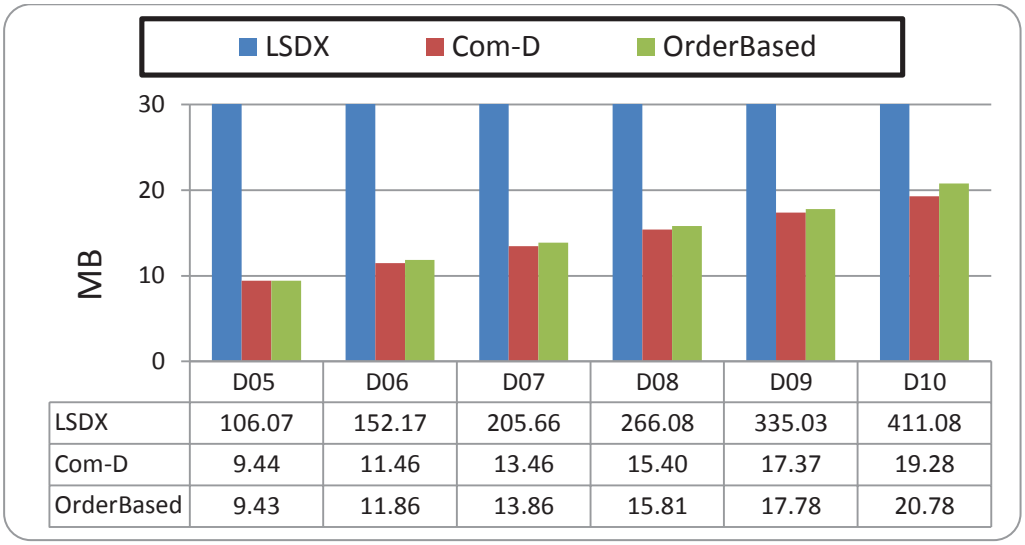
In this performance evaluation test set, the storage requirement for the three schemes is studied. For the six datasets introduced in the previous section, the sizes of labels in MB are shown in Figure 4.1.

The storage requirement of LSDX labels is the largest as compared to the rest of the two. This resulted from the fact that LSDX label size depends on fan-outs and the

height of the tree. To illustrate: for the first 25 children the size of a LSDX label is 25 characters (letter b to z) plus the label of the all its ancestors. Since after every 25th children we reach at letter z, there is a need to concatenate b. This makes the label size to increase by one character. The storage requirement for LSDX labels depend on the fan-outs and the height of the tree (since each label contains the label of its ancestor nodes). The more the number of fan-outs and the taller the tree, the larger is the label size.

Com-D is a compressed version of LSDX. The compression is done by counting the number of times a letter is consecutively repeated. For example if the LSDX label of an XML node is *abzzzzzzrr.dd*, its equivalent Com-D label *is ab6z2r.2d* [10].

As it can be seen from Figure 4.1, for all the datasets used in this performance analysis, Com-D needs the least storage requirement. Com-D label size is from 4.7% to 8.9% of LSDX label size. The figure also demonstrates that the storage requirement for OrderBased labels is from 5.1% to 8.9% of the storage requirement of LSDX labels. For dataset D05, Com-D label size is the same as that of OrderBased. However, for the rest of the datasets, the storage requirements are from 92.2% to 97.7% of the label size of OrderBased.



	D05	D06	D07	D08	D09	D10
LSDX	106.07	152.17	205.66	266.08	335.03	411.08
Com-D	9.44	11.46	13.46	15.40	17.37	19.28
OrderBased	9.43	11.86	13.86	15.81	17.78	20.78

Figure 4.1. Storage requirement

Collision is one of the drawbacks of the LSDX and Com-D labeling scheme. For every dataset used in this performance evaluation, the two schemes give the same label for more than one XML nodes. Table 4.2 demonstrates the number of collisions

detected while labeling using the LSDX and Com-D labeling schemes. For this reason, both LSDX and Com-D are impractical.

Table 4.2. Number of collisions detected by LSDX and Com-D

	D05	D06	D07	D08	D09	D10
Collision	7	43	34	13	30	86

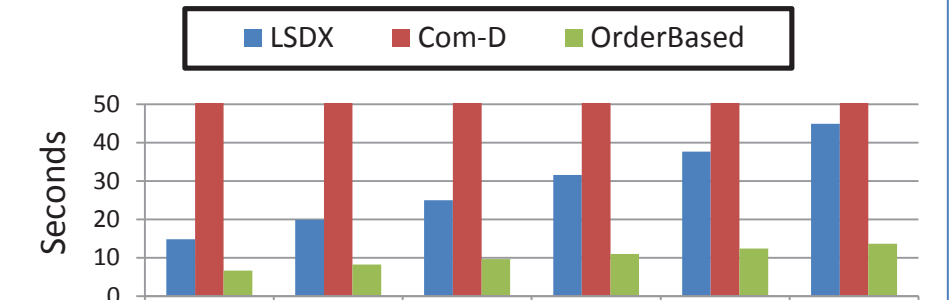
In OrderBased labeling scheme, there is no collision. It avoids collision by keeping a global level based horizontal order and parent order. Both LSDX and Com-D are impractical due to the existence of collision. OrderBased is superior to the two labeling schemes for its persistence. Moreover, it is optimized label size is superior to LSDX and nearly as good as Com-D.

4.4. Labeling Time

In this sub section, the time required to label a given XML document is studied. The time required for labeling that is seen on Figure 4.2 below is the average labeling time taken from five tests done on each dataset. The labels are generated by a depth first traversal for the three labeling schemes.

Figure 4.2 stipulates that for all the six datasets, LSDX is at 7.99 to 15.74 times faster than Com-D. With regard to labeling time, OrderBased labeling scheme is approximately 2.2 to 3.9 and 17.28 to 51.8 times faster than LSDX and Com-D labeling schemes respectively.

The labeling time performance hit of OrderBased over LSDX is due to LSDX's larger label size (In Figure 4.1, the total label size of LSDX is more than 100 to 400 times larger than the total label size of OrderBased). Even though Com-D labels need the minimum storage requirement, it takes the longest labeling time. This decrease in labeling performance results from compression overhead.



	D05	D06	D07	D08	D09	D10
LSDX	14.8582	19.937	25.025	31.543	37.674	44.9434
Com-D	118.7752	189.119	279.334	395.632	535.502	707.726
OrderBased	6.6862	8.315	9.719	11.014	12.418	13.665

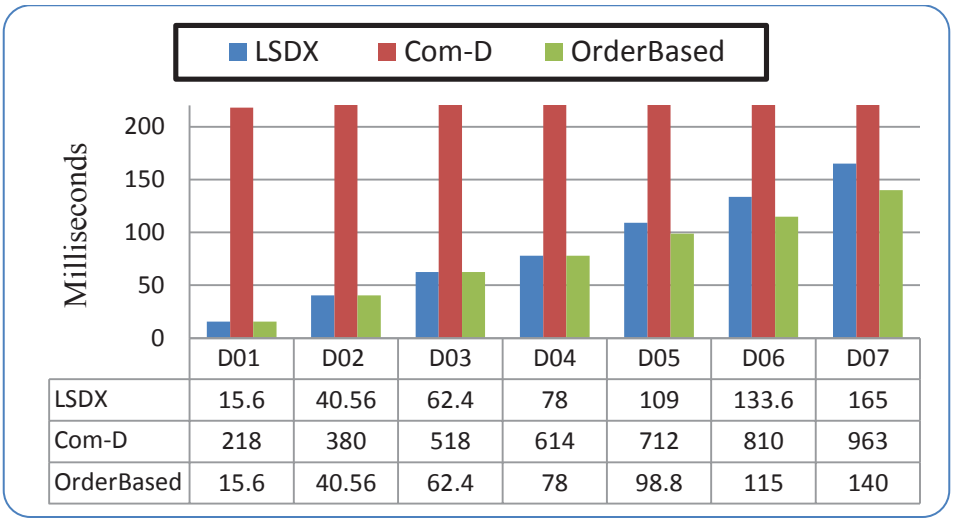
Figure 4.2. Labeling time

The labeling time test set shows that OrderBased labeling scheme takes the least labeling time compared to LSDX and Com-D labeling schemes. This labeling time performance hit of OrderBased is because of the optimal label size. From this result it can be concluded that compression degrades labeling time performance more than large label size does.

4.5. Query

In this performance evaluation part, a query which returns all descendants of the root node is run. Finding descendant of a given node depends on the time required for Parent-Child, and Sibling, and Order queries.

Given an ancestor finding its descendants is one of the structural queries found in XML querying. These types of queries are usually seen in XPath statements. The query for retrieving all descendant of a root node is equivalent to the XPath expression Site/*(since the root node of the data sets used in this performance evaluation is site).



	D01	D02	D03	D04	D05	D06	D07
LSDX	15.6	40.56	62.4	78	109	133.6	165
Com-D	218	380	518	614	712	810	963
OrderBased	15.6	40.56	62.4	78	98.8	115	140

Figure 4.3. Time required for retrieving all descendants of a given node

For a reasonably small size and small number of nodes of a given XML data set, LSDX and OrderBased take nearly the same time. However, OrderBased executes faster as the data size and the number of nodes increase. In addition, both LSDX and OrderBased labeling scheme are incomparably faster than Com-D. This performance variation comes from decompressing overhead for Com-D. Com-D querying involves decompressing of each label. It can be seen from Figure 4.3 that decompressing degrades query performance than label size does.

OrderBased labeling scheme is superior to LSDX and Com-D with respect to querying time. Such a performance hit is due to its optimized size of labels.

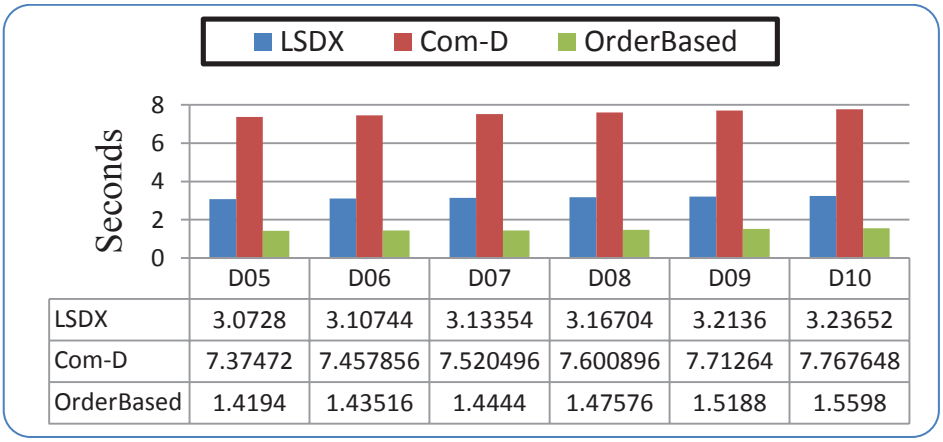
4.6. Updates

In this update performance evaluation of the study, the time needed to insert a sub tree, and delete a sub tree for the three schemes is analyzed. The most profound problem with most XML labeling schemes is that they are designed with an assumption of static document. Whenever a deletion or an insertion is done on the XML document, relabeling of all or part of the XML tree is inevitable. However, in real world applications, updating an XML document is an important and necessary operation.

4.6.1. Inserting a Sub Tree

In this performance evaluation part of the study, the time to insert a sub tree which is an XML by itself is seen. For this study, an XML dataset D01 of 11.3 MB is generated by giving 0.01 to the factor parameter of the xmlgen generator. Inserting D01 at different part of the XML tree produces same time. Thus, for convenience for all the datasets the D01 is inserted as the child of the root node.

Figure 4.4 shows that the time of insertion of DO1 to the six datasets is nearly constant irrespective of their size. Moreover, insertion time mainly depends on the size of the inserted sub tree.



	D05	D06	D07	D08	D09	D10
LSDX	3.0728	3.10744	3.13354	3.16704	3.2136	3.23652
Com-D	7.37472	7.457856	7.520496	7.600896	7.71264	7.767648
OrderBased	1.4194	1.43516	1.4444	1.47576	1.5188	1.5598

Figure 4.4. Insertion time

Com-D takes at least twice and four times longer time than that of LSDX and OrderBased labeling schemes. These performances degradations are resulted from the time needed for compression, since all labels have to be compressed. Figure 4.4 illustrates that OrderBased is superior to the rest of the schemes with respect to insertion time in that it is twice faster than LSDX and four times faster than Com-D. OrderBased insertion time performance hit is due to its reasonable small size.

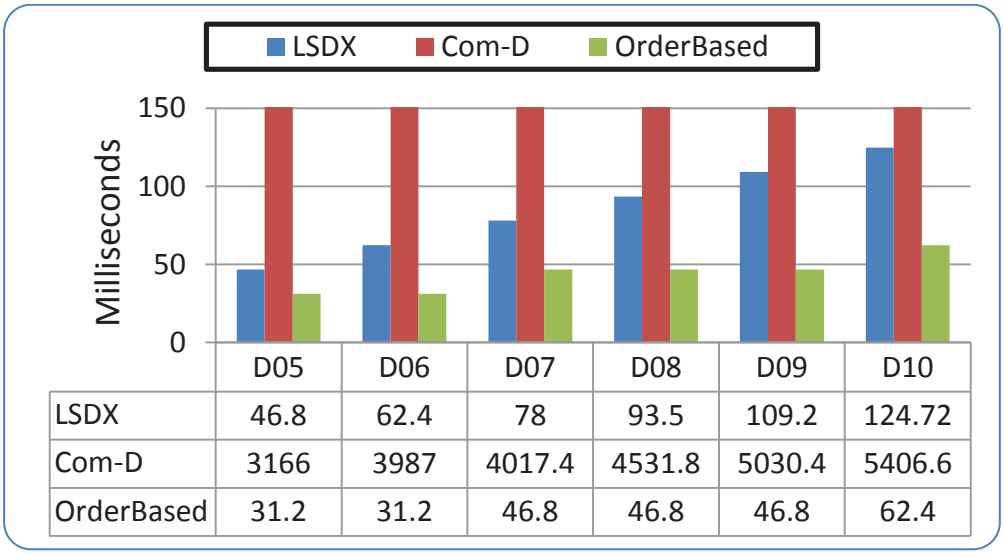
4.6.2. Deleting a Sub Tree

In this part of the performance evaluation, the time needed to delete a sub tree is studied. All the three schemes avoid relabeling after deletion. The spaces and the labels deleted can be used for future insertions.

For the B+ tree used to store the labels of XML tree nodes, a mechanism of lazy deletion is employed. Lazy deletion does not rebalance the B+ tree on deletion. Avoiding rebalancing on deletion has been justified empirically [12, 13 and 14].

Delete site/closed_auctions: delete the node with name closed_auctions.

Figure 4.5 depicts that Com-D takes the longest time to delete in all the six datasets. This is because decompressing is necessary to determine whether the nodes are descendants of the deleted node. OrderBased labeling scheme deletion is 1.5 to 2.33 faster than LSDX.



	D05	D06	D07	D08	D09	D10
LSDX	46.8	62.4	78	93.5	109.2	124.72
Com-D	3166	3987	4017.4	4531.8	5030.4	5406.6
OrderBased	31.2	31.2	46.8	46.8	46.8	62.4

Figure 4.5. Deletion time

4.7. Discussion on Results

In this performance evaluation study we have seen the storage requirement, labeling time, querying time, insertion time and deletion time for OrderBased, LSDX, and Com-D labeling schemes.

The first test set for storage requirement, LSDX labels need the largest storage requirement .Com-D labels need the least space. The storage requirement for OrderBased labels is nearly as good as the storage requirement for Com-D labels (2.34% to 7.7% greater than Com-D). Com-D reduces the total size of LSDX labels by 91% to 95%. On the other hand, the size requirement for OrderBased labels is 91.1 % to 94.94% less than the storage requirement for LSDX.

The second test set for labeling time requirement shows that OrderBased needs the least labeling time whereas Com-D takes the longest labeling time because of compression overhead. From this result it can be concluded that the larger the label size, the faster the labeling is. On the other hand, the compression reduces the label size; it degrades labeling time more than large label size does.

For querying performance, for small data sets, it seems LSDX and OrderBased take equal time. However, as the data size increases, it becomes clear that OrderBased needs the least time. Com-D has the least performance because of the need of decompression.

In the fourth test, update performance (insertion and deletion) time requirement is studied. With regard to insertion, OrderBased needs the least time. Again Com-D needs the longest time because of compression overhead. For deletion time requirement test, OrderBased needs the least time.

CHAPTER 5

CONCLUSION

XML is a standard format for structuring and transmitting information across platforms. An XML document has a natural order. In addition to the natural order, Parent- Child, Ancestor –Descendant, Siblings relationships constitute the structural component of an XML document. In short, XML is a semi structured document in which XML query processing mainly depends on the structural relationships among the nodes and their order. Labeling scheme is hence a systematic way of assigning labels to the nodes of an XML tree such that the information embedded in labels conveys information about the relationships among nodes.

XML labeling schemes can be grouped under four categories: Range based, Prefix based, Multiplication based, and Vector based. In this thesis, by giving example labeling schemes to the four aforementioned categories of labeling schemes, a detailed illustration along with the advantages, disadvantages, and improvements is presented.

Range based labeling schemes are generally characterized by incorporating <START, END> arguments to the labels. The START and END components of an XML label tell that any label enclosed in the range is regarded as the descendant or child of a given label. Labeling schemes under this category are fast in determining Ancestor-Descendant relationships. Inefficiency in update intensive environment due to the need of relabeling and inability to determine all structural relationships are the main challenges of the Range based labeling schemes.

Prefix based labeling schemes include the label of their ancestors and the self-label. This makes determination of all structural relationships easy just by looking at the labels. However, with a tall XML tree the size of the labels grow dramatically. In all prefix based labeling schemes, right most insertions adapt gracefully without the need of relabeling. Not only does this approach have an inherent impractically large label size problem but also improvements made to fully support update happen to be in persistence.

Multiplication based labeling schemes, on the other hand, uses atomic numbers and multiplication and division operations to determine structural relationships. Because unique number has to be used, the size of labels increases dramatically.

Last but not least, Vector based labeling schemes employ a vector order. The ideas used here are general and can be applied to the other categories of labeling schemes. However, insertions, and querying always need mathematical computations.

This thesis pointed out the challenges of dynamic labeling scheme for XML documents. Large storage requirement, inefficient labeling or querying time and complexity are challenges of dynamic labeling schemes. To address these problems, a novel fully dynamic labeling scheme called OrderBased is proposed.

An OrderBased label is a triple consisting of the level of the node, the horizontal distance of the given node from the left most nodes, and the order of the parent node. The level part of the label is an integer which stores the information of the level where the node is found. The order component of the level is an alphabet string from ‘b’ to ‘z’. Character ‘a’ is reserved to facilitate future insertions. The peculiar characteristic of character enables us to make skewed insertions without the need of relabeling the existing nodes. Whereas the combination of level, order, and parent order guarantee uniqueness, this approach introduces an optimization routine to reduce the size of labels. OrderBased labeling scheme is fully dynamic, persistent, compact and simple to understand and implement.

In performance evaluation studies, OrderBased labeling scheme is compared with LSDX and Com-D. The reasons for choosing the two labeling schemes for comparison are that they are fully dynamic, include level information, and intend to compress the size of labels. Last but not least, since they include the label of the parent of a given node except for the root node, they can be grouped under prefix based labeling schemes.

Storage requirement of labels, labeling time, querying time and update time are the attributes we measure in the performance evaluation studies. To avoid inconvenience, each time measuring tests were run five times and the average is recorded.

Performance evaluation study results can be summarized as follows

- Storage Requirement
    - OrderBased label size is on average 6.6% of LSDX label size

    - Com-D label size is on average 97% of OrderBased label size
- Labeling time
    - OrderBased is 2.2 to 3.9 times faster than LSDX
    - OrderBased is 17.2 to 51.8 times faster than Com-D
- Querying time
    - OrderBased is faster for large datasets than LSDX
    - OrderBased is 6.9 to 14.9 times faster than Com-D
- Update performance
    - Insertion time
        - OrderBased is on average 2 times faster than LSDX
        - OrderBased in on average 5 times faster than Com-D
    - Deletion time
        - OrderBased is 1.5 to 2 times faster than LSDX
        - OrderBased is 85 to 127 times faster than Com-D

In summary, performance evaluation studies show that OrderBased labeling scheme outperforms LSDX and Com-D with respect to labeling time, query performance, and update performance. It is also shown that the total label size for OrderBased labels from 91.1% to 91.95% smaller than label size of LSDX. Even though OrderBased label size is from 2.4% to 7.1% greater than that of Com-D, its efficient querying, labeling and update performance makes it preferable.

In future, a comprehensive research can be done to see if the rationales of OrderBased labeling scheme can be applied to other labeling schemes. Making an extended query performance comparison using real world datasets can also be considered as future work.

REFERENCES

1. S. Boag, D. Chamberlin, Mary, F. Fernandez, D.Florescu, J. Robie, and J. Simeon, "XQuery 1.0: An XML query language", W3C working draft. 2001.

2. J. Clarke and S. DeRose, "XML path language (XPath) version 1.0",W3C recommendation,1999.

3. P.F Diets "Maintaining Order in a Linked Lists", In Proceedings of the ACM Symposium on Theory of Computing, 1982.

4. Q. Li, and B. Moon,"Indexing and Querying XML Data for Regular Path Expressions", in proceedings of the VLDB, 2001.

5. Jung-Hee Y., Chin-Wan C. ,"Dynamic interval-based labeling scheme for efficient XML query and update processing", The Journal of Systems and Software,2008.

6. E.Cohen, H. Kaplan, T. ilo,"Labeling Dynamic XML Trees", in Proceedings of the ACM SIGMOD- SIGACT- SiGART,2002.

7. I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang "Storing and Querying Ordered XML Using a Relational Database System", In Proceedings of ACM SIGMOD, 2002.

8. ONeil, P.E. et al., "ORDPATHs: Insert-friendly XML node labels", In Proceedings of the ACM SIGMOD 2004.

9. M. Duong, and Y. Zhang, "LSDX: New Labeling Scheme for Dynamically Updating XML Data",In proceedings of 16th Australian Database Conference, 2005.

10. Duogn M, Zhang Y,"Dynamic Labeling Scheme for XML Data Processing", On the Move to Meaningful Internet Systems: OTM, 2008.

11. Schmidt A., Waas F., Kersten M., Carey, J., M.Manolescu I. and Busse, R. (2002): XMark: A Benchmark for XML Data Management, in Proceedings of VLDB ,2002.

12. J. Gray and A. Reuter, editors "Transaction Processing: Concepts and Techniques", Morgan Kaufmann, San Mateo, California, 1993.

13. C. Mohan and F. Levine. ARIES/IM,"An Efficient and High Concurrency Index Management Method using write-ahead logging", SIGMOD Record, 21(2):371–380, 1992.

14. M. A. Olson, K. Bostic, and M. I. Seltzer: Berkeley DB. In USENIX Annual, FREENIX Track, pages 183–191, 1999.

15. Li Y., Ma J., Sun Y." Applying Dewey Encoding to Construct XML Index of Path and Keyword Query", Proceedings of IEEE First International Workshop on Database Technology and Applications, 2009.

16. D.D.Kha, M. Yoshikawa, and S. Uemara, "A Structural Numbering Scheme for XML Data", in Lecture Notes in Computer Science 2490,2002.

17. L. Xu, Z. Bao, and T.W. Ling, "A Dynamic Labeling Scheme Using Vectors", Proceedings of the 18th International Conference on Database and Expert Systems Applications (DEXA), 2007.

18. Liang Xu, T. W. Ling, and Huayu Wu, "Labeling dynamic XML Documents: An Order-centric Approach", IEEE Transactions on Knowledge and Data Engineering, 2010.

19. X. Wu, M.L. Lee, W. Hsu, "A Prime Number Labeling Scheme for Dynamic Ordered XML Trees", In Proceedings of the 20th International Conference on Data Engineering, 2004.

20. A. Gabillon and M. Fansi " A Persistent Labeling Scheme for XML and tree Database", In Proceedings of ACI, 2006.

21. Damien K., F. Franky ,L. William M. Shui Raymond K.Wong, "Dynamic Labeling Schemes for Ordered XML Based on Type Information", Seventeenth Australasian Database Conference Technology, Vol. 49, 2006.

22. A. Silberstein et al,"BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data", In Proceedings of Inenrnational conference on Data Engineering (ICDE), IEEE CS Press, 2005.

23. R. Thonangi,"A Concise Labeling Scheme for XML Data", International Conference on Management of Data COMAD 2006, Delhi,India, 2006.

24. Liang X/ , Tok W. L. , Huayu W. , Z. Bao," DDE: From Dewey to a Fully Dynamic XML Labeling Scheme", Proceedings of the 35th SIGMOD International Conference on Management of Data, Providence, Rhode Island, USA, June 29-July 02, 2009.

25. J.H. Yun and C.W. Chung, "Dynamic Interval-Based labeling Scheme for Efficient XML Query and Update processing", Journal of Systems and Software, 81, p.p. 56-70, 2008.

26. S.C. Haw, and C.S. Lee, “Extending Path Summary and Region Encoding for Efficient Structural Query Processing in Native XML Databases”, Journal of Systems and Software, 2009.

27. C. Li, T. W. Ling, and M. Hu. “Efficient Updates in Dynamic XML Data: from Binary String to Quaternary String”. VLDB Journal, 2008.

28. Martin F. O’Connor Mark Roantree, “Desirable Properties for XML Update Mechanisms”, Updates in XML (EDBT Workshop Proceedings), Lausanne, Switzerland, 2010.

29. C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman, “On Supporting Containment Queries in Relational Database Management Systems” In SIGMOD, 2001.

30. T. Amagasa, M. Yoshikawa, and S. Uemura, “QRS: A Robust Numbering Scheme for XML Documents”, In ICDE, 2003.

31. M. Dewey, “Dewey Decimal Classification (DDC) system, http://www.oclc. org /dewey/

32. Li C., Ling T. W., Lu J., Yu T, “On Reducing Redundancy and Improving Efficiency of XML Labeling Schemes”, CIKM, 2005.

33. C. Li and T. W. Ling. An Improved Prefix Labeling Scheme: A Binary String Approach for Dynamic Ordered XML. In DASFAA, pages 125-137, 2005.

34. C. Li and T. W. Ling. QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. In CIKM, pages 501.508, 2005.

35. Haw Su-Cheng, Lee Chien-Sing, “Node Labeling Scheme in XML Query Optimization: Survey and Trend”, IETE Technical Review ,2009.

36. Martin F. O’Connor Mark Roantree, “Desirable Properties for XML Update Mechanisms”, Updates in XML (EDBT Workshop Proceedings, Lausanne, Switzerland, March 2010.

37. S. Subramaniam, Su-Cheng Haw, P. K. Hoong “XML Labeling Schemes for Dynamic Updates: Strengths and Limitations”, on the Proceeding of the International Conference on Advanced Science, Engineering and Information Technology, 2011.

38. C. Li, T. W. Ling, and M. Hu. “Efficient Processing of Updates in Dynamic XML Data”. In ICDE, page 13, 2006.

39. Fomichev A., Grinev, M., and Kuznetsov, “ Sedna: A native XML DBMS”, In 32nd Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2006.

40. H. V. Jagadish , S. Al-Khalifa , A. Chapman , L. V. S. Lakshmanan , A. Nierman , S. Paparizos , J. M. Patel , D. Srivastava , N. Wiwatwattana , Y. Wu , C. Yu, "TIMBER: A native XML database', The VLDB Journal The International Journal on Very Large Data Bases, , December 2002.

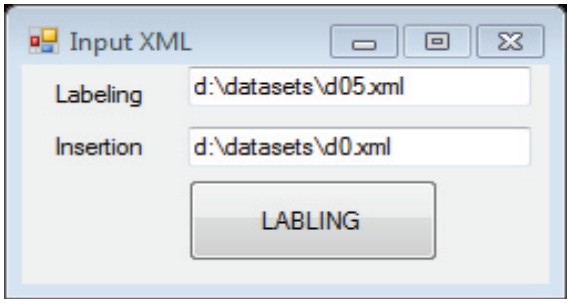
41. Sh. Pal , I. Cseri , O. Seeliger , M. Rys , G. Schaller , Wei Yu , D. Tomic , Adrian Baras , Brandon B., Denis Ch. , Eugene K. , "XQuery implementation in a relational database system", Proceedings of the 31st international conference on Very large data bases, August 30-September 02, 2005.

42. Nicola, M. and van der Linden, B. 2005. Native XML support in DB2 universal database. In *Proceedings of the 31st international Conference on Very Large Data Bases* (Trondheim, Norway, August 30 - September 02, 2005). Very Large Data Bases. VLDB Endowment, 1164-1174. .

43. Guangjun X., Qi Cheng, Jarek G.,Calisto Z, "Some Rewrite Optimizations of DB2 XQuery Navigation", CIKM'08,2008.

APPENDIX A

LABELING SCHEME IMPLEMENTATION



The input XML form is the first interface appearing while running the implementation. The txtlabel and txtinsert text fields are used to enter the location of the XML document to be labeled and inserted respectively.

```
String d=txtlabel.Text
String dinsert= txtinsert.Text
Dim document As Xml.XPath.XPathDocument = New
Xml.XPath.XPathDocument(d)

    Dim navigator As Xml.XPath.XPathNavigator =
document.CreateNavigator
    Dim nodes As Xml.XPath.XPathNodeIterator = navigator.Select("//*")
    Dim documentinsert As Xml.XPath.XPathDocument = New
Xml.XPath.XPathDocument(d_insert)
    Dim navigatorinsert As Xml.XPath.XPathNavigator =
documentinsert.CreateNavigator
    Dim nodes2 As Xml.XPath.XPathNodeIterator =
navigatorinsert.Select("//*")
```

A.1. LSDX Labeling Scheme Code

The LSDX labeling scheme accepts two parameters: My-tree which is an instance of a B+ tree class, and str is the initial label for LSDX i.e . in this case it is ‘a’.

```
Private Sub LSDX_Labeling(ByRef My_tree As BTree(Of Label), ByVal str
As String)
        Dim s As New Stack
        Dim h As Integer = 0
        Dim m As Integer = 0
        Dim k As Integer
        Dim r As String

        navigator.MoveToRoot()
```

```
        navigator.MoveToFirstChild()
        s.Push(str)
        My_tree.AddItem(h.ToString("00") & "," & s.Peek,
navigator.Name)
1:      While navigator.HasChildren
            navigator.MoveToFirstChild()
            h = h + 1
            s.Push(Replace(s.Peek, ".", "") & ".b")
            If navigator.Name.ToString.Length > 0 Then
                My_tree.AddItem(h.ToString("00") & "," & s.Peek,
navigator.Name)
            End If
            m = IIf(h > m, h, m)
        End While
2:      While navigator.MoveToNext
            If navigator.Name.Length > 0 Then
                r = s.Pop
                k = r.IndexOf(".") + 1
                s.Push(Mid(r, 1, k) & NextString(Mid(r, k + 1, Len(r)
- k)))
                My_tree.AddItem(h.ToString("00") & "," & s.Peek,
navigator.Name.ToString)

                If navigator.HasChildren Then GoTo 1
            End If
        End While
        If navigator.MoveToParent Then
            h = h - 1
            s.Pop()
            GoTo 2
        End If
    End Sub
```

A.2. OrderBased Labeling Scheme Code

For OrderBased labeling scheme, it first runs InitializeDyanamicOrder_ Basedlabeling () before labeling the XML tree. This routine does the function as the Determine-Size routine discussed on chapter three, section 3.1.

A.2.1. Determine_Size routine

```
Private Sub InitializeDyanamicOrderBasedlabeling()
        Dim h As Integer = 0
        Dim xm As Integer = 0
        c.Clear()
        c.Add(0)
        navigator.MoveToRoot()
        navigator.MoveToChild(Xml.XPath.XPathNodeType.All)

1:      While navigator.HasChildren
            navigator.MoveToFirstChild()
            h = h + 1
            If navigator.Name.Length > 0 Then
                If h = c.Count Then
                    c.Add(1)
                    xm = h
                Else
```

```
                    c.Insert(h, c(h) + 1)
                    c.RemoveAt(h + 1)
                End If
            End If
            If h > xm Then
                c.Add(0)
                xm = h
            End If
        End While
2:      While navigator.MoveToNext
            If navigator.Name.Length > 0 Then
                c.Insert(h, c(h) + 1)
                c.RemoveAt(h + 1)
                If navigator.HasChildren Then GoTo 1
            End If
            GoTo 1
        End While
        If navigator.MoveToParent Then
            h = h - 1
            GoTo 2
        End If

        If c(xm) = 0 Then
            c.RemoveAt(xm)
            xm -= 1
        End If
        max = xm
        ReDim f(max)
        f(0) = "a"
        Dim m As Integer
        For j = 1 To max
            m = IIf(c(j) = 1, 1, Math.Ceiling(Math.Log10(c(j)) /
Math.Log10(25)))
            For g = 1 To m
                f(j) &= "b"
            Next
        Next j

    End Sub
```

A.2.2. OrderBased Labeling Scheme

```
Private Sub OrderBased_Labeling(ByRef My_tree As BTree(Of Label))
        InitializeDyanamicOrderBasedlabeling()
        Dim h, maxh As Integer
        h = 0
        maxh = 0
        navigator.MoveToRoot()
        navigator.MoveToFirstChild()
        My_tree.AddItem(h.ToString("00") & "," & String.Empty & "." &
f(h),    navigator.Name.ToString)
        'lstorderbased.Items.Add(h.ToString("00") & "," & String.Empty
& "." & f(h) & "-" & navigator.Name.ToString)
1:      While navigator.HasChildren
            navigator.MoveToFirstChild()
            h += 1
            If Len(navigator.Name) > 0 Then
                If maxh >= h Then
                    f(h) = nextOrderlabel(f(h))
                End If
                My_tree.AddItem(h.ToString("00") & "," & f(h - 1) &
"." & f(h), navigator.Name.ToString)

            End If
```

```
            maxh = IIf(h > maxh, h, maxh)
        End While
2:      While navigator.MoveToNext
            If navigator.Name.Length > 0 Then
                f(h) = nextOrderlabel(f(h))
                My_tree.AddItem(h.ToString("00") & "," & f(h - 1) &
"." & f(h), navigator.Name.ToString)
               If navigator.HasChildren Then GoTo 1
            End If
        End While
        If navigator.MoveToParent Then
            h -= 1
            GoTo 2
        End If

    End Sub
```

APPENDIX B

STRUCTURAL RELATIONSHIPS

B.1. Parent Child Relationships

B.1.1. Find LSDX Parent

```
Public Function findlsdxparent(ByVal target_key As String)
        Dim d As BTreeNode(Of T) = Me
        Dim x As String = Mid(target_key, 1, target_key.IndexOf(","))
        Dim y As String = (CInt(x) - 1).ToString("00")
        If y < 0 Then Return String.Empty
        Dim a As Integer = target_key.IndexOf(",")
        Dim b As Integer = target_key.IndexOf(".")
        Dim parent_key As String = y & Mid(target_key, a + 1, b - a)
        Dim spot As Integer = 0
        Do While d.isleaf = False
            While spot < d.NumKeysUsed
                If Replace(d.Keys(spot), ".", "") >= parent_key Then Exit
While
                spot += 1
            End While
            d = d.Children(spot)
            spot = 0
        Loop
        For spot = 0 To d.NumKeysUsed - 1
            If Replace(d.Keys(spot), ".", "") >= parent_key Then
                If Replace(d.Keys(spot), ".", "") = parent_key Then
                    parent_key = d.Keys(spot)
                    Exit For
                Else
                    Return String.Empty
                End If
            End If
        Next
        Return parent_key
    End Function
```

B.1.2. Find Com-D Parent

```
Public Function findcomplsdxparent(ByVal target_key As String)

        Dim d As BTreeNode(Of T) = Me

        Dim x As String = Mid(target_key, 1, target_key.IndexOf(","))

        Dim y As String = (CInt(x) - 1).ToString("00")

        If y < 0 Then Return String.Empty
```

```
        Dim a As Integer = target_key.IndexOf(",")
        Dim b As Integer = target_key.IndexOf(".")
        Dim parent_key As String = y & uncompress(Mid(target_key, a +
1, b - a))
        Dim spot As Integer = 0
        Do While d.isleaf = False
            While spot < d.NumKeysUsed
                If Replace(d.Keys(spot), ".", "") >= parent_key Then
Exit While
                spot += 1
            End While
            d = d.Children(spot)
            spot = 0
        Loop
        For spot = 0 To d.NumKeysUsed - 1
            If   Replace(uncompress(d.Keys(spot)),   ".",   "")   >=
parent_key Then
                If   Replace(uncompress(d.Keys(spot)),   ".",   "")   =
parent_key Then
                    parent_key = d.Keys(spot)
                    Exit For
                Else
                    Return String.Empty
                End If
            End If
        Next
        Return parent_key
    End Function
```

B.1.3. Find OrderBased Parent

```
Public Function findorderbasedparent(ByVal target_key As String)
        Dim d As BTreeNode(Of T) = Me
        Dim x As String = Mid(target_key, 1, target_key.IndexOf(","))
        Dim y As String = (CInt(x) - 1).ToString("00")
        If y < 0 Then Return String.Empty
        Dim a As Integer = target_key.IndexOf(",") + 1
        Dim b As Integer = target_key.IndexOf(".")
        Dim parent_key As String = "." & Mid(target_key, a + 1, b - a)
        Dim spot As Integer = 0
        While d.isleaf = False
            Do While spot < d.NumKeysUsed
                Dim t As String = Mid(d.Keys(spot), 1,
d.Keys(spot).IndexOf(","))
                If t >= y Then
                    If t = y Then
                        Exit While
                    End If
                    Exit Do
                End If
                spot += 1
            Loop
            d = d.Children(spot)
            spot = 0
        End While
        While d.isleaf = False
            Do While spot < d.NumKeysUsed
```

```
                If Mid(d.Keys(spot), 1, d.Keys(spot).IndexOf(",")) = y
Then

                    a = d.Keys(spot).IndexOf(".")

                    Dim    t    As    String    =    Right(d.Keys(spot),
d.Keys(spot).Length - a)

                    If Right(d.Keys(spot), d.Keys(spot).Length - a) >=
parent_key Then

                        Exit Do

                    End If

                Else

                    Exit Do

                End If

                spot += 1

            Loop

            d = d.Children(spot)

            spot = 0

        End While


        For spot = 0 To d.NumKeysUsed - 1


            If  Mid(d.Keys(spot),  1,  d.Keys(spot).IndexOf(","))  =  y
Then

                a = d.Keys(spot).IndexOf(".")

                If  Right(d.Keys(spot),  d.Keys(spot).Length  -  a)  =
parent_key Then

                    parent_key = d.Keys(spot)

                    Exit For

                End If

            End If
```

```
    Next

    Return parent_key

End Function
```