View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarBank@NUS

LABELING DYNAMIC XML DOCUMENTS: AN ORDER-CENTRIC APPROACH

XU LIANG

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2010

Acknowledgments

I express my sincere appreciation to my advisor Prof. Ling Tok Wang for his guidance and insight throughout the research, without which, I never would have made it through the graduate school. It has been five years since I became a student of Prof. Ling when I started my honor year project. During the time, Prof. Ling taught me how to think critically, ask questions and express ideas. His advice and help are invaluable to me and I will remember them for the rest of my life.

Special thanks go to my thesis evaluators, Assoc. Prof. Stephane Bressan and Prof. Chan Chee Yong, for their valuable suggestions, discussions and comments. They have helped me since the very early stage of my works.

I want to say "thank you" to my seniors, Dr. Changqing Li and Dr. Jiaheng Lu, for their selfless help to me, and for always being there to answer my questions.

I am thankful to all colleagues and friends who have made my stay at the university a memorable and valuable experience. I will cherish all the good memories we shared together.

I am deeply indebted to my mother for the unconditional support and encouragement I received, which helped me go through the most difficult times of my study. Words alone cannot express my gratitude to her.

Abstract

Labeling Dynamic XML Documents:
An Order-Centric Approach

Xu Liang

The rise of XML as a de facto standard for data exchange and representation has generated a lot of interest on querying XML documents that conform to an *ordered tree-structured* data model. Labeling schemes facilitate XML query processing by assigning each node in the XML tree a unique label[8, 22, 35, 44, 51]. Structural relationships of the tree nodes, such as Parent/Child (PC), Ancestor/Descendant (AD), Sibling and Document order, can be efficiently established by comparing their labels.

In this thesis, we explore static and dynamic XML labeling schemes from a novel *order-centric* perspective: We systematically study the various labeling schemes proposed in the literature with a special focus on their orders of labels. We develop an order-based framework to classify and characterize XML labeling schemes, based on which we show that the order of labels fundamentally impacts the update processing of a labeling scheme[48].

We introduce a novel order concept, vector order[46], which is the foundation of the dynamic labeling schemes we propose. Compared with previous solutions that are based on natural order, lexicographical order or VLEI order[9, 22, 32–35, 38, 44, 51], vector order is a simple, yet most effective solution to process updates

in XML DBMS. We illustrate the application of vector order to both range-based and prefix-based labeling schemes, including Pre/post[22], Containment[51] and Dewey labeling schemes[44] to efficiently process updates without re-labeling.

Since updates are usually unpredictable, we argue that a single labeling scheme should be used for both static and dynamic XML documents. Previous dynamic XML labeling schemes, however, suffer from the complexity introduced by their insertion techniques even if there is little/no update. To further improve the application of vector order to prefix-based labeling schemes, we extend the concept of vector order and introduce Dynamic DEwey (DDE) labeling scheme[49]. DDE, in the static setting, is the same as Dewey labeling scheme which is designed for static XML documents. In addition, based on an extension of vector order, DDE allows dynamic updates without re-labeling when updates take place. We introduce a variant of DDE, namely CDDE, which is derived from DDE labeling scheme from a one-to-one mapping. Compared with DDE, CDDE labeling scheme shows slower growth in label size for frequent insertions. Both DDE and CDDE have exhibited high resilience to skewed insertions in which case the qualities of existing labeling schemes degrade severely. Qualitative and experimental evaluations confirm the benefits of our approach compared to previous solutions.

Lastly, we focus on improving the efficiency of applying vector order to range-based labeling schemes[47]. We present in this thesis a generally applicable Search Tree-based (ST) encoding technique which can be applied to vector order as well as existing encoding schemes[32–34]. We illustrate the applications of ST encoding technique and show that it can generate dynamic labels of optimal size. In addition, when combining with encoding table compression, we are able to process very large XML documents with limited memory available. Experimental results demonstrate the advantages of our encoding technique over the previous encoding algorithms.

Contents

1 Introduction **8**

1.1 Background . . . 8

1.1.1 Overview of XML and Related Technologies . . . 8

1.1.2 XML Data Model and Queries . . . 9

1.2 Research Problem . . . 12

1.2.1 XML Shredding . . . 12

1.2.2 XML Labeling Schemes . . . 13

1.3 Summary of Contributions . . . 15

1.4 Thesis organization . . . 16

2 Related work from an order-centric perspective **18**

2.1 Labeling tree-structured data . . . 18

2.1.1 Range-based labeling schemes . . . 19

2.1.2 Prefix-based labeling schemes . . . 20

2.1.3 Prime labeling scheme . . . 21

2.2 Order encoding and update processing . . . 22

2.2.1 Range-based labeling schemes and natural order . . . 23

2.2.2 Prefix-based labeling schemes and lexicographical order . . . 24

2.2.3 Transforming natural order to lexicographical order . . . 25

2.2.4 Transforming lexicographical order to generalized lexicographical order . . . 27
2.3 Summary of chapter . . . 30

3 Vector order and its applications **31**
3.1 Vector code ordering . . . 31
3.2 Vector code functions . . . 34
3.3 Applications of vector order . . . 36
3.3.1 Order-preserving transformation . . . 37
3.3.2 V-Containment labeling scheme . . . 39
3.3.3 V-Pre/post labeling scheme . . . 41
3.3.4 V-Prefix labeling scheme . . . 43
3.4 Summary of chapter . . . 48

4 Extension of vector order and its applications **49**
4.1 DDE labeling scheme . . . 49
4.1.1 Motivation . . . 49
4.1.2 Initial Labeling . . . 50
4.1.3 DDE label ordering . . . 51
4.1.4 DDE label properties . . . 52
4.1.5 Correctness of initial labeling . . . 53
4.1.6 DDE label addition . . . 54
4.1.7 Processing updates . . . 56
4.1.8 Correctness . . . 57
4.2 Compact DDE (CDDE) . . . 58
4.2.1 Initial labeling . . . 59
4.2.2 CDDE label to DDE label mapping . . . 59

4.2.3 CDDE label addition 61
4.2.4 Processing updates 62
4.3 Relationship computation 65
4.3.1 DDE labels 65
4.3.2 CDDE labels 66
4.4 Qualitative comparison 68
4.5 Experiments and results 70
4.5.1 Experimental setup 70
4.5.2 Initial labeling 70
4.5.3 Querying static document 71
4.5.4 Update processing 72
4.5.5 Querying dynamic document 75
4.6 Summary of chapter 76

5 Search Tree-based (ST) encoding techniques for range-based labeling schemes 77

5.1 Insertion-based encoding algorithms 77
5.2 Dynamic Formats 80
5.2.1 Binary strings 81
5.2.2 Quaternary strings 81
5.3 ST Encoding Technique 82
5.3.1 Seach Tree-based Binary (STB) encoding 83
5.3.2 Seach Tree-based Quaternary (STQ) encoding 86
5.3.3 Search Tree-based Vector (STV) encoding 89
5.3.4 Comparison with insertion-based approach 90
5.4 Encoding Table Compression 90
5.5 Tree Partitioning (TP) 92

5.6 Experiments and Results 95
5.6.1 Encoding Time 95
5.6.2 Memory Usage and Encoding Table Compression 96
5.6.3 Label size and query performance 97
5.7 Summary of chapter 98

6 Conclusion **99**

6.1 Summary of order-centric approach 99
6.2 Future work 103

List of Tables

1.1 Shredding XML data into node relational table 12

2.1 Summary of related work (lex is short for lexicographical) 30

3.1 Linear and recursive transformation for the range [1,18] 37

4.1 Test data sets 70

5.1 Test data sets 95

6.1 Summary of orders of different labeling schemes 100

List of Figures

1.1 A sample XML document 10

1.2 A sample XML tree 11

2.1 Range-based labeling schemes 19

2.2 Dewey labeling scheme 20

2.3 ORDPATH labeling scheme 27

3.1 Graphical representation of vector codes 32

3.2 Vector code addition and multiplication 35

3.3 Process Updates with V-Containment labeling scheme 41

3.4 Process Updates with V-Pre/post labeling scheme 42

3.5 Process Updates with V-Prefix labeling scheme 44

4.1 Processing insertions with DDE labels 55

4.2 Processing insertions with CDDE labels 63

4.3 DDE labeling after uniform insertion 66

4.4 Initial Labeling 71

4.5 Querying initial labels 72

4.6 Uniform insertions 73

4.7 Comparison of prefix-based labeling schemes after skewed insertions 74

4.8 Relationship computation time after skewed insertions 74

4.9 Comparison of range-based labeling schemes after skewed insertions 75

5.1 Applying QED encoding scheme to containment labeling scheme . . 78

5.2 STB encoding of two ranges 6 and 12 84

5.3 STQ Encoding of two ranges 6 and 12 87

5.4 STV tree 89

5.5 Compress L tables of STB and STQ by factors of 2^C and 2×3^C respectively 91

5.6 Tree partitioning 93

5.7 Encoding containment labels of multiple documents 96

5.8 Encoding table compression 97

Chapter 1

Introduction

We begin by introducing the background of eXtensible Markup Language (XML)[12] in Section 1.1. The main research problem is presented in Section 1.2 followed by the summary of our contributions in Section 1.3 and thesis organization in Section 1.4.

1.1 Background

In this section, we present the background of our research problem.

1.1.1 Overview of XML and Related Technologies

Standard Generalized Markup Language (SGML) is a standard which defines generalized markup languages for documents and has been widely used in certain high-end areas of information management and publishing, such as authoring technical documentation[1] and electronic data-gathering, analysis and retrieval[2]. By limiting SGML to a specific vocabulary of tags, Hypertext Markup Language (HTML) allows ease of use and has become the predominant markup language for web pages. Similar to HTML, the eXtensible Markup Language (XML) is a simplified subset of

SGML. However, unlike HTML which focuses on displaying and formatting data, XML is designed to capture the actual meaning and structure of the underline data. Fueled by the hope to make information self-describing and following the recommendation of the World Wide Web Consortium (W_3C), XML has quickly spread over the Web and elsewhere as a standard to exchange and represent data.

An XML document must begin with a *prolog* specifying the XML version being used and possibly some additional information. The basic logical component of XML data is an *element* which is identified by *tags*. An element can either consist of a pair of start and end tags or an empty element tag (if it does not have any sub-elements or values). Additional information about an element can be specified as *attributes* which can be included in the start tag or empty element tag.

Example 1.1: Figure 1.1 presents a simple sample XML document. The prolog of the document (line 1) declares that the it conforms to XML version 1.0 and characters are encoded with UTF-8 encoding scheme. The root element of the document is BOOK whose start and end tags (<BOOK> and < /BOOK>) can be found at line 2 and 13 respectively. Inside the start tag of BOOK, ISBN is an attribute with value 1-23456-789-0. Line 3 and 7 are the start and end tags of an element SECTION which encloses an element TITLE (line 4), a sequence of characters "W3C standard" (line 5) and an element Figure with empty element tag and an attribute CAPTION (line 6). Line 8 to 12 is another SECTION element with similar structure. □

1.1.2 XML Data Model and Queries

XML documents are commonly modeled as trees[5]. For example, the XML document in Figure 1.1 can be viewed as the tree in Figure 1.2. Some values are not represented because they are directly associated with an element or an at-

```
1<?xml version=1.0 encoding=UTF-8 ?>
2  <BOOK ISBN=1-23456-789-0>
3      <SECTION>
4          <TITLE> SGML </TITLE>
5              "W3C standard"
6          <FIGURE CAPTION="Standard Generalized Markup Language/>
7      </SECTION>
8      <SECTION>
9          <TITLE> XML </TITLE>
10             "W3C recommendation"
11         <FIGURE CAPTION=eXtensible Markup Language/>
12     </SECTION>
13 </BOOK>
```

Figure 1.1: A sample XML document

tribute. Processing XML documents of more complex models such as graph-based [23, 25, 36] is beyond the scope of this thesis.

A salient feature of XML data is its order. The elements in an XML document are implicitly ordered by the order in which their start tags are encountered when the document that contains them is parsed, which we refer to as *document order*. As the tree-structure is concerned, document order is equivalent to the pre-order defined on nodes. This is illustrated in Figure 1.2 where each node is associated with an integer indicating its order.

Several query languages, such as Lorel[7], Quilt[15], XML-QL[21], XML-GL[13], XPath[18] and XQuery[14], have been proposed to query XML and semi-structured data. Following is an example of XPath query.

Q1: /BOOK/SECTION//CAPTION

The XPath query can be interpreted as a sequence of steps separated by '/' or '//' which indicate direct containment (Parent/Child) and general containment (Ancestor/Descendant) relationships respectively. The evaluation of the query can be processed step by step, with each step applying to the result set of elements

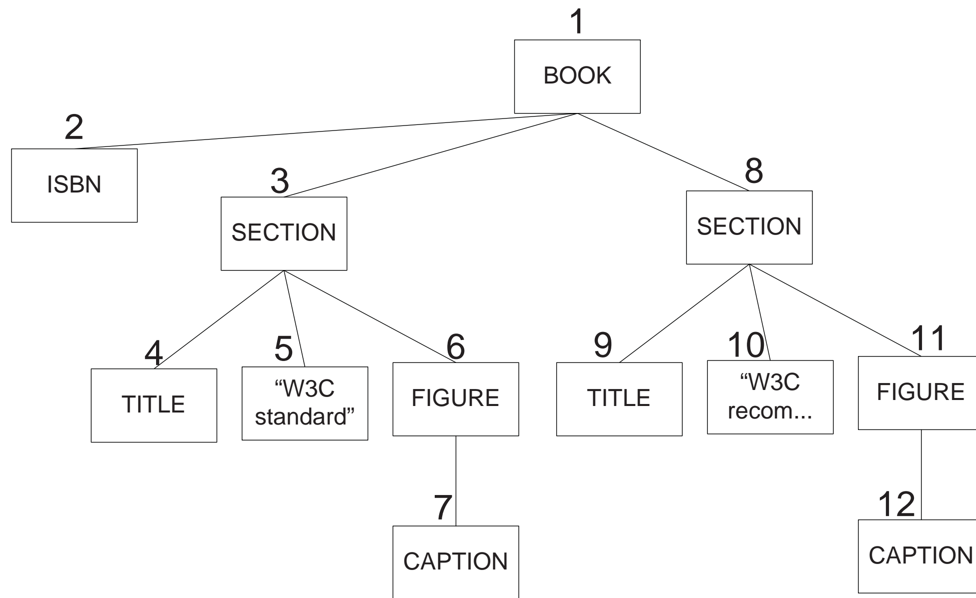


Figure 1.2: A sample XML tree

returned by the previous step. "/BOOK" evaluates to the root element with tag "BOOK", to which we apply "/SECTION" and evaluate to the set of elements with tag "SECTION" directly contained in BOOK element. By further applying "//CAPTION", the result would be the set of element nodes with tag "CAPTION" anywhere under SECTION elements in the previous result.

Document order has to be taken into consideration when evaluating XPath queries. For example, the elements in the result of a XPath query should be sorted by document order. In addition, there are XPath queries with predicates (statements inside square brackets) that explicitly make use of document order, which we illustrate with the following examples.

Q2: /BOOK/SECTION[position=2]

Q2 retrieves the second element with tag "SECTION" directly under BOOK element.

Q3: /BOOK//TITLE[3 to 6]

Label	TAG	NODE TYPE	VALUE
1	BOOK	Element	null
2	ISBN	Attribute	1-23456-780-0
3	SECTION	Element	null
4	TITLE	Element	“SGML”
5	–	Value	“W3C standard”
6	FIGURE	Element	null
7	CAPTION	Attribute	“Standard Generalized Markup Language”
8	SECTION	Element	null
9	TITLE	Element	“XML”
10	–	Value	“W3C recommendation”
11	FIGURE	Element	null
12	CAPTION	Attribute	“eXtensible Markup Language”

Table 1.1: Shredding XML data into node relational table

Q3 retrieves the third, forth, fifth and sixth elements with tag “TITLE” anywhere under the BOOK element in document order.

In summary, both tree structure and document order in XML data contain rich information that queries can exploit.

1.2 Research Problem

This thesis focuses on the problem of designing dynamic XML labeling schemes, which initially arises from a so-called “shredding” process that transforms XML data for relational storage. However, in addition to relational storage, it is worth noting that labeling schemes are useful for storage and indexing in general.

1.2.1 XML Shredding

Many solutions to store and query XML data are built on top of relational databases [10, 26, 37, 44, 51]. By transforming XML data through a “shredding” process[26, 39, 44], the result is a *node relational table*[38] that fits into relational database storage. An example of node relational table is shown in Table 1.1 which is the

result of shredding the XML document in Figure 1.1.

Each element (or a value) in Figure 1.1 is mapped into one row in Table 1.1. The tag, node type and value of the element are stored in the second, third and forth columns respectively. The first column "Label" serves as a logical identifier of that element. We refer to the assignment of labels in a node relational table as a *labeling scheme*. A labeling scheme is "lossless" if we can reconstruct the XML tree from the node relational table based on the labels. The example node relational table in which document orders are used as labels is NOT lossless because we lose structural information, such as Ancestor/Descendant, Parent/Child relationships, Sibling and Document order, necessary for the reconstruction of the XML tree. As a result, the resulting node relational table cannot provide full support for XML queries.

1.2.2 XML Labeling Schemes

As we have seen in the previous section, a lossless labeling scheme is the key to map unordered node relational table to ordered tree-structured XML data. Existing labeling schemes can be mainly classified into two families: range-based[8, 22, 35, 51] and prefix-based[6, 20, 30, 38, 44]. In this thesis, we consider the problem of designing labeling schemes in a dynamic environment where elements can be arbitrarily inserted/deleted from the XML documents. Under this setting, the following criteria are important for evaluating a labeling scheme:

1. Order and structural information. Documents obeying XML standard are intrinsically ordered and typically modeled as trees. Labeling schemes encode both document order and structural information so that queries can exploit them. While document order is essential to be encoded, the amount of structural information contained in the labels may vary. For example, sib-

ling relationship can be derived from prefix-based labeling schemes, but in general not from range-based labeling schemes.

2. Query efficiency. Deriving structural information, including Ancestor/Descendant, Parent/Child relationships, Sibling and Document order, from labels should be as efficient as possible.

3. Update efficiency. It is desirable to have a persistent labeling scheme, i.e. update operations performed on XML documents (such as insertions, deletions and modifications) should not require existing labels to be re-labeled. This is crucial for low update costs and for the users to be able to query the changes of the XML data over time[20].

4. Size. Size is an important factor that contributes to query and update efficiency.

However, designing labeling schemes that fulfill all these criteria turns out to be a challenging problem. Most early works[6, 8, 9, 20, 22, 30, 35, 44, 51] on labeling schemes can not satisfy the third criteria and requires re-labeling when updating the XML documents. More dynamic solutions[32–34, 38, 42, 45] have been proposed, however at the cost of lower query performance and less compact size even for XML documents that are seldom updated.

Given the extensive research on this topic, our first objective is to compare and characterize the various labeling schemes proposed in the literature under a unified framework. Establishing such a framework provides insight into the update behavior of existing labeling schemes as well as demonstrating the novelty of our proposed approach.

Moreover, we argue that a single labeling scheme should be designed to fit both static and dynamic labeling scheme. If different labeling schemes were to be used

for static and dynamic XML documents, different storage and query mechanisms need to be enforced, making updating and querying complicated. To make matters worse, deciding whether a document is static or dynamic in general is a difficult, if not impossible task as the updating frequency of a document can vary according to time: a document can, for example, be frequently updated for a period of time and remain unchanged after that.

1.3 Summary of Contributions

The contribution of this thesis is summarized as follows.

- Designing dynamic XML labeling schemes have received extensive research attention. In this thesis, we analyze the various labeling schemes proposed in the literature with a special focus on their orders of labels. We develop an order-based framework to classify and characterize XML labeling schemes. Based on which, we show that the order of labels fundamentally impacts the update processing of a labeling scheme.

- Different from previous labeling schemes are based on natural order[9, 22, 35, 44, 45, 51], lexicographical order[32–34, 38] or Variable Length Endless Insertable (VLEI) order[31], we introduce a novel order concept, vector order, which is the foundation of the labeling schemes propose. We illustrate the application of vector order to both range-based and prefix-based labeling schemes.

- To improve the application of vector order to prefix-based labeling schemes, we extend the concept of vector order and introduce Dynamic DEwey (DDE) labeling scheme which is tailored for static XML documents, while being dynamic enough to avoid re-labeling. A variant of DDE, Compact DDE

(CDDE), is also proposed to enhance the performance of DDE for frequent insertions.

- Vector order-based labeling schemes not only exhibit high resilience against frequent updates, but also outperforms previous labeling schemes in terms of query efficiency and size. Both qualitative and experimental comparisons demonstrate the advantages of our labeling schemes over the previous approaches.

- We propose a generally applicable Search Tree-based (ST) encoding technique. We show that ST encoding can be applied to existing encoding schemes to efficiently generate dynamic XML labels. We illustrate the applications of ST encoding technique to different dynamic formats and prove the optimality of our results. Experimental results demonstrate the high efficiency and scalability of our ST encoding techniques.

1.4 Thesis organization

This thesis is organized as follows.

In chapter 2, we systematically introduce related works with a special focus on their order of labels. An order-centric framework is established to facilitate convenient comparison of these works. Limitations of related works are presented which is the motivation of our work.

We introduce vector order in chapter 3 which represents a new approach to process updates in XML data. We illustrate how vector order can be applied to both range-based and prefix-based labeling schemes.

To improve the application of vector order to prefix-based labeling scheme, we extend the concept of vector order and introduce Dynamic DEwey labeling scheme

in chapter 4. A variant of DDE, namely CDDE which is designed for frequent insertion. Qualitative and experimental evaluations are presented to show the advantages of our proposed labeling schemes.

In chapter 5, we focus on order preserving transformation of the encoding approach. We introduce Search Tree-based (ST) encoding technique which outperforms existing encoding algorithms in terms of scalability and efficiency.

The thesis is concluded in chapter 6.

Some of the materials in this thesis are published in [46–49]. More specifically, Chapter 3 is published in [46], Chapter 4 is published in [49], Chapter 5 is published in [47] and the order-centric approach of the work is published in [48].

Chapter 2

Related work from an order-centric perspective

In this chapter, we present an order-centric study of existing works on labeling dynamic XML documents, where the orders of labels is our main focus. Our study is divided into two parts: (a) how tree structures are encoded (Section 2.1) and (b) how orders are encoded (Section 2.2). In Section 2.1, we introduce three families of labeling schemes, range-based, prefix-based and prime with focus on how they encode tree structure. In Section 2.2, the focus of our analysis is the orders of labels in static XML labeling scheme differ from those in dynamic XML labeling schemes, and why order affects their update processing.

2.1 Labeling tree-structured data

We begin by introducing how existing labeling schemes encode tree structures into compact labels.

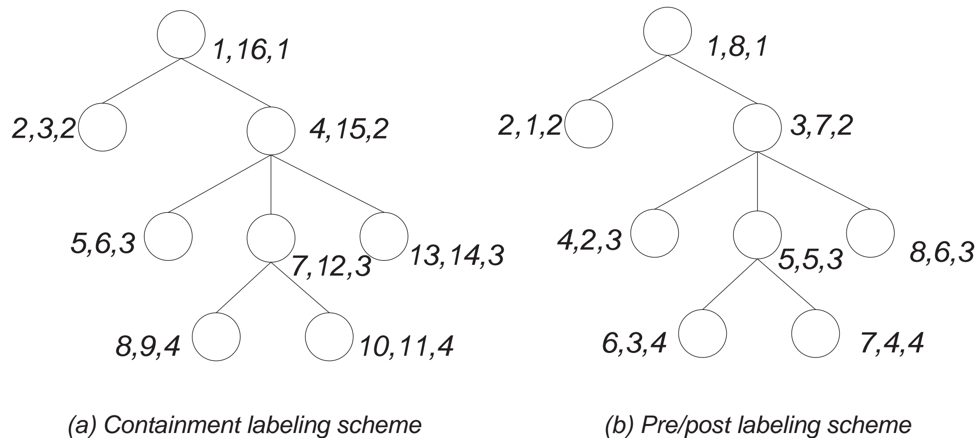


Figure 2.1: Range-based labeling schemes

2.1.1 Range-based labeling schemes

In Figure 2.1, we present examples of containment[51] and pre/post[22] labeling schemes which both belong to range-based labeling schemes.

In containment labeling scheme, each element node is assigned a label of the form *start*, *end*, *level* where *start* and *end* define a range that contains all its descendant's ranges. Each label in pre/post labeling scheme is of the form *pre*, *post*, *level* where *pre* and *post* are the ordinal numbers of the element node in preorder and postorder traversal sequences respectively. For both labeling schemes, *level* represents the level of the element node in the XML tree. Assume the level of the root is 1.

Given two containment labels $A(s_1, e_1, l_1)$ and $B(s_2, e_2, l_2)$, the following structural information can be derived:

P1 Ancestor/Descendant(AD). A is an ancestor of B if and only if $s_1 < s_2 < e_2 < e_1$, which can be simplified as $s_1 < s_2 < e_1$. The simplification is based on the observation that it is impossible to have $s_1 < s_2 < e_1 < e_2$ which implies the elements are not properly nested.

P2 Parent/Child(PC). A is the parent of B if and only if A is an ancestor of B

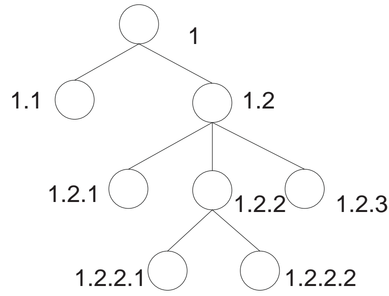


Figure 2.2: Dewey labeling scheme

and $l_1 = l_2 - 1$.

Both AD and PC relationships can be derived from pre/post labels as well. Here we highlight the following difference:

- Given two pre/post labels $A(pre_1,\ post_1,\ l_1)$ and $B(pre_2,\ post_2,\ l_2)$, A is an ancestor of B if and only if $pre_1 < pre_2$ and $post_2 < post_1$. This condition is different from that of containment labeling scheme and can not be similarly simplified.

Example 2.1: In Figure 2.1 (a), (4,15,2) is an ancestor of (8,9,4) because $4 < 8 < 15$. (7,12,3) is the parent of (8,9,4) because $7 < 8 < 12$ and 3=4-1. In Figure 2.1 (b), $3, 7, 2$ is an ancestor of $6, 3, 4$ because $3 < 6$ and $3 < 7$. □

In order/size labeling scheme[35], each label consists of a triplet *order*, *size*, *level*. order/size labeling scheme can be seen as a variation of containment labeling scheme where a range is defined by *order* and $(order + size)$.

2.1.2 Prefix-based labeling schemes

Figure 2.2 shows an example of Dewey labeling scheme[**?**], which is the representative of prefix-based labeling schemes. The order that Dewey labeling scheme makes heavy use of is the order among siblings, which we refer to as *local order*.

By concatenating the label of its parent (*parent_label*) with its own local order, a Dewey label uniquely identifies a path from the root to an element.

Given two Dewey labels $A : a_1.a_2 \ldots a_m$ and $B : b_1.b_2 \ldots b_n$, the following rules can be used to derive structural information from them:

- P1 Ancestor/Descendant(AD). A is an ancestor of B if and only if $m < n$ and $a_1 = b_1, a_2 = b_2, \ldots, a_m = b_m$.
- P2 Parent/Child(PC). A is the parent of B if and only if and only if A is an ancestor of B and $m = n - 1$
- P3 Sibling. A is the sibling of B if and only if $m = n$ and $a_1 = b_1, a_2 = b_2, \ldots, a_{m-1} = b_{m-1}$, i.e. A's *parent_label* matches B's *parent_label*.
- P4 Lowest Common Ancestor (LCA). The LCA of A and B is $C : c_1.c_2 \ldots c_l$ such that C is an ancestor of both A and B and either (1) $l = min(m, n)$ or (2) $a_{l+1} \neq b_{l+1}$.

Example 2.2: In Figure 2.2, 1.2 is an ancestor of 1.2.2.1 because 1.2 is a prefix of 1.2.2.1. 1.2.2 is the parent of 1.2.2.1 because 1.2.2 matches the *parent_label* of 1.2.2.1. 1.2.2.1 and 1.2.2.2 are siblings because they have the same *parent_label* and the same number of components. The LCA of 1.2.2.1 and 1.2.3 is 1.2. □

2.1.3 Prime labeling scheme

Prime labeling scheme[45] represents a unique approach encoding the tree structure of XML data.

In prime labeling scheme, each node is associated with a unique prime number (*self_label*). The label of a node is a number which is the product of its *self_label* and the label of its parent node (*parent_label*). Since all *self_label*s

are distinct prime numbers, the factorization of a label can be used to identify a unique path in an XML tree. Given two nodes n and m, n is an ancestor of m if and only if $label(m)$ mod $label(n)$=0. n is the parent of m if and only if $label(n) = label(m)/self_label(m)$. n and m if and only if $label(n)/self_label(n) = label(m)/self_label(m)$. The label of the LCA of n and m is greatest common devisor of $label(n)$ and $label(m)$.

Although AD, PC, Sibling and LCA can be encoded elegantly in this way, using prime numbers as labels does not provide information about document orders, which has to be encoded separately. We describe how Prime labeling scheme encodes document order in Section 2.2.1.

2.2 Order encoding and update processing

Compared to *unordered* relational data, a key difference we face when processing *ordered* XML data is how to encode the order information[44]. Important order information defined in XML documents include *document order* and *local order*.

Definition 2.1 (Document order). *Document order is the order in which the start tags of the element nodes are encountered when the document that contains them is parsed. Note that document order is equivalent to preorder defined on the element nodes if we think of XML documents as linearizations of tree structure.*

Local order is the document order among siblings which is trivially consistent with document order.

Given the one-to-one correspondence between labels and element nodes, we can derive document order from a set of labels if they and their associated element nodes have the same ordering. When XML documents are subject to updates, i.e. element nodes are be inserted or deleted at arbitrary positions in the documents,

labels have to be inserted or deleted accordingly while preserving the correct order information. This turns out to be a challenging problem especially if no existing labels should be modified. We further elaborate the problem by summarizing the orders used by different labeling schemes.

2.2.1 Range-based labeling schemes and natural order

Since document order is equivalent to preorder on the element nodes, pre/post labeling scheme naturally encodes document order by incorporating the preorder traversal ordinal numbers into the labels. Given two pre/post labels $A(pre_1, post_1, l_1)$ and $B(pre_2, post_2, l_2)$, A precedes B in document order if and only if $pre_1 < pre_2$. Similarly, the *start* values in containment labels are strictly increasing if they are ordered according to document order. Thus, document order can be derived from containment labels from their *start* values.

The ordering of pre/post and containment labels follows from the *natural order* ($<$) on integers, i.e. *pre* or *start*. As we know, insertion between two integers requires the use of some new integers which falls between them in natural order. This is not possible if the existing two integers are consecutive, in which case re-labeling is necessary. The re-labeling may have global effect, that is, the whole document has to be re-labeled in the worst case. Leaving gaps[35] in labels only delays re-labeling until some gap is filled. Quartering-Regions Scheme (QRS) [9] proposes to use floating point numbers instead of integer. This solution does not solve the problem completely because (a)In standard floating point format, the mantissa is represented by a fixed number of bits, implying that floating point numbers are of limited accuracy; (b)The mantissa can be consumed by as many as 2 bits per insertion, which can lead to overflow after 18 insertions and (c) Floating point numbers are inherently less efficient to process than integers.

Prime labeling scheme uses a list of SC (Simultaneous Congruence) values to derive the mapping from $self_label$s to document orders, which are basically ordered by natural order. Whenever a node is inserted or deleted, the global orders are reordered. As a result, on average half of the SC values have to be re-calculated based on Euler's quotient function, which has been shown to be very time consuming[34].

2.2.2 Prefix-based labeling schemes and lexicographical order

Document order can be derived from Dewey labels based on lexicographical order (denoted as $\prec_l$) which is defined as follows:

Definition 2.2 (Lexicographical order)**.** *Given two Dewey labels* $A : a_1.a_2 \dots a_m$ *and* $B : b_1.b_2 \dots b_n$*,* $A \prec_l B$ *if and only if one of the following two conditions holds:*

C1. $m < n$ *and* $a_1 = b_1, a_2 = b_2, \dots, a_m = b_m$*.*

C2. $\exists k \in [0, min(m, n)]$*, such that* $a_1 = b_1, a_2 = b_2, \dots, a_{k-1} = b_{k-1}$ *and* $a_k < b_k$*.*

Consider the Dewey labels of two consecutive sibling element nodes, they have the $parent_label$ and consecutive local orders. From C2 in lexicographical order, the comparison of two labels eventually lead to comparison of local orders in natural order if two labels have the same $parent_label$. As a result, re-labeling is unavoidable for insertion between two consecutive siblings, regardless of whether integer or floating point number is used. However, the scope of re-labeling for Dewey labeling scheme is restricted to the subtree in which the new element node is inserted. In this sense, lexicographical order already appears to be more robust than natural order against updates.

2.2.3 Transforming natural order to lexicographical order

After showing that natural order is rigid and inevitably leads to re-labeling, it becomes clear that a different order is necessary to solve the problem of updates.

Several encoding schemes[32–34] have been proposed to transform integers into bit sequences, which, if we see from the order perspective, is from natural order to lexicographical order.

CDBS encoding scheme[33] transforms integers into binary strings that end with 1, which is referred to as CDBS codes.

Definition 2.3 (Binary string). *Given a set of binary numbers $A = \{0, 1\}$ where each number is stored with 1 bits, a binary string is a sequence of elements in A.*

CDBS codes are ordered by lexicographical order and allow arbitrary insertions (details in Section 5.2). Binary strings can be physically encoded into two formats: (1) V-CDBS where a fixed length field is attached before every V-CDBS code and (2) F-CDBS where all CDBS codes are of the same length. In both cases, the representations allow limited length of CDBS codes to be encoded. Overflow problem can happen if insertions produce CDBS codes that are too long to be represented.

Variable Length Endless Insertable (VLEI) encoding scheme [31] also transforms integers to binary strings. However, unlike CDBS codes, VLEI codes are not restricted to binary strings that end with 1 and are ordered by a variation of lexicographical order, which we refer to as VLEI order (denoted as $\prec_l^*$).

Definition 2.4 (VLEI order). *Given two VLEI codes $A : a_1.a_2 \ldots a_m$ and $B : b_1.b_2 \ldots b_n$, $A \prec_{VLEI} B$ if and only if one of the following three conditions holds:*

C1. $m < n$, $a_1 = b_1, a_2 = b_2, \ldots, a_m = b_m$ *and* $b_{m+1} = 1$.

C2. $m > n$, $a_1 = b_1, a_2 = b_2, \ldots, a_n = b_n$ *and* $a_{n+1} = 0$.

C2. $\exists k \in [0, min(m,n)]$*, such that* $a_1 = b_1, a_2 = b_2, \ldots, a_{k-1} = b_{k-1}$ *and* $a_k < b_k$*.*

Based on the definition, we have $10 \prec_{VLEI} 1 \prec_{VLEI} 11$ and $100 \prec_{VLEI} 10 \prec_{VLEI} 101 \prec_{VLEI} 1 \prec_{VLEI} 110 \prec_{VLEI} 11 \prec_{VLEI} 111$.

VLEI codes have similar dynamic property of CDBS codes. Experimental results demonstrate that the application of VLEI codes has achieved reduction in update time with respect to the use of floating point numbers[9].

QED encoding scheme has been proposed to solve the overflow problem of CDBS.

Definition 2.5 (Quaternary string)**.** *Given a set of numbers* $A = \{1, 2, 3\}$ *where each number is stored with 2 bits, a quaternary string is a sequence of elements in* A*.*

Note that number 0 does not appear in quaternary string because it is used as the separator of the quaternary strings for physical encoding. A *QED code* is a quaternary string that ends with 2 or 3. As the following example illustrates, QED codes are robust enough to allow insertions without re-labeling.

Example 2.3: Let 22, 23 be two QED codes satisfying $22 \prec_l 23$, we can insert 222 which is another QED code between them and we have $22 \prec_l 222 \prec_l 23$. To continue to insert between 22 and 222, for example, we can use 2212, satisfying $22 \prec_l 2212 \prec_l 222$. □

We refer to CDBS, VLEI and QED as encoding schemes because they can be used to transform range-based and prefix-based labeling schemes into dynamic formats. The resulting labeling schemes can process updates without re-labeling. However, a common drawback of these labeling schemes is that the lengths of binary and quaternary strings increase linearly if the insertion is ordered.

We refer to CDBS-Containment, VLEI-Containment and QED-Containment

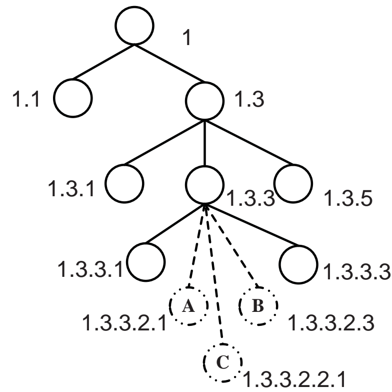


Figure 2.3: ORDPATH labeling scheme

labeling schemes as the applications of CDBS, VLEI and QED to containment labeling schemes. The resulting labeling schemes are ordered by lexicographical or VLEI order. Similarly, CDBS-Dewey, VLEI-Dewey and QED-Dewey labeling schemes are results of applying CDBS, VLEI and QED coding schemes to Dewey labeling schemes. The following section describes how they are ordered.

2.2.4 Transforming lexicographical order to generalized lexicographical order

We first introduce ORDPATH labeling scheme[38] which has been implemented in latest versions of Microsoft® SQL Server™.

Figure 2.3 shows an example of ORDPATH labeling scheme which resembles Dewey labeling scheme, except that only odd numbers are used in initial labeling. Although ORDPATH looks like Dewey labeling scheme, its processing is quite different, which is the result of the "careting in" technique used by ORDPATH labeling scheme to process insertions. We illustrate how the "careting in" technique works with the following example.

Example 2.4: In Figure 2.3, the dotted circles represent the inserted nodes which

are inserted in the alphabetical order of their associated letters. Node A is first inserted between two consecutive siblings with labels 1.3.3.1 and 1.3.3.3. We use 2 which is between 1 and 3 as the 'caret' and assign label 1.3.3.2.1 to node A which is the concatenation of the *parent_label*, the 'caret' and 1. Insertion of B can be treated like a rightmost insertion and its label is derived by increasing the last component of A by 2. Insertion of C is processed in a similar way as that of A. We attach another 'caret', 2, after 1.3.3, followed by an additional component, 1. □

Based on the 'careting in' technique, each level in an ORDPATH label is possibly represented by a variable number of even numbers followed by an odd number. This property complicates the processing of ORDPATH labels and therefore negatively affects the query performance. For example, computing the LCA of Dewey labels is equivalent to finding the longest common prefix of them. For ORDPATH labels, however, extra care has be to taken to make sure the LCA is a valid ORDPATH label. As an example, the longest common prefix of two ORDPATH labels 1.6.2.1 and 1.6.2.3.5 is 1.6.2 whereas their LCA should be 1. The complexity introduced by the 'careting in' technique *fundamentally* affects the query processing with ORDPATH labels even if no update actually takes place.

CDBS-Dewey, VLEI-Dewey, QED-Dewey and ORDPATH labeling schemes are similarly ordered, which can be captured by the generalized lexicographical order defined as follows.

Generalized lexicographical order

We propose the notions of generalized Dewey label and generalized lexicographical order to characterize the labels of prefix-based labeling schemes and their orders. First we generalize the notion of Dewey label.

Definition 2.6 (Generalized Dewey label)**.** *A generalized Dewey label is a sequence*

of logical components separated by dots, which we denote as $[a_1].[a_2]\ldots[a_m]$*. Here* $[a_i]$ *encloses a logical component which may consist of more than one component. The content of each component can be an integer, a string, a sequence of integers, etc. Nevertheless, the components should be encoded in such a way that allows them to be separable from each other.*

For example, QED-Dewey labels fit into the definition of generalized Dewey label as we can regard a QED code as a logical component and a sequence of QED codes are separated by delimiter 0. CDBS-Dewey and VLEI-Dewey labels are sequences of binary strings. In ORDPATH labeling scheme, a label can be thought of as a generalized Dewey label where each logical component is a variable of even numbers followed by an odd number. The components are separable from each other because the odd number marks the end of a component.

Generalized Dewey labels are compared based on generalized lexicographical order.

Definition 2.7 (Generalized lexicographical order)**.** *Given two generalized Dewey labels* $A : [a_1].[a_2]\ldots[a_m]$ *and* $B : [b_1].[b_2]\ldots[b_n]$*,* A *precedes* B *in generalized lexicographical order if and only if one of the two conditions holds:*

C1. $m < n$ *and* $a_1 \equiv b_1, a_2 \equiv b_2, \ldots, a_m \equiv b_m$*.*

C2. $\exists k \in [0, min(m, n)]$*, such that* $a_1 \equiv b_1, a_2 \equiv b_2, \ldots, a_{k-1} \equiv b_{k-1}$ *and* $a_k \prec b_k$*.*

$\equiv$ and $\prec$ denote generalized equivalence and generalized less than relation respectively. For generalized lexicographical order to correctly reflect document order, it has to be (a) *total* on the set of labels, i.e. any two generalized Dewey labels from the set of labels are comparable with respect to generalized lexicographical order and (b) *transitive* because document order itself is transitive.

Labeling scheme	Order	Component-wise equality	Component-wise order
Containment	natural	NA	NA
Pre/post	natural	NA	NA
QRS-Containment	natural	NA	NA
QRS-Pre/post	natural	NA	NA
Prime	natural	NA	NA
CDBS-Containment	lex	NA	NA
CDBS-Pre/post	lex	NA	NA
VLEI-Containment	VLEI	NA	NA
VLEI-Pre/post	VLEI	NA	NA
QED-Containment	lex	NA	NA
QED-Pre/post	lex	NA	NA
Dewey	lex	natural	natural
QRS-Dewey	lex	natural	natural
VLEI-Dewey	generalized lex	natural	VLEI
QED-Dewey	generalized lex	natural	lex
ORDPATH	generalized lex	natural	lex

Table 2.1: Summary of related work (lex is short for lexicographical)

2.3 Summary of chapter

In this chapter, we analyze the various labeling schemes proposed in the literature from an order-centric perspective. In Table 2.1, we summarize these labeling schemes and their orders of labels. Natural order-based labeling schemes are weak against updates and can easily lead to re-labeling. In contrast, dynamic labeling schemes are based on lexicographical order or VLEI order. In the following chapters, we propose our labeling schemes based on vector order which are fundamentally different from the existing solutions.

Chapter 3

Vector order and its applications

In this chapter, we introduce vector order which is the foundation of our labeling schemes. In addition, we present the application of vector order to both range-based and prefix-based labeling schemes.

3.1 Vector code ordering

Definition 3.1 (Vector code). *A vector code is an ordered pair of the form* (x, y) *with* $x > 0$*.*

A vector code (x, y) can be graphically interpreted as an arrow from the origin to the point (x, y) in a two dimensional plane. The arrow only falls into the first or the forth quadrant because we require $x > 0$. Three vector codes (2,3), (3,2) and (1,-2) are shown in Figure 3.1. We use the term *vector* to refer to the graphical representation of a vector code. Given the one-to-one correspondence between vector and vector codes, we will use the two terms interchangeably in the rest of the thesis.

Before formally defining vector order, we elaborate on the intuitive meaning behind it.

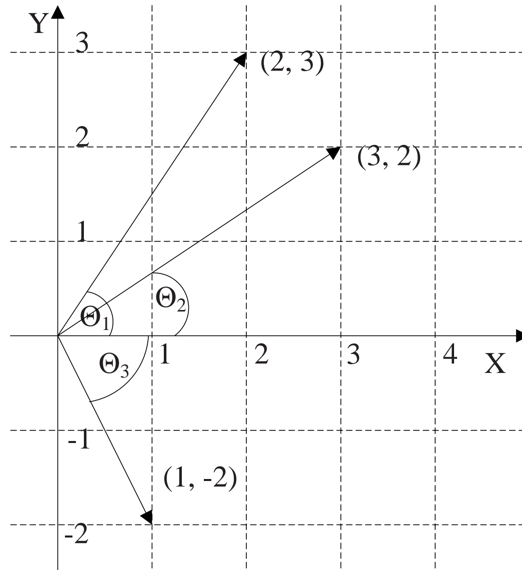


Figure 3.1: Graphical representation of vector codes

Intuitively, vector codes are ordered by $tan(\Theta)$ where Θ is the angle a vector makes with X axis. If we "rotate" a vector from the negative Y axis to the positive Y axis, Θ goes from $-90°$ (excluding $-90°$ itself) to $90°$ (excluding $90°$ itself) and $tan(\Theta)$ increases monotonously from $-\infty$ to ∞. In Figure 3.1, we have $tan(\Theta_3)$ $< tan(\Theta_2) < tan(\Theta_1)$ and the three vector codes are ordered accordingly. Note that the condition $x > 0$ restricts vector codes to be in the first and forth quadrant where vector order is a total order.

Given two vector codes $A : (x_1, y_1)$ and $B : (x_2, y_2)$, vector preorder is defined as:

Definition 3.2 (Vector preorder)**.** *A precedes B in vector preorder (denoted as* $A \preceq_v B$*) if and only if* $\frac{y_1}{x_1} \leq \frac{y_2}{x_2}$.

Vector equivalence is defined based on preorder.

Definition 3.3 (Vector equivalence)**.** *A is equivalent to B (denoted as* $A \equiv_v B$*) if and only if* $A \leq B$ *and* $B \leq A$*, or equivalently* $\frac{y_1}{x_1} = \frac{y_2}{x_2}$.

Equivalence relation is both symmetric and transitive.

Lemma 3.1 (Symmetry of vector equivalence). *If* $A \equiv_v B$, *then* $B \equiv_v A$.

Lemma 3.2 (Transitivity of vector equivalence). *Suppose* $A \equiv_v B$ *and* $B \equiv_v C$, *then* $A \equiv_v C$.

Graphically speaking, if two vector codes are equivalent, then they have the same direction. As the following lemma implies, equivalence relation can be reduced to natural equality if two vector codes have the same X component.

Lemma 3.3. *Suppose* $A \equiv_v B$ *and* $x_1 = x_2$, *then* $y_1 = y_2$.

We refer to this special form of vector equivalence as equality.

Definition 3.4 (Vector equality). *A is equal to B (denoted as A=B) if and only if* $x_1 = x_2$ *and* $y_1 = y_2$.

Given vector preorder and equivalence, vector order can be defined as follows:

Definition 3.5 (Vector order). $A \prec_v B$ *if and only if* $A \preceq_v B$ *and* $A \not\equiv_v B$ ($\not\equiv_v$ *is the negation of* $\equiv_v$), *equivalently,* $\frac{y_1}{x_1} < \frac{y_2}{x_2}$ *or* $y_1 \times x_2 < x_1 \times y_2$.

Two vector codes are comparable under vector order if and only if they are not equivalent to each other. We say a set of vector codes is *inequivalent* if it does not contain two vector codes that are equivalent to each other.

The following lemma addresses a special case where vector order can be reduced to natural less than relation.

Lemma 3.4. *Suppose* $A \prec_v B$ *and* $x_1 = x_2$, *then* $y_1 < y_2$.

Under the constraint that $x > 0$, this lemma follows immediately from Definition 3.5.

Same as equivalence relation, vector order is transitive.

Lemma 3.5 (Transitivity of vector order)**.** *If $A \prec_v B$ and $B \prec_v C$, then $A \prec_v C$.*

The following lemma establishes the connection between vector equivalence and vector order.

Lemma 3.6. *If $A \equiv_v B$ and $B \prec_v C$, then $A \prec_v C$; If $A \prec_v B$ and $B \equiv_v C$, then $A \prec_v C$.*

3.2 Vector code functions

We start by introducing two primitive functions to determine a new vector code that precedes or follows a given vector code $A : (x, y)$ in vector order.

- $BEF(A)$ return (x,y-1).

  //returns a vector code before A

- $AFT(B)$ return (x,y+1).

  //returns a vector code after A

It is readily verifiable from Lemma 3.4 that $BEF(A) \prec_v A \prec_v AFT(A)$.

To determine a new vector code that falls between two given vector codes in vector order, we introduce the following addition function.

Definition 3.6 (Vector code addition)**.** *Addition of two vector codes $A : (x_1, y_1)$ and $B : (x_2, y_2)$ is defined as:*

$$A + B = (x_1 + x_2, y_1 + y_2) \tag{3.1}$$

Multiplication function computes a vector code that is equivalent to the given vector code.

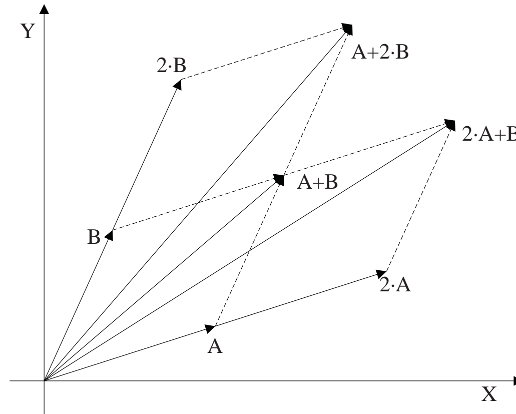


Figure 3.2: Vector code addition and multiplication

Definition 3.7 (Vector code and scalar multiplication). *Multiplication of an integer r and a vector code $A : (x, y)$ is defined as:*

$$r \cdot A = (r \times x, r \times y) \tag{3.2}$$

Addition and multiplication of vector codes are illustrated in Figure 3.2. Intuitively, a vector code and its multiples are equivalent to each other and can be represented as vectors of the same direction. That is, they make the same angle with X axis and are equivalent with respect to vector order. Given two vector codes that are not equivalent, e.g. A and B, the addition of them should produce a vector code that falls between them in vector order. Because the angle that the resulting vector makes with the X axis is between those that A and B make. We formalize our observations with the following results.

Let $A : (x_1, y_1)$ and $B : (x_2, y_2)$ be two vector codes,

Lemma 3.7. *Suppose $A \preceq_v B$, then $A \preceq_v (A + B) \preceq_v B$.*

Proof. From $A \preceq_v B$, we have $y_1 \times x_2 \leq y_2 \times x_1$. Therefore, $y_1 \times (x_1 + x_2) = y_1 \times x_1$ $+ y_1 \times x_2 \leq y_1 \times x_1 + y_2 \times x_1 = x_1 \times (y_1 + y_2)$. It follows that $A \preceq_v (A + B)$. Proof of the other half the lemma is similar, so we ignore it here. $\square$

Theorem 3.1. *Suppose $A \equiv_v B$, then $A \equiv_v (A+B) \equiv_v B$.*

Proof. $A \equiv_v B$ implies both $A \preceq_v B$ and $B \preceq_v A$. It then follows from Lemma 3.7 that $A \preceq_v (A+B) \preceq_v B$ and $B \preceq_v (A+B) \preceq_v A$. Thus, $A \equiv_v (A+B) \equiv_v B$. □

Theorem 3.2. *Suppose $A \prec_v B$, then $A \prec_v (A+B) \prec_v B$.*

Proof. It follows from $A \prec_v B$ that $A \preceq_v B$ and $A \not\equiv_v B$. Therefore, $A \preceq_v (A+B) \preceq_v B$ (1). Assume $A \equiv_v (A+B)$, we have $y_1 \times (x_1 + x_2) = x_1 \times (y_1 + y_2)$ which can be simplified as $y_1 \times x_2 = x_1 \times y_2$, and thus, $A \equiv_v B$. It is a contradiction to our assumption, therefore $A \not\equiv_v (A+B)$ (2). In the same way, we can prove $(A+B) \not\equiv_v B$ (3). Combining (1), (2) and (3), the theorem follows. □

The following corollary generalizes Theorem 3.2.

Corollary 3.1. *Given two vector codes A and B such that $A \prec_v B$, it follows that $A \prec_v \ldots \prec_v (3 \cdot A + B) \prec_v (2 \cdot A + B) \prec_v (A+B) \prec_v (A + 2 \cdot B) \prec_v (A + 3 \cdot B) \prec_v \ldots \prec_v B$.*

From Theorem 3.2, we can always find a vector code that falls between two vector codes in vector order. Together with transitivity of vector order in Lemma 3.5, the ordering among the set of vector codes after the insertion remains consistent. Given this result, we are ready to present how vector order can be applied to range-based and prefix-labeling schemes.

3.3 Applications of vector order

Both structural and order information of range-based labeling schemes depend on how the ranges in the labels are ordered. As a result, transforming the ranges into vector codes in an *order-preserving* manner can provide the flexibility to process insertions without re-labeling, while preserving all the useful information.

Order-preserving transformation		
Integer	Vector Code	
	Linear	Recursive
0		(1,0)
1	(1,1)	(5,1)
2	(1,2)	(4,1)
3	(1,3)	(3,1)
4	(1,4)	(5,2)
5	(1,5)	(2,1)
6	(1,6)	(7,5)
7	(1,7)	(5,3)
8	(1,8)	(3,2)
9	(1,9)	(4,3)
10	(1,10)	(1,1)
11	(1,11)	(4,5)
12	(1,12)	(3,4)
13	(1,13)	(2,3)
14	(1,14)	(3,5)
15	(1,15)	(1,2)
16	(1,16)	(2,5)
17	(1,17)	(1,3)
18	(1,18)	(1,4)
19		(0,1)

Table 3.1: Linear and recursive transformation for the range [1,18]

3.3.1 Order-preserving transformation

The ranges in a set of containment labels come from a sequence of integers from 1 to $2n$ for an XML tree with n elements. For pre/post labeling scheme, there are two identical sequences of integers from 1 to n. Let Z denote the set of integers and V denote the set of vector codes, a transformation $f : Z \rightarrow V$ is order-preserving if the following condition holds: $f(i) \prec f(j)$ if and only if $i < j$ for $i, j \in Z$. In this thesis, we introduce two transformations to vector codes that are order-preserving.

Recursive transformation

Column 3 of Table 3.1 shows the result of recursive transformation for integers from 1 to 18. The transformation proceeds in the following steps:

Step 1. Extend the range by adding a 0 before 1 and a 19 after 18. Assign (1,0) to 0 and (0,1) to 19 (Note that we manipulate (0,1) in the same way as a vector although it is not. (1,0) and (0,1) are used as auxiliary codes and will be discarded after the transformation).

Step 2. Calculate the middle position of the range (0,19): round(0+(19-0)/2)=10. Assign the sum of (1,0) and (0,1) to the middle position.

Step 3. Use the middle position to partition (0,19) into two sub-ranges (0,10) and (10, 19), repeat step 2 for each of the sub-ranges.

The process continues until all positions are assigned vector codes.

The reason for recursive transformation being order-preserving follows from Theorem 3.2. We can think of the transformation process as recursively inserting between existing vector codes.

Linear transformation

Linear transformation $f : Z \rightarrow V$ is defined as follows,

$$f(i) = (1, i) \qquad for \;\; i \in Z \tag{3.3}$$

An example of linear transformation for a sequence from 1 to 18 is shown in the second column of Table 3.1. It is an order-preserving transformation because, given any $i, j \in Z$ such that $i < j$, it follows from Lemma 3.4 that $(1, i) \prec_v (1, j)$. We use linear transformation to illustrate the application of vector order in this thesis.

While recursive transformation tries to minimize the maximum number used, it has to transform the whole range at a time and thus not as efficient as linear transformation to apply. Moreover, since all the vector codes transformed from linear transformation are of the form (1, i), we can compress them by assuming that a vector code with a single component has 1 as its X component. Based on our experimental results, linear transformation gives smaller label size when compressed ORDPATH format is used. Therefore we will apply linear transformation in the labeling schemes we introduce in the thesis. Our discussion also applies to recursive transformation.

3.3.2 V-Containment labeling scheme

We apply linear transformation to containment labels and refer to the resulting labels as V-Containment labels. We have described in Theorem 3.2 how to insert a new vector code between two consecutive ones in vector order. Insertion processing with V-Containment labeling schemes is slightly different, as two vector codes have to be inserted at one time which form the range of the new element node. We introduce the concept of *granularity sum* to guide such insertions.

Definition 3.8 (Granularity sum)**.** *The granularity sum of a vector code* $A:(x,y)$ *(denoted by* $GS(A)$*) is defined as* $x+y$*.*

We use granularity sum as an estimate of the size of vector codes. When inserting between two vector codes, we try to make use of the vector code of smaller granularity sum more, so that the resulting labels have a smaller overall size. The details are presented in Algorithm 1 whose correctness follows from Theorem 3.2. Note that the granularity sum we defined here is for illustration purpose only. In practice, more sophisticated measurement of the size can be used according to the physical encoding format.

Algorithm 1: InsertTwoVectorCodes(A, B)

Data: A and B which are two vector codes satisfying $A \prec_v B$
Result: C and D such that $A \prec_v C \prec_v D \prec_v B$

```
1 if GS(A) > GS(B) then
2     return (A+B) and (A+2·B);
3 else
4     return (2·A+B) and (A+B);
5 end
```

To study how insertion of a new node A can be processed with V-Containment labeling scheme, it is sufficient to consider the following three principles: (a) The range of A should be inside the range of A's parent; (b) The *start* of A should be be less than the *end* of its closest preceding sibling (if it exists) and (c) The *end* of A should be be less than the *start* of its closest following sibling (if it exists). Based on containment property, (a) obviously holds. If (b) or (c) is violated, it means there is some range that A and its sibling(s) have in common. If some new node is inserted as a descendant of A or one of A's siblings and assigned a range that is inside the common range, it would be a violation of the tree structure. Moreover, (b) and (c) guarantee that A has the correct document order. In all cases, the *level* of A equals to the level of A's parent plus 1.

Example 3.1: In Figure 3.3, the solid circles represent the elements nodes that are initially in the XML tree. Their labels are transformed from containment labeling scheme through linear transformation. Consider inserting element node A before the first child of the root. The *start* and *end* of A should fall between the *start* of A's parent and *start* of A's following sibling, that is, (1,1) and (1,2). Since $GS(1,1) = 2 < 3 = GS(1,2)$, it follows from Algorithm 1 the *start* and *end* of A should be $(3,4)$ $(= (2 \times 1 + 1, 2 \times 1 + 2))$ and $(2,3)$ $(= (1 + 1, 1 + 2))$. B is inserted after the last child of a node, its *start* and *end* should be bounded by the *end* of its preceding sibling and the *end* of its parent: (1,14) and (1,15). Applying Algorithm

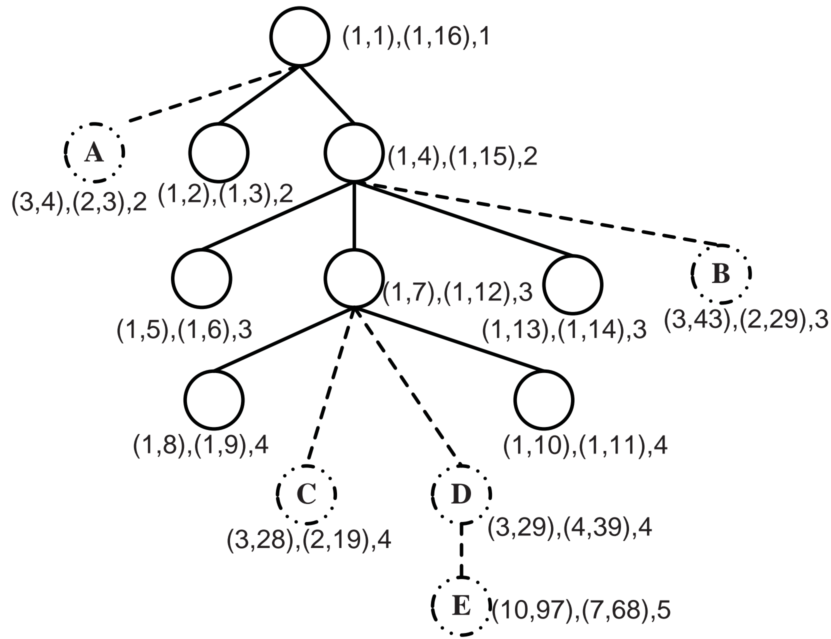


Figure 3.3: Process Updates with V-Containment labeling scheme

1, the *start* and *end* of B should be $(3, 43)$ $(= (2 \times 1 + 1, 2 \times 14 + 15))$ and $(2, 29)$ $(= (1 + 1, 14 + 15))$. C is inserted between two consecutive element nodes. Its *start* and *end* should be between the *end* of its preceding sibling and the *start* of its following siblings. From Algorithm 1, the *start* and *end* of C should be $(3, 28)$ $(= (2 \times 1 + 1, 2 \times 9 + 10))$ and $(2, 19)$ $(= (1 + 1, 9 + 10))$. Similarly, the *start* and *end* of C should be $(3, 29)$ $(= (2 \times 1 + 1, 2 \times 9 + 10))$ and $(4, 39)$ $(= (1 + 1, 9 + 10))$. The range of D is confined by its parent's range. The *start* and *end* of C should be $(10, 97)$ $(= (2 \times 3 + 4, 2 \times 29 + 39))$ and $(7, 68)$ $(= (3 + 4, 29 + 39))$. □

3.3.3 V-Pre/post labeling scheme

In V-Pre/post labeling scheme, insertion of a new node A can be processed based on two principles: (a)the *pre* of A should be between the *pre*s of two nodes that immediately precede and follow A during preorder traversal (if they exist) and (b)the *post* of A should be between the *post*s of two nodes that immediately precede

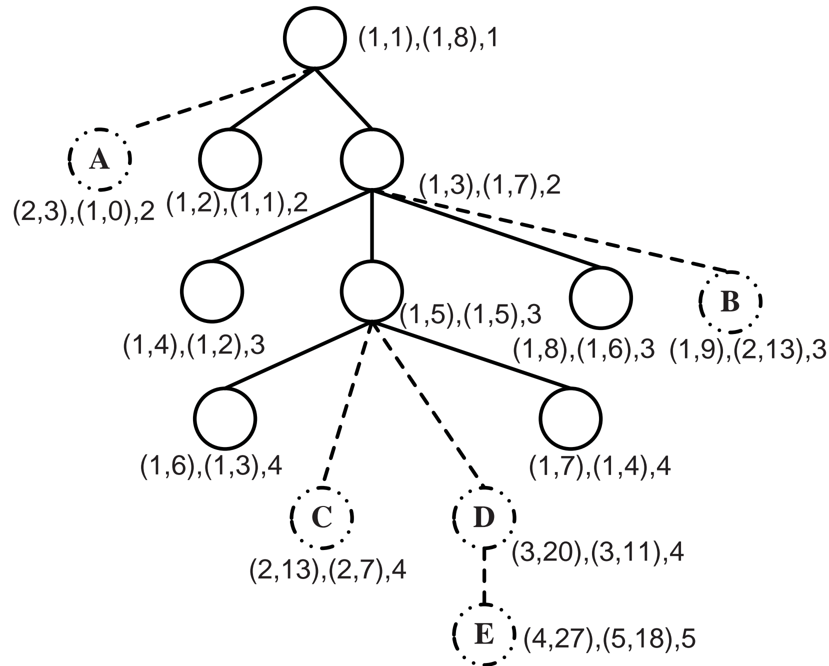


Figure 3.4: Process Updates with V-Pre/post labeling scheme

and follow A during postorder traversal (if they exist).

Example 3.2: First we consider the insertion of A which is a leftmost insertion under the root. To maintain correct document order, the *pre* of A should be between the *pre* of A's parent and A's following sibling. That gives us (2,3) which is the sum of (1,1) and (1,2). In addition, to keep AD and PC relationships, the *post* of A should be less than the *post* of A's following sibling, that is (1,1). We therefore assign $BEF(1,1) = (1,0)$ to the *post* of A. Since there is no element nodes that follow B during preorder traversal, we assign $AFT(1,8) = (1,9)$ to the $B.pre$. In addition, $B.post = (2,13) = (1,6) + (1,7)$. Insertions between two consecutive siblings (C and D) are processed in a similar manner. Insertion of a leaf node is more complicated with V-Pre/post labeling scheme than with V-Containment labeling scheme. Recall that in V-Containment labeling scheme, the label of the parent alone is sufficient to determine the label of the new leaf node. However, if we consider the insertion of D in Figure 3.4, its *pre* should fall between the *pre* of

its parent and the *pre* of its parent's following sibling. In addition, the *post* of D is confined by the *post* of its parent and the *post* of its parent's preceding sibling. Thus, the label of D is determined by three labels. □

3.3.4 V-Prefix labeling scheme

We introduce V-Prefix labeling scheme which is the most straight forward application of vector order to Dewey labeling scheme. It is derived from Dewey labeling scheme by transforming every Dewey label into a sequence of vector codes through linear transformation.

The initial labeling of V-Prefix labeling scheme is shown in Figure 3.5, with solid circles representing the element nodes initially in the XML tree. All vector codes are enclosed by brackets for easy reference.

Given a V-Prefix label of the form $(x_1, y_1).(x_2, y_2) \ldots (x_m, y_m)$, we denote it as: $v_1.v_2 \ldots v_m$ where $v_1 = (x_1, y_1)$, $v_2 = (x_2, y_2) \ldots v_m = (x_m, y_m)$. Thus, V-Prefix label can be seen as a generalized Dewey label where every component is a vector code.

V-Prefix labels are ordered by V-Prefix order.

Definition 3.9 (V-Prefix order)**.** *Given two V-Prefix labels $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_n$, A precedes B in V-Prefix order (denoted as $A \prec_{vp} B$) if and only if one of the following two conditions holds:*

C1. $m < n$ *and* $v_1 = w_1$, $v_2 = w_2$, $\ldots$, $v_m = w_m$.

C2. $\exists k \in [0, min(m, n)]$, *such that* $v_1 = w_1$, $v_2 = w_2$, $\ldots$, $v_{k-1} = w_{k-1}$ *and* $v_k \prec_v w_k$.

V-Prefix order fits into the definition of generalized lexicographical order where

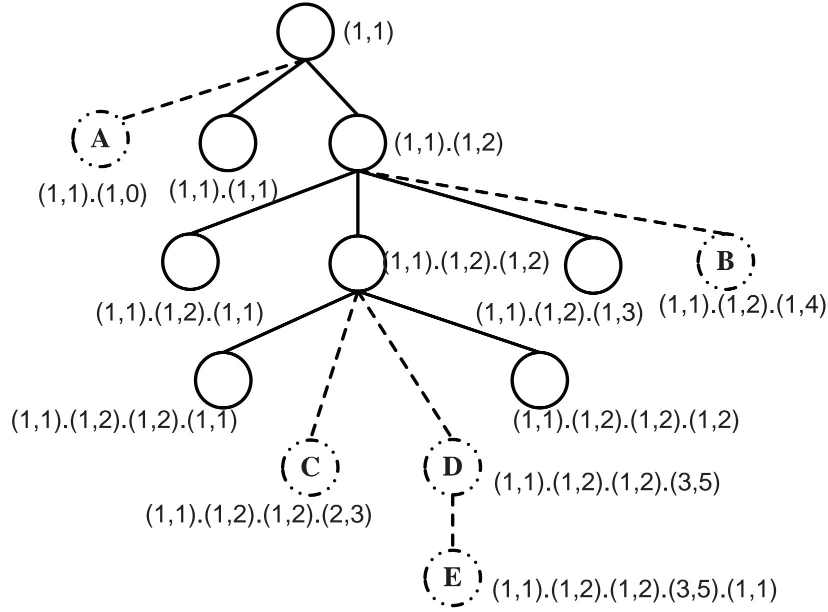


Figure 3.5: Process Updates with V-Prefix labeling scheme

equality and less than relations are those defined on vector codes (From Definition 3.4, two vector codes are equal if they have the same X and Y components).

Lemma 3.8 (Transitivity of V-Prefix order)**.** *Given three V-Prefix labels A, B and C such that* $A \prec_{vp} B$ *and* $B \prec_{vp} C$*, it follows that* $A \prec_{vp} C$*.*

This lemma can be proved based on the transitivity of vector order (Lemma 3.5).

Given two V-Prefix labels $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_n$, we summarize the properties of V-Prefix labels as follows:

P1 (AD Relationship). A is an ancestor of B if and only if $m < n$ and $v_1 = w_1$, $v_2 = w_2$, ..., $v_m = w_m$.

P2 (PC Relationship). A is the parent of B if and only if A is an ancestor of B and $m = n - 1$.

P3 (Document Order). A precedes B in document order if and only if $A \prec_{vp} B$.

P4 (Sibling Relationship). A is a sibling of B if and only if $m = n$ and $v_1 = w_1$, $v_2 = w_2$, ..., $v_{m-1} = w_{m-1}$.

These properties remain true after arbitrary insertions and deletions.

Correctness of initial labeling

First of all, structural information is kept correct after linear transformation because every V-Prefix label is still the concatenation of its *parent_label* and a vector code that represents its local order. To show the initial labeling is correct with respect to document order, it suffices to prove that the transformation from Dewey labels to V-Prefix codes is order-preserving.

Lemma 3.9. *Suppose we have two Dewey labels $A : a_1.a_2 \ldots a_m$ and $B : b_1.b_2 \ldots b_n$ such that $A \prec_l B$. Let $A' : v_1.v_2 \ldots v_m$ and $B' : w_1.w_2 \ldots w_n$ be the V-Prefix labels derived from A and B by applying linear transformation, it follows that $A' \prec_{vp} B'$.*

Proof. $A \prec_l B$ implies one of following two conditions:

C1. $m < n$ and $a_1 = b_1, a_2 = b_2, \ldots, a_m = b_m$. $a_1 = b_1$ implies $(1, a_1) = (1, b_1)$, or $v_1 = w_1$. In this way, we have $v_1 = w_1$, $v_2 = w_2$, ..., $v_m = w_m$ and therefore $A' \prec_{vp} B'$.

C2. $\exists k \in [0, min(m, n)]$, such that $a_1 = b_1, a_2 = b_2, \ldots, a_{k-1} = b_{k-1}$ and $a_k < b_k$. $a_k < b_k$ implies $(1, a_k) \prec_v (1, b_k)$. Again, we have $v_1 = w_1$, $v_2 = w_2$, ..., $v_{k-1} = w_{k-1}$, $a_m \prec_v b_m$ and therefore $A' \prec_{vp} B'$

□

V-Prefix label addition

The addition of V-Prefix labels is defined on two sibling V-Prefix labels.

Definition 3.10 (V-Prefix label addition). *Given two V-Prefix labels with sibling relationships* $A : v_1.v_2 \ldots v_{m-1}.v_m$ *and* $A' : v_1.v_2 \ldots v_{m-1}.v'_m$, $A + A'$ *is defined as:*

$$A + A' = v_1.v_2 \ldots v_{m-1}.(v_m + v'_m) \tag{3.4}$$

The following result directly follows from the property of vector code addition.

Lemma 3.10. *Let* A *and* A' *be two V-Prefix labels with sibling relationships, we have* $A \prec_{vp} (A + A') \prec_{vp} A'$.

How to process updates with V-Prefix labels are summarized as follows.

- **Leftmost insertion.** When a new node is inserted before node $v_1.v_2 \ldots v_{m-1}.v_m$ where A is the first child of a node, we assign label $v_1.v_2 \ldots v_{m-1}.BEF(v_m)$ to the new node.
- **Rightmost insertion.** When a new node is inserted before node $v_1.v_2 \ldots v_{m-1}.v_m$ where A is the last child of a node, we assign label $v_1.v_2 \ldots v_{m-1}.AFT(v_m)$ to the new node.
- **Insertion below a leaf node.** When a new node is inserted below a leaf node $v_1.v_2 \ldots v_{m-1}.v_m$, we assign label $v_1.v_2 \ldots v_{m-1}.v_m.(1, 1)$ to the new node.
- **Insertion between two consecutive siblings.** When a new node is inserted between two consecutive siblings with labels $v_1.v_2 \ldots v_{m-1}.v_m$ and $v_1.v_2 \ldots v_{m-1}.v'_m$, we assign label $v_1.v_2 \ldots v_{m-1}.(v_m + v'_m)$ to this node.

In all the cases, the *parent_label* of the new V-Prefix label remains the same as its parent's label. These algorithms are illustrated with the following example. We illustrate how to process updates with V-Prefix labels with the following example.

Example 3.3: First we consider the leftmost insertion of element node A in Figure 3.5. A should have the same *parent_label* as its parent's 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 $BEF(1,1) = (1,0)$. Since B is inserted at the rightmost position after (1.1).(1.2).(1.3), we derive its local order to be $AFT(1,3) = (1,4)$. 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, say, $(1,1)$. □

Correctness

We have pointed out that the *parent_label* of a new label is the same as its parent's label in all insertion cases. Thus, we consider it obvious that the structural information, including PC, AD and sibling, is correctly maintained. To see why document order is also kept correct, we consider insertion between two consecutive siblings. The correctness of the rest of the cases are easy to see. From Lemma 3.10, the new label falls between its preceding and following siblings in V-Prefix order. Taking transitivity of V-Prefix order (Lemma 3.8) into consideration, the new label follows its preceding sibling and all element nodes that precede its preceding sibling. Similarly, it precedes its following siblings and all element nodes that follow its following sibling. What remains to be show is that the new label follows all the descendants of its preceding sibling in V-Prefix order.

Lemma 3.11. *Given three V-Prefix labels A and B and $C = A + B$, such that A is a sibling of B and $A \prec_{vp} B$, if A' is a descendant of A, then $A' \prec_{vp} C$.*

Proof. Since A and B are siblings, we denote them as $A : v_1.v_2 \ldots v_{m-1}.v_m$ and

$B : v_1.v_2 \ldots v_{m-1}.v'_m$. Thus, C is $v_1.v_2 \ldots v_{m-1}.(v_m + v'_m)$. $A \prec_{vp} B$ implies $v_m \prec_v v'_m$. From Theorem 3.2, $v_m \prec_v (v_m + v'_m)$. Since A' is a descendant of A, A is a prefix of A'. Therefore, $A' \prec_{vp} C$. □

3.4 Summary of chapter

Labeling dynamic XML documents is a challenging problem that has been extensively studied over the years. In this chapter, we propose a novel order concept, vector order which can be widely applied to different labeling schemes to process updates without re-labeling. From the order perspective, our approach is fundamentally different from the previous approaches. The correctness of our approach follows from the properties of vector order that it is both total and transitive on vector codes. Moreover, vector order can be easily applied to containment, pre/post and Dewey labeling schemes to process updates without re-labeling. Lastly, we prove the correctness of our algorithms. In Chapter 4, we extend the concept of vector order to improve V-Prefix labeling scheme, followed by the experimental evaluations of all the labeling schemes we proposed at the end of the chapter.

Chapter 4

Extension of vector order and its applications

In this chapter, we present two prefix-based dynamic labeling schemes that are based on the extension of vector order and vector equivalence.

4.1 DDE labeling scheme

First we introduce Dynamic DEwey (DDE) labeling scheme.

4.1.1 Motivation

While V-Prefix labeling appears to be the straight forward application of vector order to Dewey labeling scheme, its drawbacks are also obvious. Transforming from integer to vector codes doubles the number of components of the labels and increases the overall label size. The added cost may be justified if (a) The users are interested in querying changes, so a persistent labeling scheme is needed and (b) The documents will be extensively updated and the update performance is

crucial. However, it is undesirable and wasteful to introduce these costs if the documents will remain static or get seldom updated. We have emphasized that a single labeling scheme should be designed to fit both static and dynamic XML documents due to the unpredictability of updates. Although V-Prefix labeling scheme can allow arbitrary insertions in the dynamic setting, its label size is far from optimal for static XML documents.

In this chapter, we improve V-Prefix to DDE labeling scheme which is dynamic enough to completely avoid re-labeling, while introducing minimum additional complexity to static documents.

4.1.2 Initial Labeling

Every DDE label is a sequence of integers separated by dots. The initial labeling of DDE labeling scheme is the same as Dewey (Figure 2.2). However, the semantic meanings of DDE and Dewey are very different. A Dewey label can be seen as a concatenation of local orders from the root to an element node whereas we interpret a DDE label as a sequence of vector codes that share a common X component. We will show that the directions of these vectors together with the ordering of them can uniquely determine a path from the root to an element node.

A more intuitive representation of DDE labels is defined in terms of vector codes.

Definition 4.1 (Vector representation of DDE label)**.** *Given a DDE label of the form* $x.y_1.y_2 \ldots y_m$*, its vector representation is* $v_1.v_2 \ldots v_m$ *where* $v_1 = (x, y_1)$*,* $v_2 = (x, y_2) \ldots v_m = (x, y_m)$*.*

We can see that the first component of a DDE label is shared by the sequence of vector codes as the X component. Note that the root element 1 does not fit into this interpretation and has to be specially dealt with. We consider a DDE label

as a generalized Dewey label where each component is a vector code (all of which share a common X component).

4.1.3 DDE label ordering

DDE labels are ordered by DDE order which can be defined as follows:

Definition 4.2 (DDE order)**.** *Given two DDE labels $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_n$, A precedes B in DDE order (denoted as $A \prec_{dde} B$) if and only if one of the following two conditions holds:*

C1. $m < n$ *and* $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, $\ldots$, $v_m \equiv_v w_m$.

C2. $\exists k \leq min(m, n)$, *such that* $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, $\ldots$, $v_{k-1} \equiv_v w_{k-1}$ *and* $v_k \prec_v w_k$.

The label of the root is minimum with respect to DDE order, i.e. it precedes all other labels. DDE order can be seen as generalized lexicographical order where component wise comparison is based on vector equivalence and vector order.

DDE order is transitive.

Lemma 4.1 (Transitivity of DDE order)**.** *Given three DDE labels A, B and C such that $A \prec_{dde} B$ and $B \prec_{dde} C$, it follows that $A \prec_{dde} C$.*

Proof. From Definition 4.2, $\prec_{dde}$ can imply one of two conditions. Therefore there are four cases to consider, which can be proved based on Lemma 3.5, Lemma 3.2 and Lemma 3.6. □

The equivalence relation on DDE labels can be defined as:

Definition 4.3 (DDE equivalence)**.** *Two DDE labels $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_n$ are equivalent (denoted as $A \equiv_{dde} B$) if and only if $m = n$ and $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, $\ldots$, $v_m \equiv_v w_m$.*

Two DDE labels are comparable with respect to DDE order if and only if they are not equivalent. We say that a set of DDE labels is *inequivalent* if there does not exist two DDE labels in the set with equivalence relation. Let A and B be two distinct DDE labels from an inequivalent set of DDE labels, we have either $A \prec_{dde} B$ or $B \prec_{dde} A$ (not both).

Lemma 4.2 (Transitivity of DDE equivalence)**.** *Given three DDE labels A, B and C, if $A \equiv_{dde} B$ and $B \equiv_{dde} C$, then $A \equiv_{dde} C$.*

This lemma easily follows from the transitivity of vector equivalence.

4.1.4 DDE label properties

A DDE label implicitly stores the level information as the number of components in that label. This property will remain true after arbitrary insertions and deletions.

Given two DDE labels $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_n$, we summarize the properties of DDE labels as follows:

- P1 (AD Relationship). A is an ancestor of B if and only if $m < n$ and $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, ..., $v_m \equiv_v w_m$. (The case where A is the root always returns true.)
- P2 (PC Relationship). A is the parent of B if and only if A is an ancestor of B and $m = n - 1$.
- P3 (Document Order). A precedes B in document order if and only if $A \prec_{dde} B$.
- P4 (Sibling Relationship). A is a sibling of B if and only if $m = n$ and $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, ..., $v_{m-1} \equiv_v w_{m-1}$.
- P5 (LCA). The LCA of A and B is C, such that C is an ancestor of both A and B, and either (1) $|C| = min(m, n)$, or (2) $v_{|C|+1} \not\equiv_v w_{|C|+1}$.

From Lemma 3.3 and Lemma 3.4, $\equiv_v$ and $\prec_{dde}$ can be reduced to $=$ and $<$ respectively if two vector codes have the same X component. Such reductions can be applied to all the initial DDE labels because they all have 1 as their first component and, as we know, the first component serves as the X component for the sequence of vector codes in every DDE label. For example, AD relationship can be simplified for initial DDE labels as follows.

P1 (AD Relationship (for initial DDE labels)). A is an ancestor of B if and only if $m < n$ and $v_1 = w_1$, $v_2 = w_2$, ..., $v_m = w_m$.

It follows from the reduction that the initial DDE labels can be treated as Dewey labels which we consider to be tailored for static XML documents.

4.1.5 Correctness of initial labeling

Lemma 4.3. *Based on DDE labeling scheme, the set of initial DDE labels is inequivalent.*

Proof. We establish the proof by contradiction. Suppose the set of initial DDE labels is not inequivalent, there exist two DDE labels $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_m$, such that $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, ..., $v_m \equiv_v w_m$. However, since all the initial DDE labels start with 1, it follows that $v_1 = w_1$, $v_2 = w_2$, ..., $v_m = w_m$, which means A and B are the same. We have a contradiction here because all DDE labels are different in initial labeling. □

Since the set of initial DDE labels is inequivalent, it follows that any two of them are comparable with respect to DDE order. In addition, DDE order can be reduced to Dewey order for the initial DDE labels because all of them start with 1. The fact that our initial label assignment is the same as Dewey implies that

document order is correct with respect to Dewey order and therefore DDE order. The same reasoning applies to all the other properties of DDE labels.

4.1.6 DDE label addition

To process dynamic insertions between DDE labels while preserving their relative order, we introduce addition operation on DDE labels. The addition operation is defined on DDE labels with the same number of components.

Definition 4.4 (DDE label addition)**.** *Given two DDE labels with the same number of components $A : v_1.v_2\ \ldots\ v_m$ and $B : w_1.w_2\ \ldots\ w_n$, $A + B$ is defined as:*

$$A + B = (v_1 + w_1).(v_2 + w_2)\ldots(v_m + w_m) \tag{4.1}$$

Note that the first integer in a DDE label, which is the common X component, only needs to be added once.

The following theorem formalizes important properties of the addition operation.

Theorem 4.1. *Given two DDE labels $A : v_1.v_2\ \ldots\ v_m$ and $B : w_1.w_2\ \ldots\ w_n$ such that A is a sibling of B and $A \prec_{dde} B$, then $A \prec_{dde} (A + B) \prec_{dde} B$.*

Proof. Since A and B are siblings, we have $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, $\ldots$, $v_{m-1} \equiv_v w_{m-1}$. From Theorem 3.1, $v_1 \equiv_v (v_1 + w_1) \equiv_v w_1$, $v_2 \equiv_v (v_2 + w_2) \equiv_v w_2$, $v_{m-1} \equiv_v (v_{m-1} + w_{m-1}) \equiv_v w_{m-1}$. In addition, $A \prec_{dde} B$ implies that $v_m \prec_v w_m$. It then follows from Theorem 3.2 that $v_m \prec_v (v_m + w_m) \prec_v w_m$. As a result, $A \prec_{dde} (A + B) \prec_{dde} B$. □

Theorem 4.2. *Given two DDE labels $A : v_1.v_2\ \ldots\ v_m$ and $B : w_1.w_2\ \ldots\ w_m$ such that $A \equiv_{dde} B$, then $A \equiv_{dde} (A + B) \equiv_{dde} B$.*

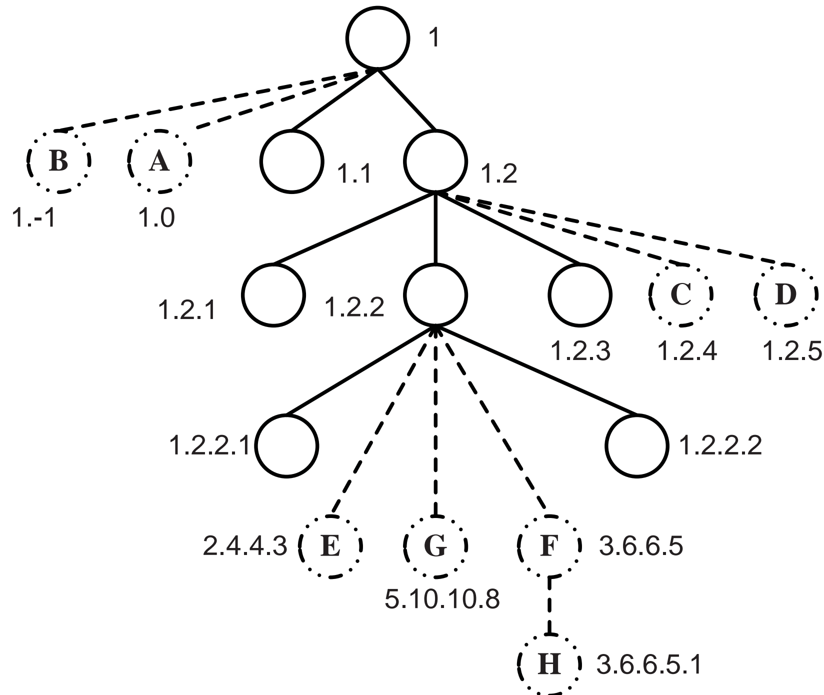


Figure 4.1: Processing insertions with DDE labels

Proof. From $A \equiv_{dde} B$, we have $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, ..., $v_m \equiv_v w_m$. Applying Lemma 3.2, we have $v_1 \equiv_v (v_1 + w_1) \equiv_v w_1$, $v_2 \equiv_v (v_2 + w_2) \equiv_v w_2$, ..., $v_m \equiv_v (v_m + w_m) \equiv_v w_m$, and therefore $A \equiv_{dde} (A + B) \equiv_{dde} B$. □

We use the following example to illustrate the properties of DDE labels that have been introduced so far.

Example 4.1: Consider the XML tree in Figure 4.1, the dotted circles represent the new nodes inserted into the XML tree. We ignore for now how their labels are generated. Node 1.2 is an ancestor of node E as $(1, 2) \equiv_v (2, 4)$ and $|1.2| < |E|$ ($|E|$ denotes the number of components in E). Node 1.2.2 is the parent of G as $(1, 2) \equiv_v (5, 10)$ and $|1.2.2| = |G| - 1$. $A \prec_{dde} E$ as $(1, 0) \prec_v (2, 4)$, so H precedes E in document order. E is a sibling of F because $|E| = |F|$ and $(2, 4) \equiv_v (3, 6)$. In addition, $E \prec_{dde} F$ as $(2, 4) \equiv_v (3, 6)$ and $(2, 3) \prec_v (3, 5)$. Note that G=E+F as $5.10.8 = 2.4.3 + 3.6.5$, since E is a sibling of F and E $\prec_{dde}$ F, we have $E \prec_{dde}$

$G \prec_{dde} F$ based on Theorem 4.1. To verify, $E \prec_{dde} G$ as $(2,4) \equiv_v (5,10)$ and $(2,3) \prec_v (5,8)$, $G \prec_{dde} F$ as $(5,10) \equiv_v (3,6)$ and $(5,8) \prec_v (3,5)$. □

4.1.7 Processing updates

Similar to that of Dewey labels, it is clear that the deletion of DDE labels does not affect the order of the other labels. The challenging part is how to handle insertions without re-labeling. Note that, like ORDPATH, we extend the domain of component values of DDE labels to positive number, negative number and 0. However, since ORDPATH only uses odd numbers at initial labeling, its labels are not as compact as DDE and Dewey.

First we introduce how DDE labeling scheme processes insertions with an example.

Example 4.2: In Figure 4.1, 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 1.2.3, we get its label 1.2.4 by adding 1 to the local order of 1.2.3. Similarly, the label of node D is 1.2.5. Node E is inserted between two nodes with labels 1.2.2.1 and 1.2.2.2 and its label is 2.4.4.3 which equals to 1.2.2.1+1.2.2.2. Likewise, the labels of node F and G are 3.6.6.5 (2.4.4.3+1.2.2.2) and 5.10.10.8 (2.4.4.3+3.6.6.5) respectively. Node H is inserted as the child of leaf node 3.6.6.5, its label is the concatenation of its parent's label and 1. □

Among the insertions shown Figure 4.1, we consider the correctness of the following special cases obvious because the resulting labels are almost the same as the initial labeling, so proofs are ignored here.

- **Leftmost insertion.** When a new node is inserted before node $A : v_1.v_2 \ldots$

v_m where A is the first child of a node, we assign label $v_1.v_2 \ldots BEF(v_m)$ to this node.

- **Rightmost insertion.** When a new node is inserted after node $A : v_1.v_2 \ldots v_m$ where A is the last child of a node, we assign label $A : v_1.v_2 \ldots AFT(v_m)$ to this node.

- **Insertion below a leaf node.** When a new node is inserted below a leaf node $A : v_1.v_2 \ldots v_m$, we assign label $A : v_1.v_2 \ldots v_m.1$ to this node.

In general, insertions can be made between any two consecutive siblings.

- **Insertion between two consecutive siblings.** When a new node is inserted between two consecutive siblings with labels A and B, we assign label $A + B$ to this node.

We prove the correctness of this case in Section 4.1.8. In conclusion, DDE labeling scheme supports insertions at arbitrary positions in an XML tree.

4.1.8 Correctness

We show the correctness of our insertion algorithm in terms of structural information and document order.

By definition, two DDE labels A and B have sibling relationship if and only if their *parent_label*s are equivalent. Given Theorem 4.2, the *parent_label* of $A+B$ is equivalent to both A and B, which from transitivity of DDE equivalence (Lemma 4.2), implies that $A + B$ is a sibling of A, B and all siblings of A and B. Sibling relationship is therefore correctly maintained.

A DDE label C is an ancestor of another DDE label A if and only if C is equivalent to a proper prefix of A. We have shown that, the *parent_label* of $A+B$

is equivalent to the *parent_label*s of A and B if A and B are siblings. As a result, transitivity of DDE equivalence (Lemma 4.2) indicates that, any ancestor of A and B is equivalent to a proper prefix of $A+B$ and is therefore an ancestor of $A+B$. The insertion is also correct with respect to PC relationship because the number of components of a DDE label is kept the same as the *level* of the corresponding element node.

The correctness of DDE insertion with respect to document order follows from Theorem 4.1, Lemma 4.1 and the following lemma.

Lemma 4.4. *Given three DDE labels A and B and $C = A+B$, such that A is a sibling of B and $A \prec_{dde} B$, if A' is a descendant of A, then $A' \prec_{dde} C$.*

Proof. We denote A and B as $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_m$ respectively. From A is a sibling of B and $A \prec_{dde} B$, we have $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, $\ldots$, $v_{m-1} \equiv_v w_{m-1}$, $v_m \prec_v w_m$. It follows from Theorem 4.2 and Theorem 4.1 that $v_1 \equiv_v (v_1+w_1)$, $v_2 \equiv_v (v_2+w_2)$, $\ldots$, $v_{m-1} \equiv_v (v_{m-1}+w_{m-1})$, $v_m \prec_v (v_m+w_m)$. Since A' is a descendant of A, we can denote A' as $v'_1.v'_2 \ldots v'_m \ldots v'_{n-1}v'_n$ where $v'_1 \equiv_v v_1$, $v'_2 \equiv_v v_2$, $\ldots$, $v'_{m-1} \equiv_v v_{m-1}$, $v'_m \equiv_v v_m$. From Lemma 3.2 and Lemma 3.6, $v'_1 \equiv_v w_1$, $v'_2 \equiv_v w_2$, $\ldots$, $v'_{m-1} \equiv_v w_{m-1}$, $v'_m \prec_v w_m$. Thus, $A' \prec_{dde} C$. □

4.2 Compact DDE (CDDE)

In this section, we introduce a variant of DDE labeling scheme which we call Compact DDE (CDDE). CDDE is designed to enhance the performance of DDE for insertions.

4.2.1 Initial labeling

The label format of CDDE is the same as DDE which is a sequence of components separated by '.'. Moreover, the initial labeling of CDDE is the same as DDE, and is therefore the same as Dewey (Figure 2.2). Unlike DDE labels whose first components are restricted to be positive decimal numbers, the first component of a CDDE label can be either positive or negative. We refer to the CDDE labels with positive first components as *positive CDDE labels* and those with negative first components as *negative CDDE labels*.

Let $A : a_1.a_2 \ldots a_m$ be a positive CDDE label, we refer to a_1 as *multiplier*, $a_1.a_2 \ldots a_{m-1}$ as *parent label* and a_m as *local order*. If $A : a_1.a_2 \ldots a_m$ is a negative CDDE label, then its multiplier, parent label and local order are a_1, $a_2.a_2 \ldots a_{m-1}$ and a_m respectively.

4.2.2 CDDE label to DDE label mapping

The properties of CDDE label, which include how various relationships can be established, are different from those of DDE. To simplify discussion, we take a shortcut by defining a mapping from CDDE label to DDE label.

Given a CDDE label $A : a_1.a_2.a_3 \ldots a_{m-1}.a_m$, we define a mapping $f^{cd} : CDDE\ label \rightarrow DDE\ label$ as:

$$f^{cd}(A) = \begin{cases} a_1.(a_1 \times a_2).(a_1 \times a_3) \ldots (a_1 \times a_{m-1}).a_m \\ \qquad\qquad\qquad\qquad when\ a_1 > 0 \\ (|a_1| \times a_2).(|a_1| \times a_3) \ldots (|a_1| \times a_{m-1}).a_m \\ \qquad\qquad\qquad\qquad when\ a_1 < 0 \end{cases}$$

Intuitively, the mapping is to apply the 'multiplier' to the parent label of the CDDE label. The multiplier is part of the parent label for positive CDDE labels, but is

followed by the parent label for negative CDDE labels. For example, CDDE label 2.2.3 maps to DDE label 2.(2×2).3=2.4.3 whereas CDDE label -3.1.3.2.1 maps to DDE label $(3 \times 1).(3 \times 3).(3 \times 2).1$=3.9.6.1.

Based on f^{cd} mapping, we define preorder (denoted as $\preceq_{cdde}$) on CDDE labels as:

Definition 4.5 (Preorder)**.** *Given two CDDE labels A and B, $A \preceq_{cdde} B$ if and only if $f^{cd}(A) \preceq_{dde} f^{cd}(B)$.*

Definition 4.6 (Equivalence relation)**.** *Two CDDE labels A and B have equivalence relation if and only if $f^{cd}(A) =_e f^{cd}(B)$.*

Similarly, CDDE order (denoted as $\prec_{cdde}$) is defined as:

Definition 4.7 (CDDE order)**.** *Given two CDDE labels A and B, $A \prec_{cdde} B$ if and only if $f^{cd}(A) \prec_{dde} f^{cd}(B)$.*

We summarize the properties of CDDE labels as:

- A CDDE label A is the parent/ ancestor/ sibling of another CDDE label B if and only if $f^{cd}(A)$ is the parent/ ancestor/ sibling of $f^{cd}(B)$.
- A CDDE label A precedes another CDDE label B in document order if and only if $A \prec_{cdde} B$.

Correctness of initial labeling

Given any CDDE label $A : 1.a_2.a_3 \ldots a_{m-1}.a_m$ in the initial labeling, we have $f^{cd}(A) = f^{cd}(1.a_2.a_3 \ldots a_{m-1}.a_m) = 1.a_2.a_3 \ldots a_{m-1}.a_m = A$, implying that the initial CDDE labels simply map those initial DDE labels. Therefore the correctness of CDDE initial labeling follows directly from that of DDE initial labeling which we have proved in Section 4.1.5.

4.2.3 CDDE label addition

Similar to DDE label addition, CDDE label addition applies to two CDDE labels with sibling relationship.

Lemma 4.5. *Let* $A : a_1.a_2.a_3 \ldots a_{m-1}.a_m$ *and* $B : b_1.b_2.b_3 \ldots b_{n-1}.b_n$ *be two CDDE labels with sibling relationship, then a)* a_1 *and* b_1 *are both positive or both negative; b)* $m = n$*; and c)* $a_2 = b_2, a_3 = b_3 \ldots a_{m-1} = b_{m-1}$.

Lemma 4.5 obviously holds for the initial CDDE labels as they are all positive labels and among them, any two siblings have the same parent label. We will show that this lemma remains to be valid after updates in Section 4.2.4.

An important difference between CDDE and DDE is how insertions are handled. We define addition operation of CDDE labels as:

Definition 4.8 (CDDE label addition)**.** *Let* $A : a_1.a_2\ .a_3 \ldots a_{m-1}.a_m$ *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:*

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

Different from DDE label addition, CDDE label addition only adds up the multipliers and local orders of two CDDE labels. As a result, the label size of CDDE increases at a slower rate than DDE after additions. However, the addition operations of DDE and CDDE labels are actually equivalent, as the following lemma implies.

Lemma 4.6. *Given two CDDE labels* $A : a_1.a_2.a_3 \ldots a_{m-1}\ .a_m$ *and* $A' : a'_1.a_2.a_3 \ldots a_{m-1}.a'_m$.

$$f^{cd}(A +_c A') = f^{cd}(A) + f^{cd}(A')$$

Proof. We consider two cases:

Both A and A' are positive CDDE labels. $f^{cd}(A +_c A') = f^{cd}((a_1 + a'_1).a_2.a_3 \ldots (a_m + a'_m)) = (a_1 + a'_1).((a_1 + a'_1) \times a_2).((a_1 + a'_1) \times a_3) \ldots ((a_1 + a'_1) \times a_{m-1}).(a_m + a'_m) = a_1.(a_1 \times a_2).(a_1 \times a_3) \ldots (a_1 \times a_{m-1}).a_m + a'_1.(a'_1 \times a_2).(a'_1 \times a_3) \ldots (a'_1 \times a_{m-1}).a'_m = f^{cd}(A) + f^{cd}(A')$

Both A and A' are negative CDDE labels. $f^{cd}(A +_c A') = f^{cd}((a_1 + a'_1).a_2.a_3 \ldots (a_m + a'_m)) = (|(a_1 + a'_1)| \times a_2).(|(a_1 + a'_1)| \times a_3) \ldots (|(a_1 + a'_1)| \times a_{m-1}).(a_m + a'_m) = (|a_1| \times a_2).(|a_1| \times a_3) \ldots (|a_1| \times a_{m-1}).a_m + (|a'_1| \times a_2).(|a'_1| \times a_3) \ldots (|a'_1| \times a_{m-1}).a'_m = f^{cd}(A) + f^{cd}(A')$

In both cases, $f^{cd}(A +_c A') = f^{cd}(A) + f^{cd}(A')$. □

Lemma 4.7. *Suppose A and B are two CDDE labels such that $A \prec_{cdde} B$, then $A \prec_{cdde} (A +_c B) \prec_{cdde} B$.*

Proof. Based on Definition 4.7, $A \prec_{cdde} B$ is equivalent to $f^{cd}(A) \prec_{dde} f^{cd}(B)$, which in turn implies that $f^{cd}(A) \prec_{dde} f^{cd}(A) + f^{cd}(B) \prec_{dde} f^{cd}(B)$ (Theorem 4.1). By Lemma 4.6, we can replace $f^{cd}(A) + f^{cd}(B)$ with $f^{cd}(A +_c B)$, which gives $f^{cd}(A) \prec_{dde} f^{cd}(A +_c B) \prec_{dde} f^{cd}(B)$. Thus, $A \prec_{cdde} (A +_c B) \prec_{cdde} B$. □

4.2.4 Processing updates

We illustrate how CDDE handles updates with an example.

Example 4.3: As illustrated in Figure 4.2, leftmost insertions (node A and B) and rightmost insertions (node C and D) are processed in the same way as DDE labels. However, 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). □

Leftmost and rightmost insertions obviously do not violate the properties of sibling relationship stated in Lemma 4.5 as only local orders are changed. Moreover,

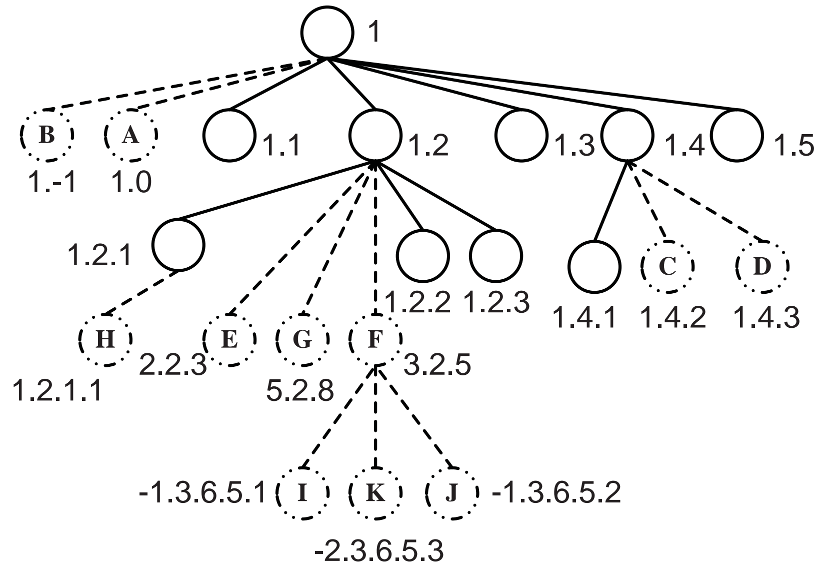


Figure 4.2: Processing insertions with CDDE labels

we can see that Lemma 4.5 still holds after insertion between two positive CDDE labels (e.g. node E, F and G) because addition of CDDE labels only adds up the multipliers and local orders while their parent labels remain to be the same.

Processing insertion below a leaf node

We have shown how to process insertion below a leaf node with DDE label. The new label can be generated by concatenating the parent's label with 1. However, this method does not work for CDDE labels because it will produce new labels with incorrect parent labels. To accommodate such insertions, we introduce another operation which is used to get the label for the new node:

Definition 4.9 (CDDE label extension)**.** *The extension operation of a CDDE label*

$A : a_1.a_2.a_3 \ldots\ a_{m-1}.a_m$ *is defined as:*

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

The next lemma shows the equivalence of DDE and CDDE label extension.

Lemma 4.8. *Given a CDDE label* $A : a_1.a_2.a_3 \ldots a_{m-1}.a_m$, $f^{cd}(EXT(A)) = f^{cd}(A).1$.

Proof. There are three cases to be considered:

$\mathbf{a}_1 > \mathbf{1}$. $f^{cd}(EXT(A)) = f^{cd}(-1.a_1.\ (a_1 \times a_2).(a_1 \times a_3) \ldots\ (a_1 \times a_{m-1}).a_m.1) = a_1.(a_1 \times a_2).(a_1 \times a_3) \ldots (a_1 \times a_{m-1}).a_m\ .1 = f^{cd}(A).1$.

$\mathbf{a}_1 = \mathbf{1}$. $f^{cd}(EXT(A)) = f^{cd}(1.a_2.a_3 \ldots a_{m-1}.a_m.1) = 1.a_2\ .a_3 \ldots a_{m-1}.a_m.1 = f^{cd}(A).1$.

$\mathbf{a}_1 < \mathbf{0}$. $f^{cd}(EXT(A)) = f^{cd}(-1.(|a_1| \times a_2).(|a_1| \times a_3) \ldots\ (|a_1| \times a_{m-1}).a_m.1) = (|a_1| \times a_2).(|a_1| \times a_3) \ldots (|a_1| \times a_{m-1})\ .a_m.1 = f^{cd}(A).1$. □

When inserting a node below a leaf node with label A, we assign $EXT(A)$ to the new label.

Example 4.4: Consider the insertion of H in Figure 4.2, given that the parent of H has label 1.2.1, the label of H is $EXT(1.2.1) = 1.2.1.1$. Similaly, 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: $-2.3.6.5.3$ $(-1.3.6.5.1 +_c -1.3.6.5.2)$. □

Lemma 4.5 still holds after insertion between two negative CDDE labels (e.g. node K) because only their multipliers and local orders are added up. The parent label of the new label remains the same as the parent labels of its left and right siblings.

Correctness

Theorem 4.3. *To insert between two consecutive sibling nodes with CDDE labels: A and B where $A \prec_{cdde} B$, assigning $A +_c B$ to the new node is correct with respect to AD, PC, document order, sibling relationships and LCA computation.*

Proof. Based on f^{cd} mapping, the properties of DDE labels can be adapted for CDDE labels. Moreover, it follows from Lemma 4.6 that the addition operations of DDE and CDDE labels are equivalent. Intuitively, assigning CDDE label $A +_c B$ to the new node is equivalent to the way that DDE labeling scheme handles insertion where the new DDE label is $f^{cd}(A) + f^{cd}(B)$. Thus, its correctness becomes the immediate consequence of the correctness of DDE. □

4.3 Relationship computation

In this section, we address the issue of how the various relationships of DDE and CDDE labels can be computed efficiently.

4.3.1 DDE labels

We have shown that DDE order, along with other properties of DDE labels, are generalized forms of dewey order and other properties of Dewey labels. Given two DDE labels $A : a_1.a_2 \ldots a_m$ and $B : b_1.b_2 \ldots b_m$, they can be compared based on dewey order without any generalization if $a_1 = b_1 > 0$. Considering the fact that

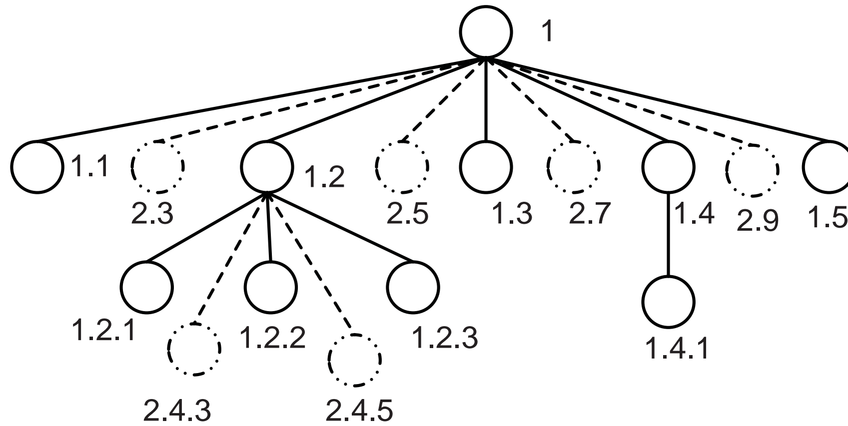


Figure 4.3: DDE labeling after uniform insertion

all the initial DDE labels start with 1, the chance that we have $a_1 = b_1 > 0$ is actually very high if the number of insertions is not too large or if the insertions are relatively uniform. As shown in Figure 4.3, if the insertions are performed uniformly between every two consecutive siblings, the new labels all have 2 as their first components. Moreover, since DDE labels can keep level information as their numbers of components after random updates, they are able to support fixed-cost computation of DDE order and other relationships even in the case of highly skewed insertions. In summary, the computation of various relationships is very efficient with DDE labels.

4.3.2 CDDE labels

The properties of CDDE, on the other hand, are defined by mapping CDDE labels to DDE labels. Therefore, it is natural to compute the various relationships between two CDDE labels by converting them to DDE labels. However, we will show that the conversion cost can actually be avoided from the following analysis.

Lemma 4.9. *Assume A, B, A', B' are four DDE labels such that $A =_e A'$ and $B =_e B'$, then A is an ancestor of B if and only if A' is an ancestor of B'. The*

same result holds for PC, document order and sibling relationships. Let C be the LCA of A and B, C' be the LCA of A' and B', we have $C =_e C'$.

Intuitively, it follows from Lemma 4.9 that any two DDE labels with equivalence relation are indeed *equivalent* in DDE labeling scheme. For example, we can replace a DDE label 2.4.6 with 1.2.3 ($2.4.6 =_e 1.2.3$), while not compromising the correctness of DDE labeling scheme.

Given a CDDE label $A : a_1.a_2.a_3 \ldots a_{m-1}.a_m$, we define a simple mapping f^{scd} : $CDDE\ label \rightarrow DDE\ label$:

$$f^{scd}(A) = \begin{cases} 1.a_2.a_3 \ldots a_{m-1}.\frac{a_m}{a_1} & when\ a_1 > 0 \\ a_2.a_3 \ldots a_{m-1}.\frac{a_m}{|a_1|} & when\ a_1 < 0 \end{cases}$$

For ease of exposition and simplicity, we allow a relaxed form of DDE labels where each component can be represented as a fraction of two decimal numbers. Note that the relaxed form is used for the purpose of comparison only.

Lemma 4.10. *Let A be a CDDE label, we have $f^{cd}(A) =_e f^{scd}(A)$.*

Proof. We consider the following two cases:

A **is a positive CDDE label.** $f^{cd}(A) = a_1.(a_1 \times a_2).\ (a_1 \times a_3) \ldots (a_1 \times a_{m-1}).a_m$ and $f^{scd}(A) = 1.a_2.a_3 \ldots a_{m-1}.\ \frac{a_m}{a_1}$. Since $\frac{a_1}{1} = \frac{a_1 \times a_2}{a_2} = \frac{a_1 \times a_3}{a_3} = \ldots = \frac{a_1 \times a_{m-1}}{a_{m-1}} = \frac{a_m}{\frac{a_m}{a_1}} = a_1$, $f^{cd}(A) =_e f^{scd}(A)$.

A **is a negative CDDE label.** $f^{cd}(A) = (|a_1| \times a_2).(|a_1| \times a_3) \ldots (|a_1| \times a_{m-1}).a_m$ and $f^{scd}(A) = a_2.a_3 \ldots a_{m-1}.\ \frac{a_m}{|a_1|}$. Since $\frac{|a_1| \times a_2}{a_2} = \frac{|a_1| \times a_3}{a_3} = \ldots = \frac{|a_1| \times a_{m-1}}{a_{m-1}} = \frac{a_m}{\frac{a_m}{|a_1|}} = |a_1|$, $f^{cd}(A) =_e f^{scd}(A)$. □

Lemma 4.9 and Lemma 4.10 together provide a very useful alternative for computing the relationships of CDDE labels. Give two CDDE labels A and B, their

relationships can be computed based on $f^{scd}(A)$ and $f^{scd}(B)$ instead of $f^{cd}(A)$ and $f^{cd}(B)$.

How sibling relationships of CDDE labels can be computed directly is given in Lemma 4.5. Other optimizations are possible if we distinguish between positive and negative CDDE labels as the following lemmas illustrate:

Lemma 4.11. *Suppose* $A : a_1.a_2.a_3 \ldots a_{m-1}.a_m$ *and* $B : b_1.b_2.b_3 \ldots b_{n-1}.b_n$ *are two positive CDDE labels, A is an ancestor of B if* $m < n$, $a_2 = b_2, \ldots a_{m-1} = b_{m-1}$ *and* $a_m = b_m \times a_1$.

Proof. Since A and B are positive CDDE labels, we have $f^{scd}(A) = 1.a_2.a_3 \ldots a_{m-1}.\frac{a_m}{a_1}$ and $f^{scd}(B) = 1.b_2.b_3 \ldots b_{n-1}\ .\frac{b_n}{b_1}$. A is an ancestor of B if $f^{scd}(A)$ is an ancestor of $f^{scd}(B)$, that is, $m < n$ and $\frac{1}{1} = \frac{a_2}{b_2} = \frac{a_3}{b_3} = \ldots = \frac{a_{m-1}}{b_{m-1}} = \frac{\frac{a_m}{a_1}}{b_m}$. Therefore, we have $a_2 = b_2, \ldots a_{m-1} = b_{m-1}$ and $a_m = b_m \times a_1$. □

Lemma 4.12. *Suppose* $A : a_1.a_2.a_3 \ldots a_{m-1}.a_m$ *and* $B : b_1.b_2.b_3 \ldots b_{n-1}.b_n$ *are two negative CDDE labels, A is an ancestor of B if* $m < n$, $\frac{a_2}{b_2} = \frac{a_3}{b_3} = \ldots = \frac{a_{m-1}}{b_{m-1}} = \frac{a_m \times b_1}{b_m \times a_1}$.

Proof. Since A and B are negative CDDE labels, we have $f^{scd}(A) = a_2.a_3 \ldots a_{m-1}.\frac{a_m}{a_1}$ and $f^{scd}(B) = b_2.b_3 \ldots b_{n-1}\ .\frac{b_n}{b_1}$. A is an ancestor of B implies that $f^{scd}(A)$ is an ancestor of $f^{scd}(B)$ and therefore, $\frac{a_2}{b_2} = \frac{a_3}{b_3} = \ldots = \frac{a_{m-1}}{b_{m-1}} = \frac{\frac{a_m}{a_1}}{\frac{b_m}{b_1}}$. Or equivalently, $\frac{a_2}{b_2} = \frac{a_3}{b_3} = \ldots = \frac{a_{m-1}}{b_{m-1}} = \frac{a_m \times b_1}{b_m \times a_1}$. □

Similarly, we can compute other relationships based on f^{scd} mappings.

4.4 Qualitative comparison

In this section, we qualitatively compare our vector order-based labeling schemes with previous labeling schemes.

Although natural order is easy to compare, it is too rigid to allow dynamic insertions without re-labeling. Lexicographical order appears to be more robust because, intuitively, both the value of each component and the number of components contribute to the ordering of labels. Insertion between two components that are consecutive in value can be accommodated by extending the number of components. However, extending the number of components appears to be an expensive operation that can lead to significant increase in the overall size. For example, QED-based labeling schemes perform poorly for ordered insertions with increase in length at 2 bits per insertion.

In addition, QED based labeling schemes come with additional encoding costs. That is, the time and computational costs spent on transforming containment, pre/post or Dewey labels to the corresponding QED codes. The process is especially complicated for Dewey labels, considering that the encoding has to be applied to every sibling group from root to leaf. Each component in ORDPATH labeling scheme, as we have seen, consists of a variable number of even numbers followed by an odd number. This fact complicates the processing of ORDPATH labels in several ways. First of all, all ORDPATH labels in the initial labeling have to skip even numbers, which makes them less compact than Dewey. Moreover, the number of components in an ORDPATH label do not necessarily reflect the level of the associated element nodes. We have to count the number of odd numbers in an ORDPATH label to derive the level information. This also leads to more complicated relationship computation such as PC and Sibling, even if the XML document does not get updated at all. Vector order, on the other hand, does not introduce additional processing complexity if there is no update because it can be reduced to natural less than relationship, nor does vector equivalence which can be reduced to natural equality.

Dataset	Size (MB)	Total No. of nodes	Max/average fan-out	Max/average depth
XMark	113	1666315	25500/3242	12/6
Nasa	23.8	476646	2435/225	10/7
Treebank	85.4	2437666	56384/1623	36/8
DBLP	127	3332130	328858/65930	6/3

Table 4.1: Test data sets

4.5 Experiments and results

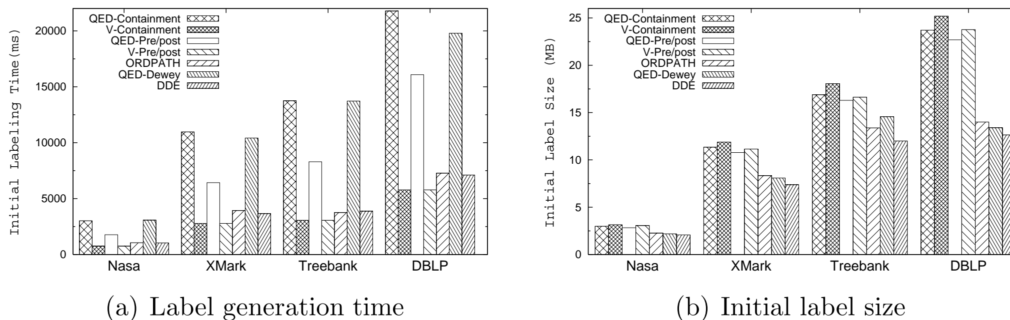
4.5.1 Experimental setup

We focus on the comparison of our vector order-based labeling schemes against QED-based labeling schemes and ORDPATH which are all persistent labeling schemes. It has been shown that persistent labeling schemes have much lower updating time than labeling schemes that require re-labeling[34]. When performing updates, we take the common approach of inserting a single node at a time. Insertion of a subtree can be achieved through a sequence of single insertions. However, the orders on which those insertions are performed do have an impact on the qualities of the resulting labeling schemes, as shown in the subsequent section.

The evaluation of these labeling scheme was performed with XMark Benchmark[4], Nasa, Treebank and DBLP [3] data sets and their characteristics are shown in Table 4.1. All the experiments were conducted on a 2.33GHz dual-core PC with 4 GB of RAM.

4.5.2 Initial labeling

The evaluation of initial labeling is shown in Figure 4.4, with measures of label generation time and label size. It can be seen that the label generation time of vector order-based and ORDPATH labeling schemes are approximately the same, which is dominated by scanning the the document once. QED-based labeling schemes have



(a) Label generation time

(b) Initial label size

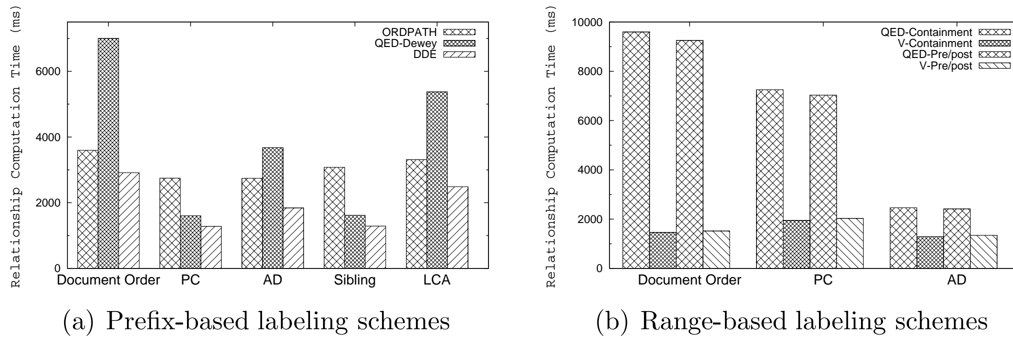
Figure 4.4: Initial Labeling

much higher label generation time, because, in addition to scanning the document, they have to perform encoding into QED codes.

The labels of vector order-based and ORDPATH labeling schemes are stored in compressed ORDPATH format[38]. QED-based labeling schemes use their own physical storage format, with 0 as the separator between every two QED codes. The label size of range-based labeling schemes is generally larger than that of prefix-based labeling schemes. For range-based labeling schemes, the label size of QED-based labeling schemes is slightly less than that of vector order-based ones. For prefix-based labeling schemes, DDE has the most compact initial label size for all the four data sets.

4.5.3 Querying static document

We test the query performance on all the four data sets. We present the results from Treebank data set as the other three data sets shown similar trends. Without any updates, the labels used for processing queries remain the same as the initial labels. We evaluate the query performance on initial labels by computing the most commonly used five relationships: document order, AD, PC, sibling and LCA. We choose the first 10000 labels from the initial labels of Treebank data set in document order and, for each pair of the labels, we compute all the five relationships. Note



(a) Prefix-based labeling schemes (b) Range-based labeling schemes

Figure 4.5: Querying initial labels

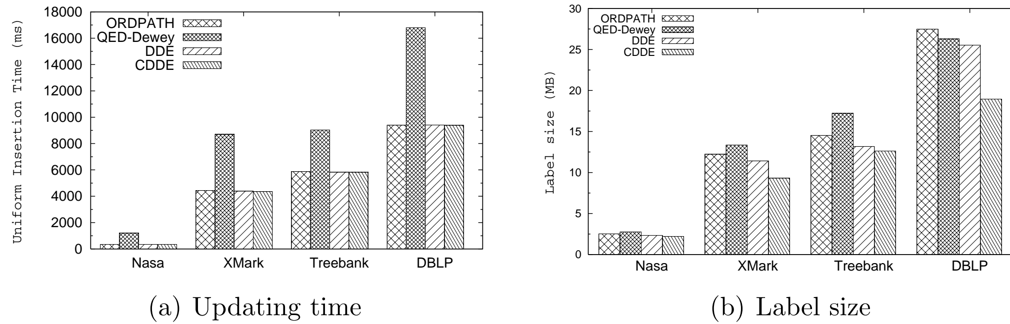
that as pointed out in [43], the LCA of a set of nodes is effectively the LCA of the first and the last node of the set in document order. Therefore we consider computing the LCA of two labels as a common function instead of many labels.

The querying time for prefix-based labeling schemes are shown in Figure4.5 (a) on all the five relationships. CDDE is not shown here because its performance is the same as DDE for static documents. While QED-Dewey is more efficient than ORDPATH for computing PC and sibling relationships, it is significantly slower for comparing document order and less efficient for AD relationship and LCA computation. For all the five relationships, our DDE outperforms ORDPATH and QED-Dewey.

Range-based labeling schemes are evaluated based on three relationships including document order, AD and PC. Sibling and LCA are excluded because they are not supported by range-based labeling schemes. Results in Figure 4.5 (b) show that V-Containment and V-Prefix support the three relations more efficiently than QED-based labeling schemes.

4.5.4 Update processing

For update processing, successive insertions are performed through a sequence of single insertions. For insertion of a new node, we retrieve its two closest nodes in



(a) Updating time (b) Label size

Figure 4.6: Uniform insertions

terms of document order (e.g. its left and right siblings if they are leaf nodes). A new label is then generated based on the insertion algorithm specific to that labeling scheme. Then we create a dummy node with the new label and insert it to the database we have.

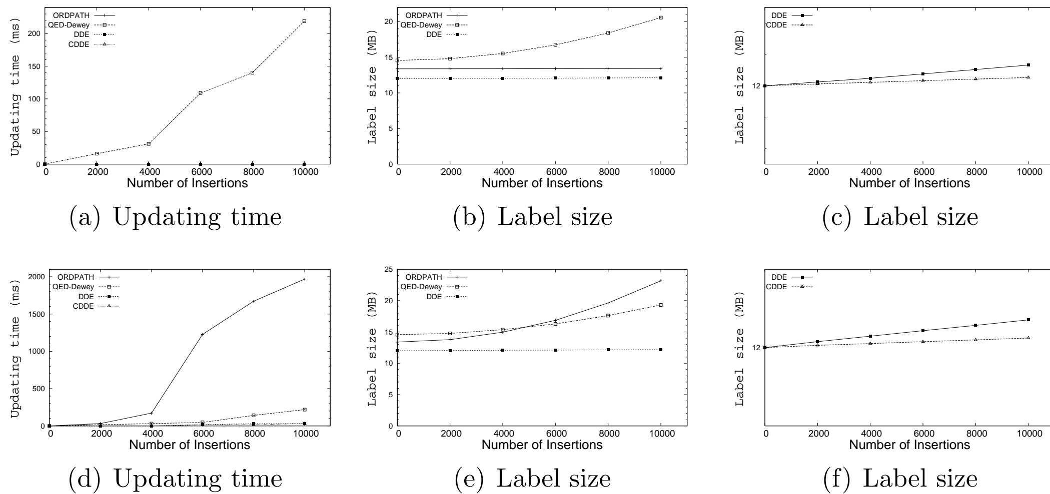
Uniform insertions

We test with insertions made uniformly between every two consecutive siblings. How these labeling schemes respond to uniform insertions is shown in Figure 4.6. The insertion time of ORDPATH is approximately the same as our DDE and CDDE whereas QED shows a slower updating time, as illustrated in Figure 4.6 (a). In Figure 4.6 (b), the comparison of label size after uniform insertions remains similar to that for the initial labels (Figure 4.4 (b)), with CDDE giving the most compact labels. The comparison of range-based labeling schemes and query performance after uniform insertions are ignored here, since the quality of these labeling schemes is not much affected by uniform insertions.

Skewed insertions

We classify skewed insertions into two different cases that are common in practice:

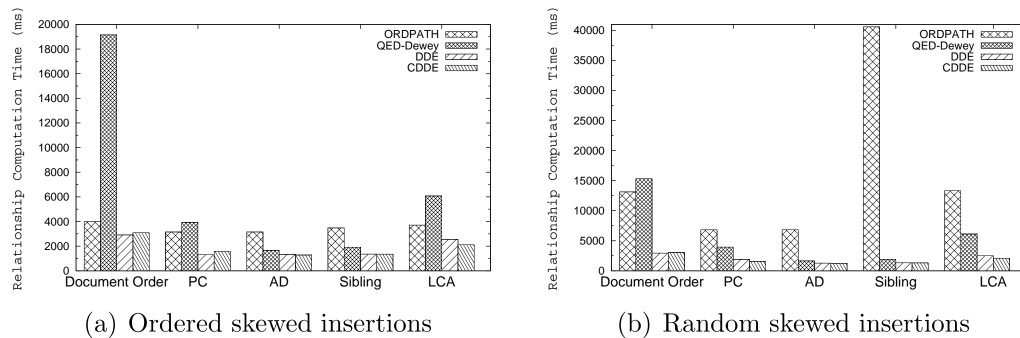
- **Ordered skewed insertion** refers to repeatedly inserting before or after a



(a) Updating time (b) Label size (c) Label size

(d) Updating time (e) Label size (f) Label size

Figure 4.7: Comparison of prefix-based labeling schemes after skewed insertions



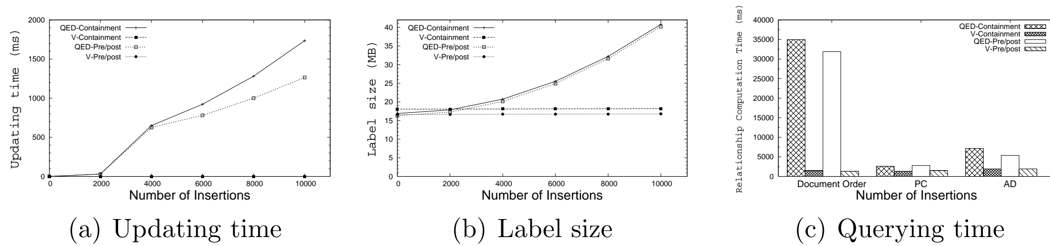
(a) Ordered skewed insertions (b) Random skewed insertions

Figure 4.8: Relationship computation time after skewed insertions

particular node.

- **Random skewed insertion** refers to repeatedly inserting between two nodes in random order.

Compared with uniform insertions, skewed insertions can have a more significant impact on the resulting qualities of labels. Figure 4.7 (a) (b) and (c) shows the updating cost and label size after ordered skewed insertions. The insertion time of ORDPATH, DDE and CDDE are negligible and their label sizes only increase slightly. In contrast, QED-Dewey has relatively higher updating time and its label size has shown a much higher increase. This result conforms to our previous dis-



(a) Updating time (b) Label size (c) Querying time

Figure 4.9: Comparison of range-based labeling schemes after skewed insertions

cussions that the lengths of QED codes can increase at 1 or 2 bits per insertion in case of ordered skewed insertion, resulting in the fast increase of the overall label size. The results for random skewed insertions are shown in Figure 4.7 (d), (e) and (f). The updating time and label size of ORDPATH increase at a much faster rate than the other labeling schemes. This is because random skewed insertions greatly increase the amount of 'caret's that are needed to be used in ORDPATH labels. For both types of insertions, our DDE and CDDE have shown the best performance in terms of updating time and label size. In addition, the label size of CDDE increases at a slower rate than DDE, which is what we have expected. Figure4.9 shows the response of range-based labeling schemes to ordered skewed insertions. The result for random skewed insertions is similar. It can be seen that V-Containment and V-Prefix labeling schemes are little affected by ordered insertion sequence while QED encoded range-based labeling schemes have shown much higher growth rate in label size.

4.5.5 Querying dynamic document

To compare the query performance on dynamic XML documents, we adopt the same settings as the static case except the 10000 labels chosen include 2000 labels that are newly inserted. Figure 4.8 (a) gives the comparison of relationship computation time after ordered skewed insertions. Given the fast increase of QED-Dewey

label size, it conforms to our expectation that its query response time also increases significantly, especially for document order. The comparison after random skewed insertions is shown in Figure 4.8 (b) where the query response time of ORDPATH increases significantly, particularly for sibling relationship. Nevertheless, our DDE and CDDE have demonstrated robust performance regardless of the order and number of insertions. Their query response times are least affected after both types of skewed insertions. We have similar observation for range-based labeling schemes in Figure 4.9 (c).

4.6 Summary of chapter

Since updates are usually unpredictable, we emphasize that a single labeling scheme should be used for both static and dynamic XML documents. Previous dynamic XML labeling schemes, however, all suffer from the complexity introduced by their insertion techniques even if there is little/no update. In this chapter, we have presented a novel labeling scheme called DDE which is designed with both static and dynamic settings in mind. DDE, in the static setting, is the same as Dewey labeling scheme which designed for static XML documents. In addition, based on an extension of vector order, DDE allows dynamic updates without re-labeling when updates take place. We introduce a variant of DDE, namely CDDE, which is derived from DDE labeling scheme from a one-to-one mapping. Compared with DDE, CDDE labeling scheme shows slower growth in label size for frequent insertions. Both DDE and CDDE have exhibited high resilience to skewed insertions in which case the qualities of existing labeling schemes degrade severely. Extensive experimental evaluation has demonstrated the benefits of our proposed labeling schemes over previous approaches.

Chapter 5

Search Tree-based (ST) encoding techniques for range-based labeling schemes

Labeling schemes can be mainly classified as range-based and prefix-based labeling schemes. The previous Chapter focused on improving the application of vector order to prefix-based labeling schemes. In this Chapter, we tackle the problem of improving range-based labeling schemes.

We begin by introducing the insertion-based encoding approach adopted by previous works[32][33][34] in Section 5.1 and dynamic formats in Section 5.2. In Section 5.3, we introduce our Search Tree-based (ST) encoding technique which is designed for efficient and scalable order-preserving transformation.

5.1 Insertion-based encoding algorithms

The following example illustrates the applications of QED encoding scheme to containment labeling scheme.

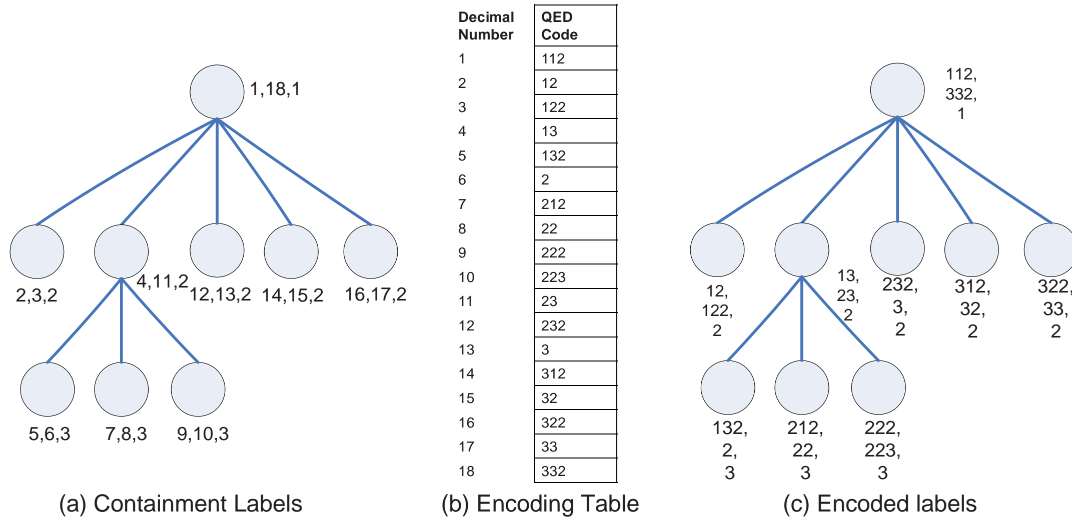


Figure 5.1: Applying QED encoding scheme to containment labeling scheme

Example 5.1: In Figure 5.1 (a), every node in the XML tree is labeled with a containment label that consists of three values: *start*, *end* and *level*. When applying QED encoding scheme, the *start* and *end* values are encoded with QED codes based on the encoding table in (b). As a result, the containment labels are transformed into QED-Containment labels shown in (c), which not only preserve the property of containment labels, but also allows dynamic insertions with respect to lexicographical order. □

Formally speaking, we consider an encoding scheme as a mapping f from the original labels to the target labels. Let X and Y denote the set of order-sensitive codes in the original labels and target labels respectively, f maps each element x in X to an element $y = f(x)$ in Y. For the mapping to be both correct and effective, f should satisfy the following properties:

1. **Order Preserving**: The target labels must preserve the order of the original labels, i.e. $f(x_i) < f(x_j)$ if and only if $x_i < x_j$ for any x_i, $x_j \in$ X. An order preserving transformation ensures that both document order and structural information are kept correctly.

2. **Optimal Size**: To reduce the storage cost and optimize query performance, the target labels should be of optimal size, i.e. the total size of $f(x_i)$ should be be minimized for a given range. To satisfy this property, f has to take the range to be encoded into consideration. The mappings are different for different ranges of different documents.

The following example illustrates how this mapping in Figure 5.1 (b) is derived based on QED encoding scheme.

Example 5.2: To create the encoding table in Figure 5.1 (b), QED encoding scheme first extends the encoding range to (0, 19) and assigns two empty QED codes to positions 0 and 19 (they are discarded after the encoding process). Next, the $(1/3)^{th}$ (6=round(0+(19-0)/3)) and $(2/3)^{th}$ (13=round(0+(19-0)×2/3)) positions are encoded by applying an insertion algorithm with the QED codes of positions 0 and 19 as input. The QED insertion algorithm takes two QED codes as input and computes two QED codes that are lexicographically between them which are as short as possible (such insertions are always possible because QED codes are dynamic). The output QED codes are assigned to the $(1/3)^{th}$ and $(2/3)^{th}$ positions which are then used to partition range (0, 19) into three sub-ranges. This process is recursively applied for each of the three sub-ranges until all the positions are assigned QED codes. CDBS and recursive Vector encoding schemes adopt similar algorithms. □

We classify these algorithms as *insertion-based* approach since they make use of the property that the target labels allow dynamic insertions. However, a drawback of the insertion-based approach is that by assuming the entire encoding table fits into memory, it may fail to process large XML documents due to memory constraint. Since the size of the encoding table can be prohibitively large for large XML documents and main memory remains the limiting resource, it is desirable to have

a memory efficient encoding algorithm. Moreover, the insertion-based approach requires costly table creation for every range, which is computationally inefficient for encoding multiple ranges of multiple documents.

In this chapter, we show that only a single encoding table is needed for the encoding of multiple ranges. As a result, encoding a range can be translated into indexing mapping of the encoding table which is not only very efficient, but also has an adjustable memory usage. The main contributions of this chapter include:

- We propose a novel Search Tree-based (ST) encoding technique which has a wide application domain. We illustrate how ST encoding technique can be applied to binary string, quaternary string and vector codes, and prove the optimality of our results.
- We introduce encoding table compression which can be seamlessly integrated into our ST encoding techniques to adapt to the amount of memory available.
- We propose Tree Partitioning (TP) technique to further enhance the performance of ST encoding for multiple documents.
- Experimental results demonstrate the high efficiency and scalability of our ST encoding techniques.

5.2 Dynamic Formats

We have introduced binary strings and quaternary strings in chapter 2. In this section, we illustrate in details how insertions can be processed with them.

5.2.1 Binary strings

[33][34] introduced the following theorem which formalizes the dynamic property of binary strings that end with 1.

Theorem 5.1. *Given two binary strings C_l and C_r which both end with 1 such that C_l precedes C_r in lexicographical order (denoted as $C_l \prec C_r$), we can always find C_m which also ends with 1 and $C_l \prec C_m \prec C_r$.*

Theorem 5.1 can be proved based on Algorithm 2.

Algorithm 2: InsertBinaryString(C_l, C_r)

```
  Data: C_l and C_r which are both binary strings that end with 1 and C_l ≺ C_r
  Result: C_m which ends with 1 and C_l ≺ C_m ≺ C_r
1 if length(C_l) ≥ length(C_r) then
2     C_m = C_l ⊕ 1                          /* ⊕ means concatenation */;
3 end
4 else  C_m = C_r with the last number 1 change to 01;
5 return C_m;
```

Example 5.3: Let 0101, 011 and 0111 be three binary strings, we have 0101 $\prec$ 011 $\prec$ 0111. Insertion between 0101 and 011 will produce 01011, since length(0101) $>$ length(011) (0101$\oplus$1, Algorithm 2 line 2). And insertion between 011 and 0111 leads to 01101, since length(011) $<$ length(0111) (0111 with the last 1 change to 01, Algorithm 2 line 3). □

5.2.2 Quaternary strings

[32] introduced the following theorem which formalizes the dynamic properties of quaternary strings.

Theorem 5.2. *Given two quaternary strings C_l and C_r which end with 2 or 3 and $C_l \prec C_r$, we can always find C_m which is also a quaternary string such that $C_l \prec C_m \prec C_r$.*

From the results in [32], C_m can be derived from Algorithm 3 whose proof of the correctness can be found in [34].

Algorithm 3: InsertQuaternaryString(C_l, C_r)

```
   Data: C_l and C_r which are two quaternary strings that end with 2 or 3 and
         C_l ≺ C_r
   Result: C_m which ends with 2 or 3 and C_l ≺ C_m ≺ C_r
 1 if length(C_l) > length(C_r) then
 2     if C_l ends with 2 then
 3         C_m = C_l with the last number 2 change to 3;
 4     else
 5         C_m = C_l⊕ 2                       /* ⊕ means concatenation */;
 6     end
 7 end
 8 if length(C_l) = length(C_r) then  C_m = C_l⊕ 2;
 9 else  C_m = C_r with the last number change to 12;
10 return C_m
```

Example 5.4: Let 222, 223 and 23 be three quaternary strings, we have 222 $\prec$ 223 $\prec$ 23 based on lexicographical order. Insertion between 222 and 223 leads to 2222 since length(222) = length(223) (222$\oplus$2, Algorithm 3, line 6). And insertion between 223 and 23 produces 2232 since length(223) > length(23) and 223 ends with 3 (223$\oplus$2, Algorithm 3, line 5). □

5.3 ST Encoding Technique

In this section, we present the details of our ST encoding technique which can be applied to binary string, quaternary string and vector codes and are called STB, STQ and STV encoding schemes respectively.

5.3.1 Seach Tree-based Binary (STB) encoding

Data structure

Our STB encoding is based by the data structure we call STB tree.

Definition 5.1 (STB tree)**.** *An STB tree is a complete binary tree where each node is associated with an STB code which is a binary string that ends with 1.*

The STB code of the root is 1.

Let n be a node in the STB tree, we denote the STB code associated with it as C_n. Given a node n in the STB tree, the STB code of its left child lc and right child rc can be derived as follows:

- C_{lc}=C_n with the last 1 replaced with 01
- C_{rc}=$C_n \oplus$ 1 ($\oplus$ means concatenation)

Two STB trees with 6 and 12 nodes are shown in Figure 5.2 (b) and (c).

Lemma 5.1. *The left subtree of a node n contains only STB codes lexicographically less than C_n; The right subtree of n contains only STB codes lexicographically greater than C_n.*

Proof. [**Sketch**] Given any STB code n which is a binary string that ends with 1, we denote C_n as "S1" where "S" is a binary string or an empty string. It follows that C_{lc}="S01" and similarly, $C_{lc.lc}$="S001" and $C_{lc.rc}$="S011". Now it is easy to see that all the STB codes in the left subtree have "S0" as their prefix. Since "S0" precedes "S1" in lexicographical order, all the STB codes in the left subtree are lexicographically less than C_n. The rest of the lemma follows similarly. □

Theorem 5.3. *An STB tree is a complete binary search tree based on lexicographical order.*

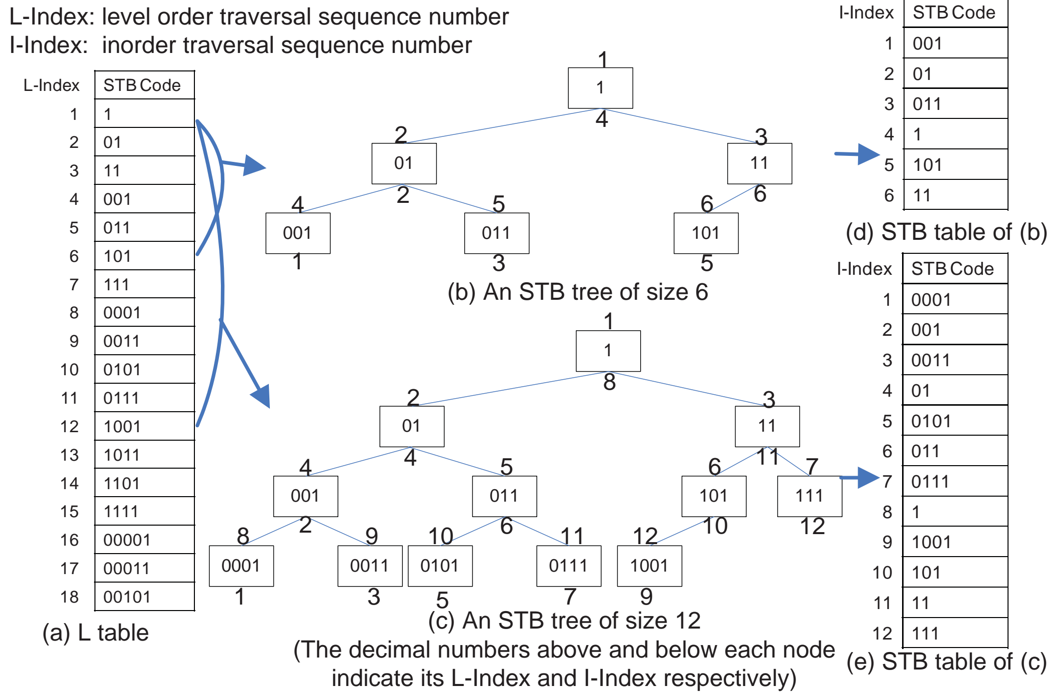


Figure 5.2: STB encoding of two ranges 6 and 12

Proof. Theorem 5.3 follows directly from Lemma 5.1. □

Given an STB tree, an **L table** stores its STB codes based on *Level order traversal sequence*. We denote the index of an L table as **L-Index** and use L to denote the set of decimal numbers in L-Index. An important observation about L table is that it can be shared by STB trees of different sizes: the first m rows of the L table represent an STB tree of size m in level order.

An **STB table** orders the STB codes in an STB tree based on *Inorder traversal sequence*. Note that an STB table represents the result of STB encoding and need not be physically stored. We denote the index of an STB table as **I-Index** and use I to denote the set of decimal numbers in I-Index.

Example 5.5: Consider the two STB tree of size 6 and 12 in Figure 5.2 (b) and (c). If we order their codes according to level order traversal sequence, they match the first 6 and 12 rows of the L table in (a). Their corresponding STB tables are

shown in (d) and (e). □

Algorithms

Given a range m, the goal of STB encoding is to realize the mappings represented by an STB table of size m. Intuitively, this can be achieved by traversing the STB tree of size m in inorder.

Formally speaking, STB encoding defines a mapping $f : I \rightarrow B$ where B denotes the set of STB codes. More specifically, f is established through two levels of mappings: $f(i) = h(g(i))$ where $g : I \rightarrow L$ and $h : L \rightarrow B$. Deriving h is straight forward from the L table. Depending on the range to be encoded, the size of L table can be extended dynamically.

How g can be established is shown in Algorithm 4 which is based on inorder traversal of a binary tree. First a stack *path* is initialized to store the L-Indices of a root-to-leaf path(line 1). Then we proceed to call Function **PushLeftPath** which pushes the L-Index of the leftmost path (starting from the root) into *path* (line 2). For each $i \in I$, we map i to the top element in *path* (Recall that during an inorder traversal, the leftmost element is always visited first). Then the L-Index of the leftmost path that starts from the right child of the top element is pushed into *path* (line 3 to 6).

Next we show that STB encoding is order preserving and of optimal size.

Theorem 5.4. *To encode a range m with STB encoding, let C_j and C_k be the encoded STB codes for j and k where $1 \leq j < k \leq m$, it follows that $C_j \prec C_k$.*

Proof. Since an STB tree is a binary search tree (Theorem 5.3), an inorder traversal of the STB tree visits the STB codes in increasing lexicographical order. In other words, STB encoding is order preserving. □

Algorithm 4: ItoLMapping(m)

```
Data: m which is the range to be encoded.
Result: The mapping from I-Index to L-Index stored in an array
        ItoL[1...m].
1 Initialize Stack path;
2 PushLeftPath(path, 1, m);
3 for i=1 to m do
4     l=path.Pop();
5     ItoL[i] = l;
6     PushLeftPath(path, 2 × l + 1, m)    /* 2 × l + 1 ⟶ right child */
7 end
```

Function PushLeftPath($path$, l, m)

```
while l ≤ m do
    path.Push(l);
    l = 2 × l                              /* 2 × l ⟶ left child */
end
```

Lemma 5.2. *Level i of an STB tree has 2^{i-1} STB codes (except possibly the last level) of length i. (Assume the root is of level 1).*

Lemma 5.2 easily follows from the properties of STB trees.

Since an STB code is a binary string that ends with 1, there are 2^{i-1} possible STB codes of length i. From Lemma 5.2, we can see that an STB tree has all the possible STB codes of length i at level i (except possibly the lowest level). The fact that an STB tree is a complete binary tree implies that STB codes with length i are always used up before STB codes with length $i+1$ are used. Therefore STB encoding produces labels with optimal size.

5.3.2 Seach Tree-based Quaternary (STQ) encoding

We illustrate our STQ encoding scheme using the data structure we call STQ tree. An **STQ tree** is a complete ternary tree. Each node of the STQ tree is associated with two STQ codes: left code (L) and right code (R) where $R = L$ with the last

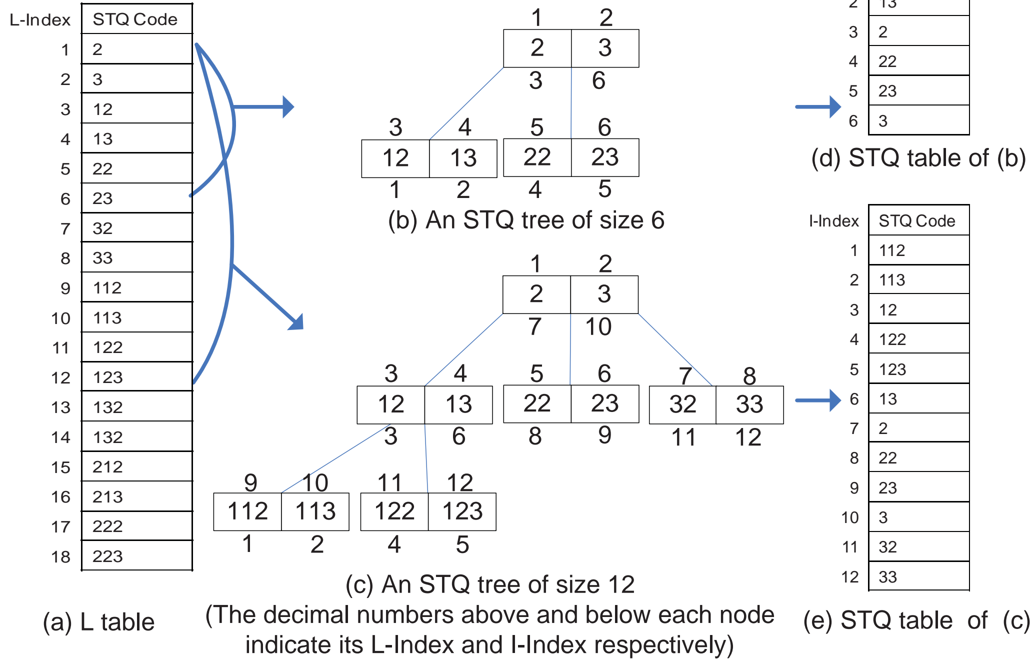


Figure 5.3: STQ Encoding of two ranges 6 and 12

number 2 change to 3. L and R of the root are 2 and 3 respectively.

Given a node n in the STQ tree, the left code of its left child (lc), middle child (mc) and right child (rc) can be derived as follows:

- L_{lc}= L_n with the last number 2 change to 12;
- L_{mc}= $L_n \oplus$ 2
- L_{rc}= $R_n \oplus$ 2 ($\oplus$ means concatenation).

In all cases, we have $R = L$ with the last number 2 change to 3.

Two STQ trees with 6 and 12 codes are shown in Figure 5.3 (b) and (c).

Lemma 5.3. *The left subtree of a node n contains only STQ codes lexicographically less than L_n; The middle subtree of n contains only STQ codes lexicographically*

between L_n and R_n; The right subtree of n contains only STQ codes lexicographically greater than R_n.

The proof is similar to that of Lemma 5.1, so we omit it here. Given Lemma 5.3, an STQ tree can be seen as a search tree if we define the inorder traversal sequence to be in order of: (1) Traverse the left subtree; (2) Visit L of the root; (3) Traverse the middle subtree; (4) Visit R of the root and (5) Traverse the right subtree. In this way, we can define **I-Index**, **L-Index**, **STQ table** and **L table** similar to those of STB tree.

Algorithm 6: ItoLMapping(m)

```
Data: m which is range to be encoded.
Result: The mapping from I-Index to L-Index stored in an array
        ItoL[1...m].
1  Initialize Stack path;
2  PushLeftPath(path, 1, m);
3  for i=1 to m do
4      l=path.Pop();
5      ItoL[1...m] = l;
6      if l mod 2 =1 then                          /* l ⟶ lcode */
7          PushLeftPath(path, 3×l+2, m)   /* 3×l+2 ⟶ middle child
           */
8      else                                        /* l ⟶ rcode */
9          PushLeftPath(path, 3×l+1, m) /* 3×l+1 ⟶ right child */
10     end
11 end
```

Function PushLeftPath($path$, l, m)

```
while l ≤ m do
    path.Push(l+1);
    path.Push(l);
    l = 3×l                              /* 3×l ⟶ left child */
end
```

STQ encoding defines the mapping from I-Index to STQ codes which is achieved through two levels of mappings: from I-Index to L-Index and from L-Index to STQ

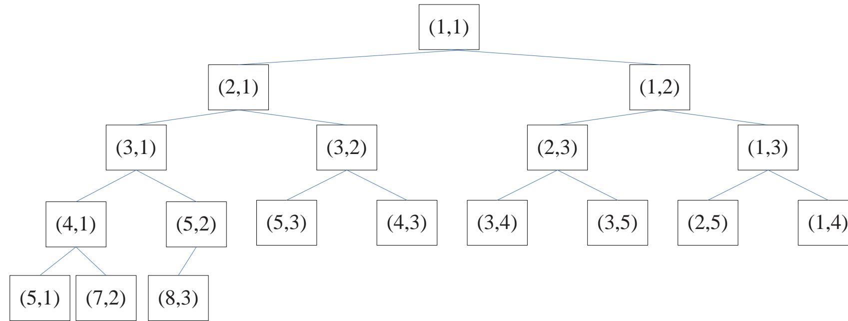


Figure 5.4: STV tree

codes. As shown in Figure 5.3, the mappings from L-Index to STQ codes are stored a single L table (a) which can be shared by multiple ranges. The mappings from I-Index to L-Index can be derived from Algorithm 6 which performs an inorder traversal of the STQ tree.

The correctness of our STQ encoding algorithms follows from the fact that its inorder traversal visits the STQ codes in increasing lexicographical order. The resulting label size is also optimal because our algorithm favors STQ codes with smaller lengths.

5.3.3 Search Tree-based Vector (STV) encoding

Our STV encoding scheme is based on the data structure we call STV tree. It is a complete binary tree where each node is associated with a vector code: C. The vector codes of the root, its left child and right child are (1,1), (2,1) and (1,2) respectively.

Given a node n and its parent p in the STV tree, the vector codes of its left child (lc) and right child (rc) can be derived as follows: If n is the left child of p, C_{lc}=2 $\times$ C_n - C_p; C_{rc}=C_n + C_p; Else, C_{lc}=C_n + C_p; C_{rc}=2 $\times$ C_n - C_p. An example of STV tree is shown in Figure 5.4.

Theorem 5.5. *An STV tree is a binary search tree based on vector order.*

The proof is based on mathematical induction, we omit it here. Given the STV tree, we can define L table similar to that of STB encoding which stores the mapping from L index to Vector codes. Moreover, since STV tree is a binary search, Algorithm 4 can be directly applied to derive the mapping from I to L index. We ignore the details of STV encoding, given that it is similar to STB encoding.

5.3.4 Comparison with insertion-based approach

Compared with the insertion-based approach, our design of ST encoding as a two level mapping has the following advantages: (1) Since $h : L \rightarrow STB/STQcode$ remains the same for different ranges, the cost of encoding a new range is only to compute $g : I \rightarrow L$. By sharing h for different ranges, we avoid costly table creation for every range; (2) Compression technique can be conveniently applied to L table to provide high flexibility of memory usage (Section 5.4). The compression technique is easily incorporable because compressing L table only affects h while h and g are independent of each other; (3) By exploiting the common mappings of different ranges, we can further speed up the encoding of multiple ranges (Section 5.5).

5.4 Encoding Table Compression

The L table of STB is shown in Figure 5.5 (a). Considering its STB codes with indices from 2 onwards, we can see that every STB code at index $2i + 1$ can be deduced from the STB code at index $2i$ by changing the second last number to 1. Therefore we can compress this L table to half by only retaining the rows with even

L	STB Code
1	1
2	01
3	11
4	001
5	011
6	101
7	111
8	0001
9	0011
10	0101
11	0111
12	1001
13	1011
14	1101
15	1111
16	00001
17	00011
18	00101

(a) The original L table of STB

L	STB Code
1	01
2	001
3	101
4	0001
5	0101
6	1001
7	1101
8	00001
9	00101

(b) Compressed L table with *C*=1

L	STB Code
1	001
2	0001
3	1001
4	00001

(c) Compressed L table with *C*=2

L	STQ Code
1	2
2	3
3	12
4	13
5	22
6	23
7	32
8	33
9	112
10	113
11	122
12	123
13	132
14	133
15	212
16	213
17	222
18	223

(d) The original L table of STQ

L	STQ Code
1	2
2	12
3	22
4	32
5	112
6	122
7	132
8	212
9	222

(e) Compressed L table with *C*=0

L	STQ Code
1	12
2	112
3	212

(f) Compressed L table with *C*=1

Figure 5.5: Compress L tables of STB and STQ by factors of 2^C and 2×3^C respectively

indices ((b)). Thus, the mapping from L-Index to STB codes for becomes:

$$h(l) \rightarrow \begin{cases} LTable[l/2] & , when\ l\ mod\ 2 = 0 \\ \\ LTable[\lfloor l/2 \rfloor] with\ the\ sec\text{-} \\ ond\ last\ number\ change\ to\ 1 & , when\ l\ mod\ 2 = 1 \end{cases} \tag{5.1}$$

The table in (b) can be further compressed by a factor of 2 if we consider the STB codes with indices from 2 onwards. We exclude the STB codes with odd indices since they can be derived from the STB codes with even indices by changing the *third* last number to 1 ((c)). In this way, we can compress the L table of STB by factors of $2, 4, 8 \ldots 2^C$ and we denote C as the compression factor.

By analyzing the L table of STQ in Figure 5.5 (d), the straight forward com-

pression is to exclude the STQ codes with even indices since they can be derived from the STQ codes with odd indices by changing the last 2 to 3 ((b)). Therefore the mapping from L-Index to STQ codes becomes:

$$h(l) \rightarrow \begin{cases} LTable[\lceil l/2 \rceil] & , when\ l\ mod\ 2 = 1 \\ \\ LTable[l/2]\ with\ the \\ last\ number\ change\ to\ 3 & , when\ l\ mod\ 2 = 0 \end{cases} \tag{5.2}$$

Consider the table in Figure 5.5 (e), it can be further compressed by a factor of 3 if we consider the STQ codes from index 2 onwards. The STQ codes at indices $3i$ and $3i+1$ can be derived from the STQ code at index $3i-1$ by changing the second last number to 2 and 3. Therefore we exclude the STQ codes at indices $3i$ and $3i+1$ and the resulting table is shown in (f). In summary the L table of STQ can be compressed by factors of $2, 6, 18 \ldots 2 \times 3^C$.

5.5 Tree Partitioning (TP)

STB encoding technique, as we have shown, is a mapping $f(i) = h(g(i))$ where $g : I \rightarrow L$ and $h : L \rightarrow B$. Since h remains the same for different ranges, the cost of encoding a range is dominated by g. The motivation for TP optimization is that, given multiple ranges to be encoded, the computational cost of g can be reduced if we can exploit the common mappings for ranges that are close to some extent.

We introduce Tree Partitioning (TP) to exploit these common mappings, thus further enhancing the performance of ST encoding techniques. We use STB tree to illustrate the idea of TP. Our optimization technique can be easily adapted for STQ and STV trees.

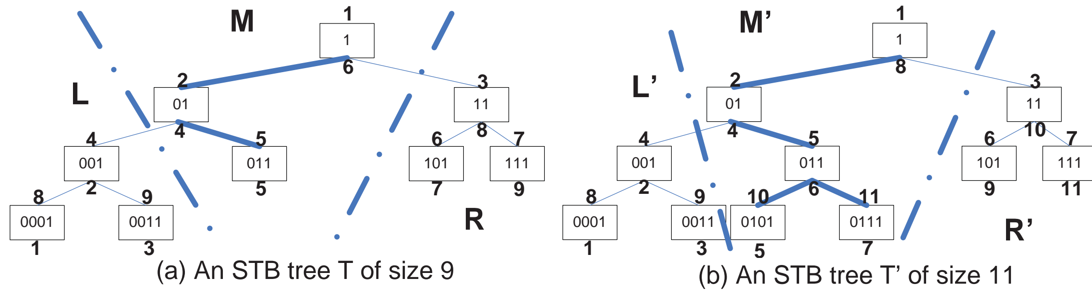


Figure 5.6: Tree partitioning

Suppose there are two STB trees T of size s_1 and T' of size s_2 (without loss of generality, we assume $s_1 < s_2$), we analyze the common mapping of the two trees when they have the same height, say k, i.e. $2^k \leq s_1 < s_2 < 2^{k+1}$.

Our TP algorithm divides T' into three partitions:

L' partition All the nodes on the left of the path from the root to the node with L-Index=$s_1 + 1$.

R' partition All the nodes on the right of the path from the root to the node with L-Index=s_2

M' partition The rest of the nodes in the STB tree

T is also divided into three partitions: L, R and M. L' and L partitions have the same L-Index and so do R' and R partitions. And the rest of the nodes fall into M'. g in L and L' partitions are the same as the two partitions overlap and are visited first during inorder traversal. If we increase all the I-Index in R by $s_2 - s_1$, g in R and R' also coincide.

Example 5.6: Two STB trees T and T' in Figure 5.6 (a) and (b) are partitioned based on our TP algorithm. In the resulting partitions, g in L and L' are the same. g in region R can be derived from that in R' if we increase the L-Index in R by $11 - 9 = 2$. □

Since both M and M' bounded by two root-to-leaf paths, Algorithm 4 can be easily modified to compute the mappings in them. Here we give formulas of how an intermediate state can be computed directly for the STB tree. That is, given any number $i \in I$, we can compute its mapping $g(i)$.

Assume the range is m, a=2 and b=1. For simplicity, we define the following constants.

$$s = m - a^{\lfloor \log_a m \rfloor} + 1 \tag{5.3}$$

$$h = \begin{cases} \lfloor \log_a m \rfloor \quad , & when \quad i \leqslant a/b\ s \\ \\ \lfloor \log_a m \rfloor - 1 \quad , & when \quad i > a/b\ s \end{cases} \tag{5.4}$$

$$j = \begin{cases} i \quad , & when \quad i \leqslant a/b\ s \\ \\ i - s \quad , & when \quad i > a/b\ s \end{cases} \tag{5.5}$$

$$p = MaxPower(a, j) \tag{5.6}$$

Where $MaxPower$ is a function that takes two integers a and j as input. Assume $j = x \times a^p$ where x is not divisible by a and p is a natural number, $MaxPower$ returns p as the output.

The following equation determines $l = g(i)$.

$$l = a^{h-p} + \left\lfloor bj/a^{p+1} \right\rfloor \tag{5.7}$$

Example 5.7: Suppose m=18 and let $l = 10$. From Equation 5.3, we have $s = 3$.

Data set	Max/average fan-out	Max/average depth	No. of nodes
XMark	25500/3242	12/6	179689
Treebank	56384/1623	36/8	1666315
SwissProt	50000/301	5/3	2437666
DBLP	328858/65930	6/3	3332130

Table 5.1: Test data sets

Since $10 > 3 \times 2$, we have $h = 3$, $j = 7$ and $p = 0$. Then from Equation 5.7, we have $l = 11$. As the result, 10 in the I-Index maps to 11 in the L-Index. □

Computing $g(i)$ for STQ trees is similar except that a=3 and b=2.

By partitioning the range to be encoded, we can re-use some of the previously-computed mappings and avoid re-computing g for the whole range.

5.6 Experiments and Results

In this section, we experimentally evaluate and compare the various encoding techniques developed in this chapter against the insertion-based encoding schemes including CDBS and QED.

We used data sets from XMark benchmark, Treebank, SwissProt and DBLP datasets for our experiments. The characteristic of these data sets are shown in Table 5.1. We used JAVA for our implementation and our experiments are performed on Pentium IV 3 GHz with 1G of RAM running on windows XP.

5.6.1 Encoding Time

First we evaluate the encoding time of these encoding schemes using containment labels of the XMark data set. We randomly generated 80 XMark documents whose sizes range from 1 MB to 90 MB. The documents are encoded in random order. We run the experiment three times and take the average time. In Figure 5.7, we

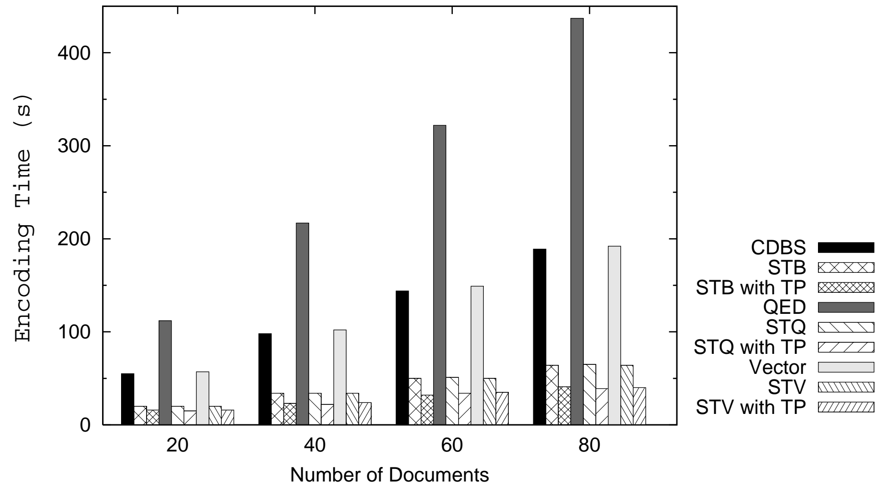
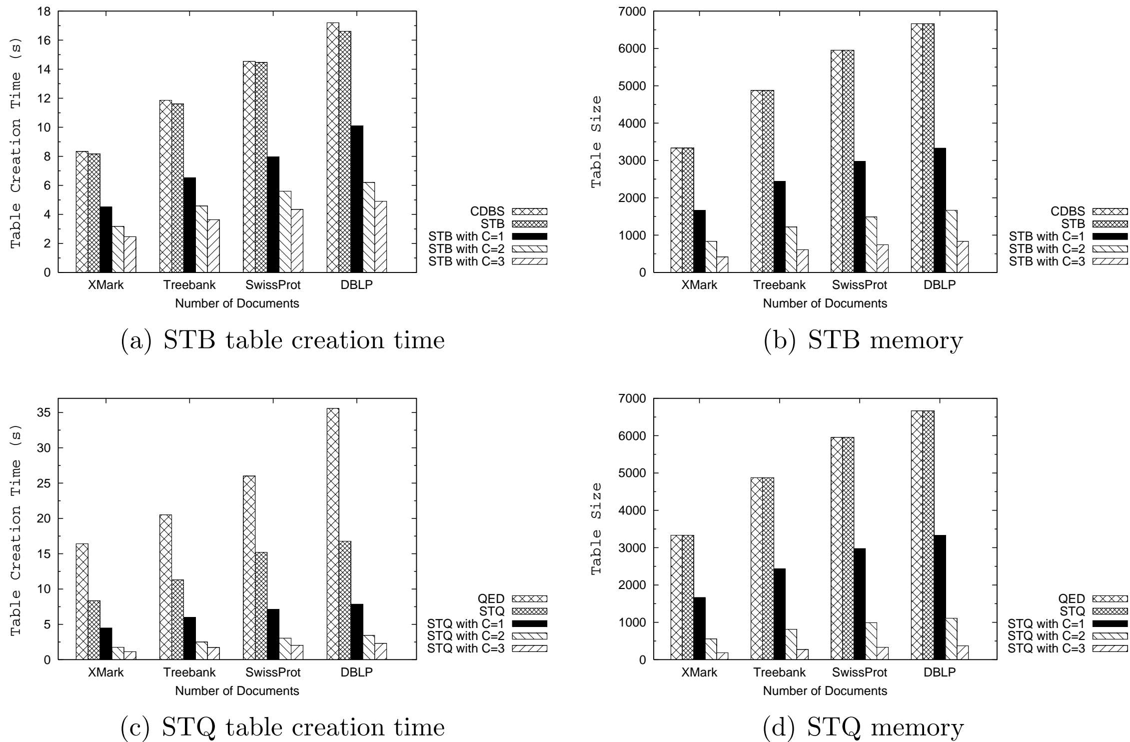


Figure 5.7: Encoding containment labels of multiple documents

observe clear time difference between ST encodings and insertion-based encodings: our STB and STV encoding is approximately 3 times faster than CDBS encoding and recursive Vector encoding. Moreover, our STQ encoding is approximately 7 times faster than QED encoding. The reason is clear from the comparison of algorithms: insertion-based encodings need to create an encoding table for every range, which is significantly slower than our ST encodings that perform index mapping of a single table. The advantages of ST encoding are more significant when we apply TP optimization which exploits common mappings of encoding multiple ranges. Overall ST encodings with TP are by a factor of 5-11 times faster than insertion-based encodings for containment labels. The results confirm that our ST encoding techniques are highly efficient for encoding multiple ranges and substantially surpass the insertion-based encodings.

5.6.2 Memory Usage and Encoding Table Compression

We compare the memory usage of different algorithms which is dominated by the size of the encoding tables and the results are shown in Figure 5.8. Without any



(a) STB table creation time

(b) STB memory

(c) STQ table creation time

(d) STQ memory

Figure 5.8: Encoding table compression

compression, the table size of STB and CDBS are the same, and so are their table creation times. However, unlike CDBS whose table size is fixed, our STB encoding can adjust its table size by varying the compression factor C. A larger C yields a smaller table size and less table creation time. Similar observation can be made in Figure 5.8 (c) and (d) for quaternary strings. The table creation time of STQ is less than that of QED due to the complexity of the QED insertion algorithms. By adjusting the compression factor, our ST encoding can process large XML data sets with limited memory available.

5.6.3 Label size and query performance

We have proved that both STB and STQ encodings produce labels of optimal sizes. The label sizes of STV and recursive Vector encoding differ by only a small amount,

which is overall negligible. Moreover, since the labels produced by ST encoding and its insertion-based counterpart are of the same format, their query performance is also the same. In summary, the labels produced by our ST encoding techniques are of optimal quality.

5.7 Summary of chapter

In this chapter, we take the initiative to address the problem of efficient label encoding to make range-based labeling schemes dynamic. When encoding multiple ranges for multiple documents, previous insertion-based algorithms need to create an encoding table for every range, resulting in high computational and memory costs. We propose ST encoding techniques which can be widely applied to existing dynamic formats and generate dynamic labels with optimal size. ST encoding techniques use only a single encoding table to encode multiple ranges and are therefore highly efficient. Moreover, complemented by encoding table compression, our ST encoding techniques are able to process very large XML documents with limited memory available.

Chapter 6

Conclusion

We summarize this thesis in this chapter and outline on the future work.

6.1 Summary of order-centric approach

In this thesis, we have developed an order-centric perspective on existing XML labeling schemes.

We summarize the orders of the different labeling schemes in Table 6.1.

Among range-based labeling schemes, Containment, Pre/post labeling schemes are designed for static XML documents. Their labels are ordered by natural order and require frequent re-labeling for insertions. QRS-Containment and QRS-Pre/post use floating point numbers instead of integers, which are still ordered by natural order and only delay re-labeling to some extent. Among range-based labeling schemes, those based on lexicographical order, VLEI order and vector order allow dynamic updates without re-labeling, thus greatly reducing the update costs. Note that VLEI order is similar to lexicographical order where both number of components and their values contribute to the ordering.

Prefix-based labeling schemes can be transformed into dynamic labeling schemes

Labeling scheme	Order	Component-wise equality	Component-wise order	Relabel
Containment	natural	NA	NA	Y
Pre/post	natural	NA	NA	Y
QRS-Containment	natural	NA	NA	Y
QRS-Pre/post	natural	NA	NA	Y
Prime	natural	NA	NA	Y
CDBS-Containment	lex	NA	NA	N
CDBS-Pre/post	lex	NA	NA	N
VLEI-Containment	VLEI	NA	NA	N
VLEI-Pre/post	VLEI	NA	NA	N
QED-Containment	lex	NA	NA	N
QED-Pre/post	lex	NA	NA	N
V-Containment	vector	NA	NA	N
V-Pre/post	vector	NA	NA	N
Dewey	lex	natural	natural	Y
QRS-Dewey	lex	natural	natural	Y
VLEI-Dewey	generalized lex	natural	VLEI	N
QED-Dewey	generalized lex	natural	lex	N
ORDPATH	generalized lex	natural	lex	N
V-Prefix	generalized lex	natural	vector	N
DDE	generalized lex	v-equivalence	vector	N
CDDE	generalized lex	v-equivalence	vector	N

Table 6.1: Summary of orders of different labeling schemes

if their component-wise order is lexicographical order, VLEI order or vector order. We have shown how generalized lexicographical order can be used to characterize existing prefix-based dynamic labeling schemes.

In Prime labeling scheme, tree structure and document order are encoded separately. To insert a new node with prime labeling scheme, an unused prime number can be used, without affecting other labels. Meanwhile, document orders are still ordered by natural order, requiring re-ordering whenever a node is inserted or deleted. In this sense, Prime labeling scheme is only dynamic for *unordered tree-structured* data.

In addition to different orders, it is worth noting the inherent differences between prefix-based and range-based labeling schemes. Compared with range-based labeling schemes, an obvious advantage of prefix-based labeling schemes is its ability to determine Sibling and LCA relationships. However, the performance of prefix-based labeling schemes is sensitive to the structure of the XML documents as the size of a prefix label increases linearly with its level. Range-based labeling scheme, on the other hand, perform consistently regardless of the depth of the XML tree.

Although natural order is easy to compare, it is too rigid to allow dynamic insertions without re-labeling. Lexicographical order and VLEI order appear to be more robust because, intuitively, both the value of each component and the number of components contribute to the ordering of labels. Insertion between two components that are consecutive in value can be accommodated by extending the number of components. However, frequent extensions of components can lead to significant increase in the overall size. For example, QED-based labeling schemes perform poorly for ordered insertions with increase in length at 2 bits per insertion.

In addition, QED based labeling schemes come with additional encoding costs. That is, the time and computational costs spent on transforming containment,

pre/post or Dewey labels to the corresponding QED codes. The process is especially complicated for Dewey labels, considering that the encoding has to be applied to every sibling group from root to leaf. Each component in ORDPATH labeling scheme, as we have seen, consists of a variable number of even numbers followed by an odd number. This fact complicates the processing of ORDPATH labels in several ways. First of all, all ORDPATH labels in the initial labeling have to skip even numbers, which makes them less compact than Dewey. Moreover, the number of components in an ORDPATH label do not necessarily reflect the level of the associated element nodes. We have to count the number of odd numbers in an ORDPATH label to derive the level information. This also leads to more complicated relationship computation such as PC and Sibling, even if the XML document does not get updated at all.

Based on extensive analysis of previous labeling schemes, our observation is that they all come with considerable costs even for documents that are not updated at all. To solve this problem we introduce vector order, which, as illustrated in Table 6.1, is different from orders adopted by all previous approach including natural order, lexicographical order or VLEI order. We show that vector order is widely applicable to both range-based and prefix-based labeling schemes and the resulting labeling schemes have compact size and high query performance, while being able to avoid re-labeling when updating.

To further improve the application of vector order to prefix-based labeling schemes, we extend the concept of vector order and propose Dynamic DEwey (DDE) labeling scheme. DDE, in the static setting, is the same as Dewey labeling scheme which has the most compact label size among all the labeling schemes we compare. Moreover, DDE labels can be queried in the same way as Dewey labels for static documents, which is highly efficient. Based on an extension of vector

order, DDE allows dynamic updates without re-labeling when updates take place. In addition, we introduce a variant of DDE, namely CDDE, which is derived from DDE labeling scheme from a one-to-one mapping. Compared with DDE, CDDE labeling scheme shows slower growth in label size for frequent insertions. Both DDE and CDDE have exhibited high resilience to skewed insertions in which case the qualities of existing labeling schemes degrade severely. Extensive experimental evaluation has demonstrated the benefits of our proposed labeling schemes over previous approaches.

From the order perspective, transforming static labeling schemes into dynamic ones is to transform natural order to some other order in an *order-preserving* manner. It guarantees both tree structure and document order are kept correct. When encoding multiple ranges for multiple documents, previous insertion-based algorithms need to create an encoding table for every range, resulting in high computational and memory costs. We propose ST encoding techniques which can be widely applied to existing dynamic formats and generate dynamic labels with optimal size. ST encoding techniques use only a single encoding table to encode multiple ranges and are therefore highly efficient. Moreover, complemented by encoding table compression, our ST encoding techniques are able to process very large XML documents with limited memory available.

6.2 Future work

The order framework proposed in this thesis paves the way for future research on this topic. Our separation of encoding tree structure and document order provides an opportunity to adapt our vector order-based encoding techniques for other problems involving order-sensitive updates. In addition, new orders can be proposed to

encode document order with new characteristics.

Another promising future research direction is to study how to label and update XML documents of more complex models. In addition to tree structure, extensive research have focused on labeling Directed Acyclic Graph (DAG) to answer reachability queries and distance queries[16, 17, 19, 24, 27–29, 40, 41, 50]. Because DAGs are generally much more complex than trees and generating labels is more expensive, there has also been research work on how to iteratively recompute labels in response to updates[11]. Labeling DAG is closely related to labeling tree structure because tree structure can be considered a special subset of DAG, where reachability queries would be translated to Ancestor/Descendant queries and distance queries are equivalent to Ancestor/Descendant queries plus computing the level difference.

Although existing works claim labeling DAG can be applied to XML documents of general graph model, none of them has taken document order into consideration. Therefore, how to encode orders and process order-sensitive updates for XML documents modeled as DAG remain challenging future research topics.

Bibliography

[1] DocBook. http://www.docbook.org.

[2] EDGAR. http://www.sec.gov/edgar.shtml.

[3] University of Washington XML Repository. http://www.cs.washington.edu/research/xmldatasets/.

[4] XMark - An XML Benchmark Project. http://monetdb.cwi.nl/xml/downloads.html.

[5] World Wide Web Consortium. Document Object Model (DOM) Level 3 Core Specification. *W3C Recommendation* (2001).

[6] Abiteboul, S., Alstrup, S., Kaplan, H., Milo, T., and Rauhe, T. Compact Labeling Scheme for Ancestor Queries. *SIAM J. Comput.* (2006).

[7] Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. L. The lorel query language for semistructured data. *Int. J. on Digital Libraries 1*, 1 (1997), 68–88.

[8] Agrawal, R., Borgida, A., and Jagadish, H. V. Efficient management of transitive relationships in large data and knowledge bases. *SIGMOD Rec. 18*, 2 (1989), 253–262.

[9] Amagasa, T., Yoshikawa, M., and Uemura, S. QRS: A Robust Numbering Scheme for XML Documents. In *ICDE* (2003).

[10] Beyer, K. S., Cochrane, R., Hvizdos, M., Josifovski, V., Kleewein, J., Lapis, G., Lohman, G. M., Lyle, R., Nicola, M., Özcan, F., Pirahesh, H., Seemann, N., Singh, A., Truong, T. C., der Linden, R. C. V., Vickery, B., Zhang, C., and Zhang, G. Db2 goes hybrid: Integrating native xml and xquery with relational data and sql. *IBM Systems Journal 45*, 2 (2006).

[11] Bramandia, R., Choi, B., and Ng, W. K. On incremental maintenance of 2-hop labeling of graphs. In *WWW* (2008).

[12] Bray, T., Paoli, J., C.M.Sperberg-McQueen, E.Maler, and Yergeau, F. Extensible Markup Language (XML) 1.0: fourth edition *W3C recommendation*, 2006.

[13] Ceri, S., Comai, S., Damiani, E., Fraternali, P., Paraboschi, S., and Tanca, L. Xml-gl: A graphical language for querying and restructuring xml documents. In *WWW* (1999).

[14] Chamberlin, D., Florescu, D., Robie, J., Simon, J., and Stefanescu, M. XQuery: A Query Language for XML W3C working draft. *World Wide Web Consortium* (2001).

[15] Chamberlin, D., Robie, J., and Florescu, D. Quilt: An xml query language for heterogeneous data sources. In *WebDB* (2000).

[16] Cheng, J., Yu, J. X., Lin, X., Wang, H., and Yu, P. S. Fast computation of reachability labeling for large graphs. In *EDBT* (2006).

[17] Cheng, J., Yu, J. X., Lin, X., Wang, H., and Yu, P. S. Fast computing reachability labelings for large graphs with high compression rate. In *EDBT* (2008).

[18] Clark, J., and DeRose, S. XML Path Language (XPath) version 1.0 w3c recommendation. *World Wide Web Consortium* (1999).

[19] Cohen, E., Halperin, E., Kaplan, H., and Zwick, U. Reachability and distance queries via 2-hop labels. *SIAM J. Comput. 32*, 5 (2003), 1338–1355.

[20] Cohen, E., Kaplan, H., and Milo, T. Labeling Dynamic XML Trees. In *SPDS* (2002).

[21] Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. A query language for xml. In *WWW* (1999).

[22] Dietz, P. F. Maintaining Order in a Linked List. In *In Annual ACM Symposium on Theory of Computing* (1982).

[23] Goldman, R., and Widom, J. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB* (1997).

[24] Jagadish, H. V. A compression technique to materialize transitive closure. *ACM Trans. Database Syst. 15*, 4 (1990), 558–598.

[25] Jason, M., and Jennifer, W. Query optimization for xml. In *VLDB* (1999).

[26] Jayavel, S., Eugene, S., Jerry, K., Rajasekar, K., Efstratios, V., Jeffrey, N., and Igor, T. A general technique for querying xml documents using a relational database system. *SIGMOD Rec. 30*, 3 (2001).

[27] Jiefeng, C., and Xu, Y. J. On-line exact shortest distance query processing. In *EDBT* (2009).

[28] Jin, R., Xiang, Y., Ruan, N., and Fuhry, D. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD* (2009).

[29] Jin, R., Xiang, Y., Ruan, N., and Wang, H. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD* (2008).

[30] Kaplan, H., Milo, T., and Shabo, R. A comparison of labeling schemes for ancestor queries. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms* (2002), pp. 954–963.

[31] Kobayashi, K., Liang, W., Kobayashi, D., Watanabe, A., and Yokota, H. Vlei code: An efficient labeling method for handling xml documents in an rdb. In *ICDE* (2005).

[32] Li, C., and Ling, T. W. QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. In *CIKM* (2005).

[33] Li, C., Ling, T. W., and Hu, M. Efficient Processing of Updates in Dynamic XML Data. In *ICDE* (2006).

[34] Li, C., Ling, T. W., and Hu, M. Efficient Updates in Dynamic XML Data: from Binary String to Quaternary String. *VLDB J.* (2008).

[35] Li, Q., and Moon, B. Indexing and Querying XML Data for Regular Path Expressions. In *VLDB* (2001).

[36] Mchugh, J., Widom, J., Abiteboul, S., Luo, Q., and Rajaraman, A. Indexing semistructured data. Tech. rep., 1998.

[37] Michael, R. Xml and relational database management systems: inside microsoft®sql server™2005. In *SIGMOD* (2005).

[38] O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., and Westbury, N. ORDPATHs: Insert-friendly XML Node Labels. In *SIGMOD* (2004).

[39] P., B., Freire J., R. P., and J, S. From xml schema to relations: A cost-based approach to xml storage. In *ICDE* (2002).

[40] R., A., A., B., and V., J. H. Efficient management of transitive relationships in large data and knowledge bases. *SIGMOD Rec. 18*, 2 (1989), 253–262.

[41] Schenkel, R., Theobald, A., and Weikum, G. Hopi: An efficient connection index for complex xml document collections. In *EDBT* (2004).

[42] Silberstein, A., He, H., Yi, K., and Yang, J. BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data. In *ICDE* (2005).

[43] Sun, C., Chan, C.-Y., and Goenka, A. K. Multiway SLCA-based Keyword Search in XML Data. In *WWW* (2007).

[44] Tatarinov, I., Viglas, S., Beyer, K. S., Shanmugasundaram, J., Shekita, E. J., and Zhang, C. Storing and Querying Ordered XML Using a Relational Database System. In *SIGMOD* (2002).

[45] Wu, X., Lee, M. L., and Hsu, W. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *ICDE* (2004).

[46] Xu, L., Bao, Z., and Ling, T. W. A Dynamic Labeling Scheme Using Vectors. In *DEXA* (2007).

[47] Xu, L., Ling, T. W., Bao, Z., and Wu, H. Efficient Label Encoding for Range-based Dynamic XML Labeling Schemes. In *DASFAA* (2010).

[48] Xu, L., Ling, T. W., and Wu, H. Labeling Dynamic XML Documents-An Order-Centric Approach. *TKDE J.* (2010).

[49] Xu, L., Ling, T. W., Wu, H., and Bao, Z. DDE: From Dewey to a Fully Dynamic XML Labeling Scheme. In *SIGMOD* (2009).

[50] Yangjun, C., and Yibin, C. An efficient algorithm for answering graph reachability queries. In *ICDE* (2008).

[51] Zhang, C., Naughton, J. F., DeWitt, D. J., Luo, Q., and Lohman, G. M. On Supporting Containment Queries in Relational Database Management Systems. In *SIGMOD* (2001).