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

QUERYING AND UPDATING XML DATA BASED ON NODE LABELING SCHEMES

LI CHANGQING
(Master of Engineering, Peking University, China)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgements

First of all, I gratefully acknowledge the persistent support and encouragement from my supervisor, Professor Ling Tok Wang. Prof Ling patiently guided and advised me throughout the various phases of my research. His meticulosity greatly impressed me which makes me think thoroughly and do carefully. Not only has Prof Ling provided constant academic guidance to my research, he also gave me suggestions on how to overcome the difficulties that I met in my life. There is a famous Chinese saying "One day's teacher is your father for your whole life". To me, Prof Ling is a great supervisor and my second father in my life.

I wish to express my deep gratitude to Dr Ang Chuan Heng and Dr Chan Chee Yong for serving on my thesis evaluation committees. Thank them for going through such a long document and giving me valuable feedbacks. Their comments on my thesis are precious. Great thanks to all the reviewers who have read or will read this thesis.

It is also my pleasure to express my thanks to Dr Lee Mong Li and Dr Wynne Hsu who gave me a chance to do research work together with them. Their guidance and suggestions are important to my future research.

Dr Gary Tan Soon Huat, who gave me valuable suggestions on my research. The several months that I worked together with him gave me an unforgettable research experience.

I also want to thank all the academic and administrative staffs in School of Computing, Register Office, and Office of Student Affairs of National University of Singapore for their help in different areas of my life in the these years.

In my lab, I have to acknowledge the support and friendship I received from so many friends: Wu Xiaodong, Lu Jiaheng, Chen Ting, Ni Wei, He Qi, Chen Zhuo, Chen Yabing, Yang Xia, Jiao Enhua, Yu Tian, Zhang Wei, Xia Chenyi, Xiang Shili, Li Yingguang, Ni Yuan, Cheng Weiwei, Hu Jing and many others not appearing here.

On a personal note, it is important for me to thank my wife, Hu, for her love and support during my Ph.D. study and for her braveness to give the birth to our baby, in July, 2005, which makes our life happy. I am also grateful to my parents for their efforts to bring me up and provide me with the best possible education, to my parents-in-law for their help in taking care of my wife.

Summary

The method of assigning labels to the nodes of an XML tree is called a node labeling (or numbering) scheme. Based on the labels only, both ordered and un-ordered queries can be processed without accessing the original XML file. The core issue for XML query is to efficiently determine the following four basic relationships: ancestor-descendant (A-D), parent-child (P-C), sibling and ordering relationships.

The existing node labeling schemes, i.e. containment, prefix and prime number schemes, are not efficient to determine all the four basic relationships. For instance, the containment scheme is very *inefficient* to determine the sibling relationship; it needs to search the parent of a node, then decide whether another node is a child of this parent; the search of the parent needs a lot of parent-child relationship determinations which is very expensive. The prefix scheme is efficient to determine all the four basic relationships if the XML tree is shallow, however when the XML tree becomes deeper, the prefix scheme becomes not efficient because the labels of the prefix scheme become longer and the comparisons of node labels become *expensive*. The prime number scheme has very large label size and it employs the modular and division operations to determine the relationships which is *expensive*. Thus in this thesis, we firstly propose the P-Containment scheme which can *determine*

all the four basic relationships efficiently no matter what XML structure is. In addition, P-Containment is used to efficiently process the internal node updates and to completely avoid re-labeling.

One more important point for the labeling scheme is to process updates when nodes are inserted into or deleted from the XML tree. All the existing node labeling schemes, i.e. containment, prefix and prime number schemes, have high update cost, therefore in this thesis we propose a **novel *Compact Dynamic*** Binary String (CDBS) encoding to encode the labels of different labeling schemes and based on CDBS encoding, updates can be efficiently processed. CDBS encoding has two important properties which form the foundations of this thesis: (1) CDBS compares codes based on the lexicographical order, and it supports that codes can be inserted between any two *consecutive* CDBS codes ***with the orders kept and without re-encoding*** the existing numbers; (2) CDBS is orthogonal to specific labeling schemes, e.g. containment, prefix and prime number schemes, thus it can be applied ***broadly*** to different labeling schemes or other applications to efficiently process the updates. Moreover, because the fixed size length field of CDBS will encounter the *overflow* problem, we improve CDBS to Compact Dynamic *Quaternary* String (CDQS) encoding. Though the label size of CDQS is larger and its update cost is larger, it can ***completely*** avoid re-labeling in XML updates no matter what labeling schemes XML data employs.

We report the experimental results to show that CDBS and CDQS encodings are superior to previous approaches to process updates in terms of the number of nodes to re-label (none for CDQS) and the time for updating. When P-Containment

scheme is combined with CDBS (for intermittent updates and uniformly frequent updates) or CDQS (completely avoid re-labeling) encoding, both queries and updates can be efficiently processed.

Table of Contents

Acknowledgements .... **ii**

Summary .... **iv**

1 Introduction **1**

1.1 Background 1

1.1.1 XML 2

1.1.2 XML Technologies 3

1.1.3 XML Query 4

1.1.4 XML Update 6

1.2 Problem Statement and Motivation 7

1.3 Overview of Contributions 8

1.4 Organization of Thesis 10

2 Background and Related Works **12**

2.1 Node Labeling Schemes 13

2.1.1 Containment Labeling Scheme 13

2.1.2 Prefix Labeling Scheme 18

2.1.3 Prime Labeling Scheme 24

2.2 Encoding Approaches to Store the Labels of Labeling Schemes....................29

2.2.1 Binary Number Encodings....................29

2.2.2 UTF8 Encoding....................30

2.2.3 OrdPath Encodings....................31

2.2.4 Binary String and Quaternary String Encodings....................33

2.3 Summary....................34

3 P-Containment Scheme....................38

3.1 A Node Labeling Scheme: P-Containment Scheme....................39

3.2 Summary....................42

4 CDBS Encoding of Node Labels to Efficiently Process XML Updates....................44

4.1 Lexicographical Order for Binary Strings....................45

4.2 The Compact Dynamic Binary String Encoding (CDBS)....................49

4.2.1 CDBS Encoding Algorithm....................54

4.2.2 Size Analysis....................56

4.3 Applying CDBS to Different Labeling Schemes....................58

4.4 Processing of XML Updates Based on Different Labeling Schemes Encoded with CDBS....................62

4.4.1 Leaf Node Updates....................63

4.4.2 Internal Node Updates....................66

4.4.3 Subtree Updates....................71

4.4.4 Uniformly and Skewed Frequent Updates....................73

4.5 Experimental Evaluation and Comparisons....................74

4.5.1 Experimental Setup 74
4.5.2 Performance Study on Static XML Data....76
4.5.3 Performance Study on Intermittent Updates in Dynamic XML Data....82
4.5.4 Summary of Experimental Results....88
4.6 Summary 89

5 CDQS Encoding of Node Labels to Completely Avoid Re-labeling 91

5.1 The Compact Dynamic Quaternary String Encoding (CDQS) for Node Labels 92
5.1.1 CDQS Encoding Algorithm 95
5.1.2 Size Analysis....97
5.2 Applying CDQS to Different Labeling Schemes....98
5.3 Completely Avoiding Re-labeling in XML Updates 102
5.4 Extensions of CDQS 105
5.5 Experimental Evaluation and Comparisons 105
5.5.1 Performance Study on Static XML Data....105
5.5.2 Performance Study on Frequent Updates in Dynamic XML Data....108
5.5.3 Performance Study on CDOS and CDHS 113
5.6 Summary 114

6 Controlling the Increase in Label Size 116

6.1 Finding the Codes with the Smallest Size between Two Codes 117
6.2 Handling Insertion Skew....123
6.3 Experimental Evaluation....124

6.3.1 Comparisons of Algorithm 4.1 and Algorithm 6.1 125

6.3.2 Processing the Skewed Insertion 126

6.4 Summary 127

7 Conclusion 129

7.1 Summary of Contributions 129

7.2 Future Works 132

Appendices 133

Appendix A: Meanings of Abbreviations 133

Appendix B: Calculation of the SC Value for Prime Scheme 134

Appendix C: Size Calculations for V-CDBS and CDQS 136

C1: Calculation of the Total Code Size for V-CDBS 136

C2: Calculation of the Total Code Size for CDQS 136

Appendix D: Calculation of the Positions Based on V-CDBS 138

Appendix E: Publications During Ph.D. Period 139

Bibliography 142

List of Tables

Table 2.1: UTF8 encoding 30

Table 2.2: OrdPath1 encoding..32

Table 2.3: OrdPath2 encoding..32

Table 2.4: Comparisons on queries..36

Table 2.5: Comparisons on updates 37

Table 4.1: Binary and CDBS encodings 50

Table 4.2: Test datasets 75

Table 4.3: Test queries on the scaled D1 79

Table 4.4: Number of nodes to re-label in leaf node updates 83

Table 4.5: Number of nodes to re-label for internal node updates.......................86

Table 5.1: CDQS encoding 93

Table 6.1: V-CDBS encoding 117

List of Figures

Figure 1.1: An XML document example 3

Figure 1.2: An ordered XML tree 5

Figure 2.1: Dietz's containment scheme using preorder and postorder.......................15

Figure 2.2: Li's containment scheme with order and interval size 15

Figure 2.3: Zhang's containment scheme 15

Figure 2.4: DeweyID prefix scheme 19

Figure 2.5: BinaryString prefix scheme...21

Figure 2.6: OrdPath prefix scheme 22

Figure 2.7: Prime scheme..26

Figure 3.1: The existing containment scheme and P-Containment scheme.................40

Figure 4.1: V-CDBS-Containment scheme...60

Figure 4.2: V-CDBS-Prefix scheme (for Figure 2.4)..60

Figure 4.3: Leaf node insertions based on V-CDBS-Prefix scheme............................63

Figure 4.4: Leaf node insertions based on V-CDBS-Containment scheme.................64

Figure 4.5: Leaf node insertions based on the existing prefix scheme 65

Figure 4.6: Leaf node insertions based on the existing containment scheme 65

Figure 4.7: V-CDBS-P-Containment scheme 67

Figure 4.8: Internal node insertions based on V-CDBS-P-Containment scheme 69

Figure 4.9: Internal node insertions based on the prime number scheme 70

Figure 4.10: Subtree insertion based on V-CDBS-Prefix scheme 72

Figure 4.11: Subtree insertion based on V-CDBS-P-Containment scheme.................73

Figure 4.12: Label sizes of different labeling schemes..78

Figure 4.13: Query performance of different labeling schemes..................................80

Figure 4.14: Log2 of total time (CPU time + I/O time) for leaf node updates 83

Figure 4.15: Log2 of total time (CPU time + I/O time) for internal node updates 86

Figure 4.16: Label size increasing speed when inserting subtrees...............................88

Figure 5.1: CDQS-P-Containment scheme 99

Figure 5.2: CDQS-Prefix scheme...100

Figure 5.3: Insertions based on CDQS-P-Containment scheme 102

Figure 5.4: Insertions based on CDQS-Prefix scheme...104

Figure 5.5: Label sizes of different labeling schemes 106

Figure 5.6: Response time of different queries based on different labeling schemes 107

Figure 5.7: Uniformly frequent updates 110

Figure 5.8: Skewed frequent updates 112

Figure 5.9: Label sizes of different labeling schemes 114

Figure 6.1: Comparison of Algorithm 4.1 and Algorithm 6.1 for CDBS in the update environment with both insertions and deletions...126

Figure 6.2: Processing of skewed insertions 127

Chapter 1

Introduction

Since the eXtensible Markup Language (XML) [10] emerged as a new standard for information representation and exchange on the Web, the problems of storing, indexing, querying and updating XML documents have been among the major issues of database research. In this thesis, we mainly research on how to improve the query efficiency of the existing labeling schemes for XML data, and more important we propose novel techniques to efficiently update XML data.

In this chapter, we firstly introduce the background of XML related technologies in Section 1.1. Next in Section 1.2 we outline the objective of this thesis. The main contributions of this thesis are summarized in Section 1.3, and Section 1.4 describes the whole organization of this thesis.

1.1 Background

In this section, we present XML related technologies.

1.1.1 XML

The eXtensible Markup Language (XML) [10] is a representation language as well as an exchange language. As a representation language, XML was originally designed as a new document format for large-scale electronic publishing, which is derived from the Standard Generalized Markup Language (SGML). As an exchange language, XML has played and is now still playing an increasingly important role in the exchange of a wide variety of data on the Web. This is because XML can describe both structured and semi-structured data. In addition, XML is extensible, platform-independent, and fully Unicode compliant.

We use an example to illustrate what is an XML.

Example 1.1 *Figure 1.1 depicts a simple XML document. XML identifies data using tags, which are identifiers enclosed in angle brackets. Collectively, the tags are known as "markup". XML document in Figure 1.1 starts with a prolog markup that identifies the document as an XML document that conforms to version 1.0 of XML specification and uses the 8-bit Unicode character encoding scheme. Next, there is one line of comments, which will be ignored by XML parsers. After that, "<doc>…</doc>" is an element, and it is the root of the document. Generally, each XML document has a single root element. In Figure 1.1, "<student_employee ID="HD1234567">…</student_employee>" is also an element. The "ID" in this element is an attribute and the "HD1234567" is the value of the attribute "ID". Similarly "<name>John</name>" etc. are also elements, however they are nested in the "student_employee" element. "John" is the value or content of the element "name".*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- An XML document about student_employee, courses and part_time -->

<doc>
  <student_employee ID="HD1234567">
    <name>John</name>
    <contact_no>9876543</contact_no>
    <course ID="CS4321">
      <name>database</name>
    </course>
    <part_time>
      <position>programmer</position>
    </part_time>
  </student_employee>
</doc>
```

Figure 1.1: An XML document example

As the relationships between elements in an XML document are defined by nested structures, XML documents are often modeled as trees.

1.1.2 XML Technologies

XML support is being added to existing database management systems (DBMSs) and native XML systems are being developed both in industry and in academia. XBench [77] is a family of XML benchmarks which can capture diverse application domains in different XML DBMSs very well. To efficiently manipulate, structure, and transform XML, some XML related technologies are developed. They are:

- **XML schema languages.** An XML schema language is used to describe the structure and content of an XML document. There are several schema languages existing for XML. Currently, XML DTD and XML Schema Definition Language [38] (XSD) from W3C are widely accepted.

- **Tree model-based APIs.** An XML document is represented as a tree of nodes with a tree model API. Typically, it loads an XML document in memory all at once. The dominant tree model API is the W3C Document Object Model (DOM) [37]. Developers can use the DOM for programmatic reading, manipulation and modification of an XML document.
- **Event-driven APIs.** An event-driven API processes an XML document without storing much more than the context of the current node being processed in memory. The most popular event-driven API is the Simple API for XML (SAX) [36].

This thesis focuses on how to efficiently query and update XML data no matter XML data are schema oblivious or schema-conscious. SAX will be used in the implementation to parse XML file in XML query and update processing.

1.1.3 XML Query

In the definition of XML, one element is allowed to refer to another, therefore theoretically an XML is a graph. However for simplicity, most of the researches [1, 23, 56, 64, 74, 80, 83] process queries over XML data that conform to an *ordered tree-structured* data model. With the tree model, data objects, e.g. elements, attributes, text data, etc., are modeled as the nodes of a tree, and relationships are modeled as the edges to connect the nodes of the tree. Without loss of generality, in this thesis, we also omit the references in XML, and all queries are based on *the ordered tree-structured* representation of XML data. Figure 1.3 shows an ordered XML tree.

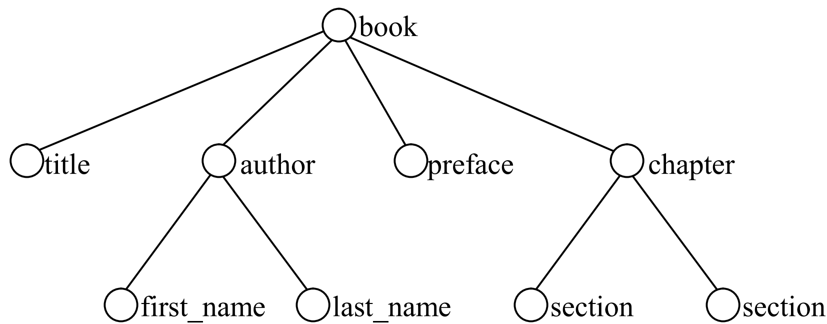


Figure 1.2: An ordered XML tree

The growing number of XML documents on the Web has motivated the development of languages and index techniques to query XML data efficiently. Several query languages, such as XML-QL [25], XML-GL [14], Quilt [15], XPath [8], XQuery [9], and XTree [19], have been proposed to query XML and semi-structured data. These query languages express the structure of XML documents as linear paths or twig patterns. For example, the XPath query:

/book[/title]//section[2]/preceding-sibling::section

finds all the *section* nodes that are siblings of *section[2]* (section[2] means the second section) and these *section* sibling nodes should be before *section[2]* ("*preceding*-sibling"). Meanwhile, *section[2]* should be a descendant of *book* ("//"). In addition, *book* should satisfy the restriction that it has a child *title* ("/").

No matter the query is a linear path or a twig pattern, the core operation for an XML query is to efficiently determine the *ancestor-descendant (A-D)*, *parent-child (P-C)*, *sibling* and *ordering* relationships.

To facilitate the determination of these relationships, two main index techniques are proposed, namely structural index and labeling (numbering) scheme.

The structural index approaches, such as Dataguides [31, 59, 60], 1-index [61], 2-index [61], A(k)-index [44], D(k)-index [65], M(k)-index [35], Index Fabric[24], F&B index[42], APEX [22] and Representative Objects [62], can help to traverse the hierarchy of XML, but this traversal is costly and the overhead of the traversal can be substantial if the path lengths are very long or unknown. As a result, such approaches can be fairly inefficient.

On the other hand, the labeling scheme approaches, such as containment scheme [3, 26, 56, 80, 83], prefix scheme [23, 41, 50, 64, 70] and prime number scheme [74], require smaller storage space, yet they can efficiently determine the ancestor-descendant (A-D) etc. relationships between any two elements based on the labels only. Both the ordered and un-ordered queries can be processed without accessing the original XML file. In addition, the labeling schemes can be used to query XML no matter XML is schema oblivious or schema-conscious. In this thesis, we focus on the labeling schemes.

1.1.4 XML Update

In this section, we discuss XML updates based on the structural index technique and the labeling scheme technique.

Structural index of XML data is not a schema predefined but only a structure summary from the original data. While the data could be changed gradually, the index should be updated accordingly to keep the consistence. [34, 43, 65, 79] are techniques

to update the structural index which iteratively split the nodes to make the index correct and merge all the nearby nodes to make the index size to be minimum without violation. The splitting and merging of nodes are costly, therefore the update of structural index is inefficient.

As for the labeling schemes, if XML is dynamic, how to efficiently update the labels of the labeling schemes is now becoming an important research topic. [13, 23, 28, 69, 70, 75] can process the updates (inserts or deletes nodes) efficiently if the order of XML elements is not taken into consideration. However as we know, the elements in XML are intrinsically ordered, which is referred to as the document order (the element sequence in XML), i.e. the preorder traversal of an XML tree. The relative order of two paragraphs in XML is important because the order may influence the semantics of XML, therefore the standard XML query languages (e.g., XPath[8] and XQuery [9]) require the output of queries to be in document order by default. In addition, XPath and XQuery include both ordered and un-ordered queries. The ordered query needs to determine the ordering relationship between two elements. Thus it is very important to maintain the document order when XML is updated; otherwise some semantics of XML will be lost and the ordered queries can not be answered. Hence it is very important to maintain the document order when XML is updated.

1.2 Problem Statement and Motivation

Though labeling schemes are more efficient than structural index in determining the four basic relationships in XML query, each labeling scheme is not efficient to

determine all the four basic relationships. For instance, the containment scheme is very *inefficient* to determine the *sibling* relationship; it needs to search the parent of a node, then decides whether another node is a child of this parent. The prefix scheme is very *inefficient* in determining all the four relationships if the XML tree is *deep*. The prime number scheme has large label size and it employs the modular and division operations to determine the relationships which is very expensive. Thus the *first objective* of this thesis is to propose a labeling scheme that can efficiently determine all the four basic relationships no matter what XML structure is.

It is important to efficiently update the labels of the labeling schemes when XML is updated, and it is especially important to maintain the document order in XML updating. Some research [6, 23, 50, 52, 64, 68, 70, 74] has been done to maintain the document order in XML updating. However the update costs of these approaches are still high. Therefore the *second and the most important objective* of this thesis is to dramatically reduce the order-sensitive update cost; while completely avoid re-labeling in XML updates.

Furthermore, none of the existing labeling schemes can process the *internal* node update efficiently. Therefore we also propose techniques to process the internal node update efficiently.

1.3 Overview of Contributions

To accomplish the above objectives, we propose techniques to improve the query efficiency as well as dramatically decrease the update cost. The main contributions of this thesis are summarized as follows:

- Firstly, we propose the P-Containment (P represents the "Parent_Start" value of a node) scheme. The P-Containment scheme can efficiently determine all the four basic relationships in XML queries, more important it can be used to efficiently process internal node updates and to completely avoid re-labeling.
- Secondly, the most important contribution of this thesis is that we propose novel encoding approaches for encoding node labels which can process XML updates much more efficiently. The most important feature of Compact Dynamic Binary String (CDBS) encoding and Compact Dynamic Quaternary String (CDQS) encoding is that we compare the CDBS and CDQS codes based on the *lexicographical* order. We can always find a binary (or quaternary) string between any two consecutive CDBS (or CDQS) codes with the *orders* kept and *without re-encoding or re-labeling the existing numbers or nodes*. Meanwhile, CDBS and CDQS encodings are very compact. In addition the CDBS (or CDQS) encoding is orthogonal to specific labeling schemes, thus it can be applied broadly to different labeling schemes.
- When P-Containment labeling scheme is combined together with our CDBS (or CDQS) encoding, both the queries and updates can be efficiently processed.
- We conduct comprehensive experiments to demonstrate the benefits of our approaches over the previous approaches in processing both queries and updates.

1.4 Organization of Thesis

To the end, we outline the organization of this thesis. The rest of this thesis is organized in 6 chapters.

Chapter 2 reviews the research work that is closely related to this thesis. Three main labeling schemes, i.e. containment, prefix and prime labeling schemes, are presented in this chapter. Also we introduce different encodings to store the labels. Meanwhile the deficiencies of these labeling schemes and encoding approaches are analyzed.

In Chapter 3, we propose the P-Containment (P represents the “Parent_Start” value of a node, and the “Parent_Start” value of a node is the “Start” value of its parent) scheme which makes the determination of sibling relationships much faster than the existing containment labeling scheme. Also P-Containment is faster than the existing containment scheme in determining the parent-child relationship. The P-Containment scheme is also helpful to process the *internal node* updates (see Section 4.4.2 of Chapter 4) and to *completely avoid re-labeling* (see Section 5.3 of Chapter 5).

Chapter 4 to Chapter 6 are all about how to efficiently process XML updates. They are the most important contributions of this thesis.

In Chapter 4, we illustrate that the most important feature of our approach is that we compare labels based on the ***lexicographical order***; an algorithm that can insert a binary string between two binary strings with the orders kept is also proposed in this chapter which is the ***first foundation*** of this thesis. In this chapter, we also propose ***Compact*** *Dynamic* Binary String (CDBS) encoding and indicate that CDBS encoding can be applied ***broadly*** *(the* ***second foundation*** *of this thesis)* to different

labeling schemes. Based on the CDBS encoding, we also discuss how to process the leaf node updates, internal node updates, subtree updates, and uniformly and skewed updates for XML in this chapter.

Chapter 5 thoroughly discusses that CDBS will encounter the *overflow* problem, therefore we further improve CDBS to CDQS. Though the label size of CDQS is larger than the label size of CDBS and the update cost of CDQS is a little higher, CDQS ***completely*** avoids re-labeling in order-sensitive updates.

In Chapter 6, we describe how to control the increase in label size. Two techniques are discussed. The first one is that we designed an algorithm which can find the label with the smallest size between two labels in the update environment with both insertions and deletions, thus the label size will increase slow; meanwhile the orders can be maintained. The second one is that we discuss how to process the skewed insertion problem to control the increase of label size.

Finally, Chapter 7 summarizes the contributions of this thesis and discusses the future works.

All the works in this thesis have been published in international conferences and journals. The work in Chapter 3 has been published in [51]. The work in Chapter 4 has been published in [48]. The work in Chapter 5 has been published in [50][1]. The work in Section 6.1 of Chapter 6 has been published in [49], and the work in Section 6.2 of Chapter 6 has been published in [52]. Also we summarize the update works in Chapters 4, 5 and 6 into [55] which has been accepted by VLDB Journal.

[1] Note that in [50] we use the "QED" to represent the quaternary encoding. In this thesis, in order to make the name consistent with the CDBS in [48], we change the title "QED" to "CDQS", but the contents of "QED" and "CDQS" are exactly the same.

Chapter 2

Background and Related Works

Some labeling (numbering) schemes have been proposed for network routing [30], object programming [4, 26, 27, 73], knowledge representation systems [1], and recently XML search engines [3, 20, 23, 24, 41, 56, 64, 70, 74, 80, 83]. [21] further applies the labeling schemes to search the semantic web (see [11, 33, 47, 53, 54] for more details about the semantic web).

In this thesis, we focus on XML queries based on labeling schemes. XML query can be expressed as linear paths [2, 29, 40, 82] or twig patterns [12, 17, 18, 57, 58, 66, 81]. The next-of-kin (NoK) pattern matching in [82] can speed up the node-selection step and reduce the join size significantly. Jiao et al. [40] evaluate the path queries with “not” predicates. Bruno et al. [9] propose a holistic approach which uses stacks to match twig patterns. Zhang et al. [81] propose the Blossom Tree to evaluate correlated paths in a FLWOR expression that can generate highly efficient query plans in different environments.

The difference between path query and twig pattern query is not an emphasis of this thesis. Instead, we focus on improving the efficiency of labeling schemes which can facilitate both the path query and twig pattern query because both the path query and twig pattern query are based on labeling schemes. Also we focused on

updates based on labeling schemes. After updating, the labeling schemes still can efficiently support both the path query and twig pattern query. Also different encoding approaches are proposed to store the labels of the labeling schemes.

The rest of this chapter is organized as follows. In Section 2.1, we introduce different labeling schemes to process XML queries. In Section 2.2, we introduce the encoding approaches which are used to encode the labels of labeling schemes in storing. We summarize this chapter in Section 2.3.

2.1 Node Labeling Schemes

The labeling scheme is used to label the nodes of an XML tree, and based on the labeling scheme, XML queries can be processed without accessing the original XML document.

In this section, we survey three families of labeling (numbering) schemes, viz. containment [3, 26, 45, 46, 56, 80, 83], prefix [23, 41, 50, 64, 70], and prime [74].

2.1.1 Containment Labeling Scheme

The containment labeling scheme was first suggested by Santoro and Khatib [67]. Yoshikawa and Amagasa [80] also proposed a variant of containment labeling scheme. To label an XML tree based on the containment scheme, different tree traversal methods (e.g. pre-and-postorder[26], extended preorder[56]) are used.

(1) Dietz's containment labeling scheme [26] uses tree traversal order to determine the ancestor-descendant relationship between any two nodes of an XML

tree. Figure 2.1 shows Dietz's containment scheme. Each node is labeled with a pair of preorder and postorder numbers. For any two nodes u and v of an XML tree, u is an ancestor of v if and only if u occurs before v in the preorder traversal of the XML tree and after v in the postorder traversal.

In the tree shown in Figure 2.1, node [1, 9] is an ancestor of node [4, 2], because node [1, 9] comes before node [4, 2] in the preorder (*i.e.*, 1 < 4) and after node [4, 2] in the postorder (*i.e.*, 9 > 2). An obvious benefit from this approach is that the ancestor-descendant relationship can be determined in constant time by examining the preorder and postorder numbers of tree nodes.

(2) Li et al. [56] uses an *extended preorder* and a *range of descendants*. Every node is assigned two variables: "order" and "size". These two variables represent an interval [order, order + size]. Figure 2.2 shows Li's labeling scheme. For any two nodes u and v, u is an ancestor of v iff order(u) < order(v) < order(u) + size(u).

In the tree shown in Figure 2.2, node [1, 150] is an ancestor of node [52, 10], because the order of node [1, 150] is 1 which is smaller than the order 52 of node [52, 10], and 52 is smaller than order([1, 150]) + size([1, 150]) = 1 + 150 = 151.

(3) Zhang et al. [83] use a labeling scheme in which every node is assigned three values: "start", "end" and "level". For any two nodes u and v, u is an ancestor of v iff u.start < v.start and v.end < u.end. Node u is a parent of node v iff u is an ancestor of v and v.level – u.level = 1. Node u is a sibling of node v iff the parent of node u is also a parent of node v. Node u is a preceding (following) node of node v iff u.start < (>) v.start. Example 2.1 is a concrete example to show how Zhang's containment scheme works on determining the four basic relationships.

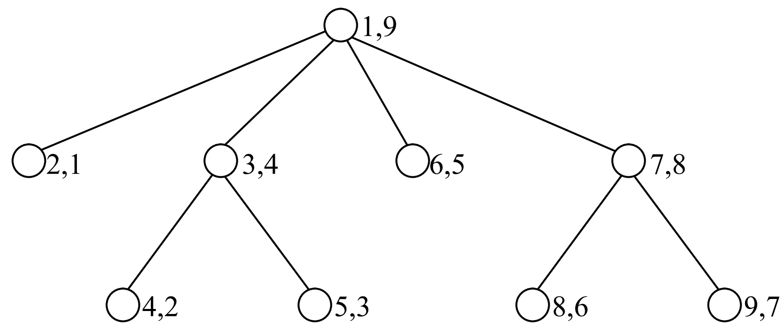


Figure 2.1: Dietz's containment scheme using preorder and postorder

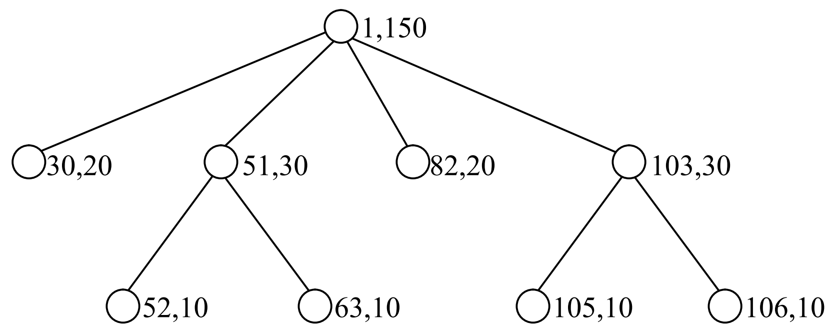


Figure 2.2: Li's containment scheme with order and interval size

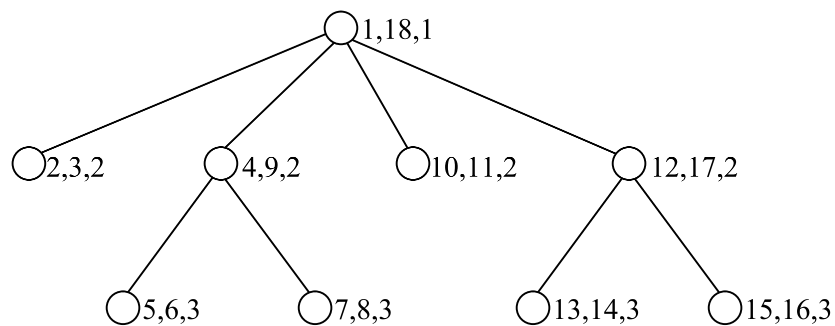


Figure 2.3: Zhang's containment scheme

Example 2.1 *Figure 2.3 shows Zhang's containment labeling scheme [83] based on the XML tree shown in Figure 1.2. The values near each node are the "start", "end" and "level" values.*

Ancestor-Descendant determination: "5,6,3" is a descendant of "1,18,1" because interval [5, 6] is contained in interval [1, 18].

Parent-Child determination: "5,6,3" is a child of "4,9,2" because interval [5, 6] is contained in interval [4, 9], and the level of "5,6,3" minus the level of "4,9,2" is 3 – 2 = 1.

Sibling determination: To determine whether "7,8,3" is a sibling of "5,6,3", the containment scheme needs to search the parent of "5,6,3" firstly, then decide whether "7,8,3" is a child of this parent. The search of the parent needs a lot of parent-child determinations which is very expensive.

Ordering determination: "7,8,3" is before (a preceding node of) "13,14,3" in document order because the "start" of "7,8,3" is smaller than the "start" of "13,14,3" i.e. 7 < 13.

[83] carries out a depth-first traversal of an XML tree (see Figure 2.3). It utilizes a counter which has an initialized value 1. The "start" of the interval for the root is 1, then from the root to leaves, the "start" of the interval for each node is the counter plus 1. When reaching a leaf node, the "end" of the interval is the current counter value plus 1. Based on the depth-first traversal, the "end" and "start" of the rest intervals can be determined.

The labeling schemes shown in Figure 2.1, Figure 2.2 and Figure 2.3 all have the same property to determine the ancestor-descendant etc. relationships, that is, if the interval of node v is *contained* in the interval of node u, node u is an ancestor of node v, therefore they are all called *containment* schemes. There are some other

containment labeling schemes, and they all have the same property to determine the ancestor-descendant etc. relationships. Here we do not show them further.

Dietz's containment scheme is the early work which has not discussed how to process the parent-child and sibling relationships yet. Li's containment scheme supports updates to some extent with the unused values; on the other hand, the unused values are a waste of numbers. Zhang's containment scheme can determine different relationships. In the later parts of this thesis, we mainly *focus on Zhang's containment scheme (Figure 2.3)* to represent the containment scheme if Dietz's and Li's containment schemes are not explicitly mentioned, and in fact our encoding approaches can be applied to all the other containment labeling schemes also.

2.1.1.1 Deficiencies of the Containment Schemes on Queries

In this section, we show what are the deficiencies of the containment schemes in determining the relationships in XML queries.

It can be seen from Example 2.1 that it is very *inefficient* for the containment scheme to determine the *sibling* relationship; it needs to search the parent of one node and determine whether another node is the child of this parent, which needs a lot of parent-child determinations and is very costly.

2.1.1.2 Deficiencies of the Containment Schemes on Updates

Although the ancestor-descendant relationship can be determined in constant time by the containment scheme, the insertion of a node will lead to a re-labeling of all the ancestor nodes of this inserted node and all the nodes after this inserted node in document order (see Figures 2.1 and 2.3; more details can be found in Example 4.12 of Chapter 4). This problem may be alleviated if the interval size is increased with

some values unused [56] (see Figure 2.2). However, large interval size wastes a lot of numbers which causes the increase of storage, while small interval size is easy to lead to re-labeling.

To solve the re-labeling problem, in [6] Float-point values are used for the "start" and "end" of the intervals. It seems that Float-point solves the re-labeling problem [70]. But in practice, the Float-point values are represented in a computer with a fixed number of bits [6, 70]. As a result, at most 18 nodes can be inserted at a fixed place [6] *since* [6] uses the consecutive integer values at the initial labeling. Even if [6] uses values with large gaps, it still can not avoid re-labeling due to the float-point precision. No one has ever proposed using variable length encoding of real values to maintain orders since it is not convenient for variable length codes to execute the addition, division etc. operations. Therefore, using real values instead of integers only provides limited benefits for the label updating [70, 74]. In fact, the Float-point [6] is equivalent to the approach that leaves some values unused [56].

It should be noted that the re-labeling in the containment scheme is not only for maintaining the document order. If the XML tree is not re-labeled after a node is inserted, the containment scheme can not work correctly to determine the ancestor-descendant, parent-child etc, relationships. Therefore it is very important to efficiently process the updates of labels in the containment labeling schemes.

2.1.2 Prefix Labeling Scheme

In the prefix labeling scheme, the label of a node is that its parent's label (prefix) concatenates its own (self) label. Label(u) represents the label of node u,

prefix_label(u) represents the prefix label of node u (the label of the parent of node u), and self_label(u) represents the self_label of node u. The following discussions show how the prefix labeling scheme determines the four basic relationships, i.e. ancestor-descendant, parent-child, sibling and ordering relationship, and Example 2.2 for the DeweyID prefix scheme [70] is a concrete example to show how the prefix schemes work on determining the four basic relationships. For any two nodes u and v, u is an ancestor of v iff label(u) is a substring of label(v), i.e. suppose the length of label(u) is L, then the first L number of symbols of label(v) are exactly the same as label(u). Node u is a parent of node v iff prefix_label(v) is equal to label(u). Node u is a sibling of node v if prefix_label(u) = prefix_label(v). Node u is a preceding (following) node of node v iff label(u) is smaller (larger) than label(v) when comparing label(u) and label(v) component by component from left to right (the component is separated by the delimiters; see Example 2.2 for what is a component).

We will discuss three prefix labeling schemes, i.e. DeweyID, BinaryString and OrdPath, and outline their weak points.

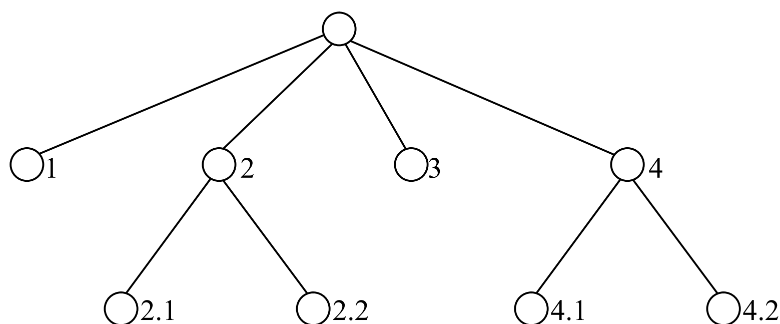


Figure 2.4: DeweyID prefix scheme

(1) DeweyID

DeweyID [70] labels the nth child of a node with an integer n, and this n should be concatenated to the prefix (its parent's label) and delimiter (e.g. ".") to form the complete label of this child node. It should be noted that the label of the root of the XML tree is an empty string (for all the prefix labeling schemes). Figure 2.4 shows DeweyID.

Example 2.2 *Based on DeweyID (see Figure 2.4), we show how the prefix schemes work on determining the four relationships in XML queries.*

Ancestor-Descendant determination: "2.1" is a descendant of the root because the empty string is a prefix substring of "2.1".

Parent-Child determination: "2.1" is a child of "2" because the prefix_label of "2.1" is "2" which is equal to label "2".

Sibling determination: "2.2" is a sibling of "2.1" because they have the same prefix_label "2".

Ordering determination: "2.1" is before "4.1" in document order because the "2" in "2.1" is smaller than the "4" in "4.1" i.e. we compare "2.1" and "4.1" from left to right to see the component in which labels is smaller.

(2) Binary String

Cohen et al. [23] use Binary Strings to label the nodes, called *BinaryString* in this thesis. Figure 2.5 shows the BinaryString prefix scheme. The root of the tree is labeled with an empty string. The first child of the root is labeled with "0", the second child with "10", the third with "110", and the fourth with "1110" etc. Similarly for

any node u, the first child of u is labeled with label(u).“0”, the second child of u is labeled with label(u).“10”, and the ith child with label(u).“$1^{i-1}0$”. The determinations of the four basic relationships based on the BinaryString prefix scheme is similar to the determinations based on DeweyID prefix scheme (see Example 2.2). The deficiency of BinaryString is that its label size is too large.

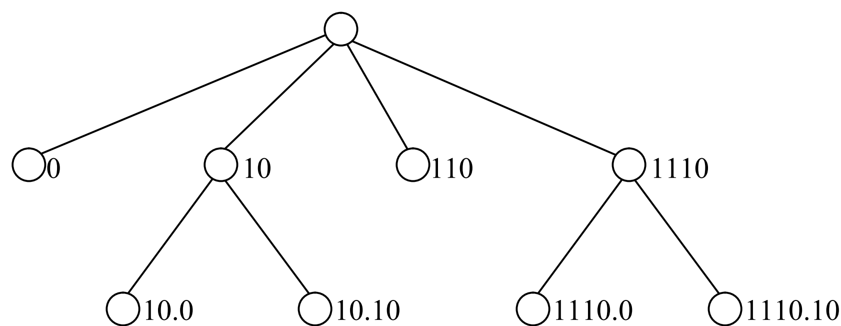


Figure 2.5: BinaryString prefix scheme

(3) OrdPath

OrdPath [64] is similar to DeweyID, but it only uses the odd numbers at the initial labeling (see Figure 2.6). When an XML tree is updated, it uses the even number between two odd numbers to concatenate another odd number (see Example 2.3 for details). OrdPath wastes half of the total numbers. The query performance of OrdPath is worse since it needs more time to decide the prefix levels based on the even and odd numbers. We use the following example to illustrate OrdPath.

Example 2.3 *Given three DeweyID labels “1”, “2” and “3”, we can easily know that they are siblings. In addition, given two DeweyID labels “2” and “2.1”, we can easily know that “2” is a parent of “2.1”. But for OrdPath (see Figure 2.6), its labels*

are "1", "3", "5" etc.; when inserting a label between "1" and "3", it uses the even number between "1" and "3" i.e. "2" to concatenate another odd number e.g. "1" ("1" has smaller size in OrdPath encodings; see Tables 2.2 and 2.3) as the label of this inserted node, i.e. the inserted label is "2.1". In OrdPath, "2.1" is at the same level as "1", 3" etc., i.e. "2.1" is a sibling of "1" and "3". Furthermore, when inserting one more node between "1" and "2.1", OrdPath uses "2.-1" as the inserted label. Moreover, when inserting one more node between "2.-1" and "2.1", the inserted label will be "2.0.1". The OrdPath labels "1", "2.-1", "2.0.1", "2.1" and "3" are all siblings, but from these labels, they look at different levels. OrdPath needs more time to determine the sibling, parent-child etc. relationships in XML query processing. Thus OrdPath gets better update performance by decreasing the query performance. That is not what we expected.

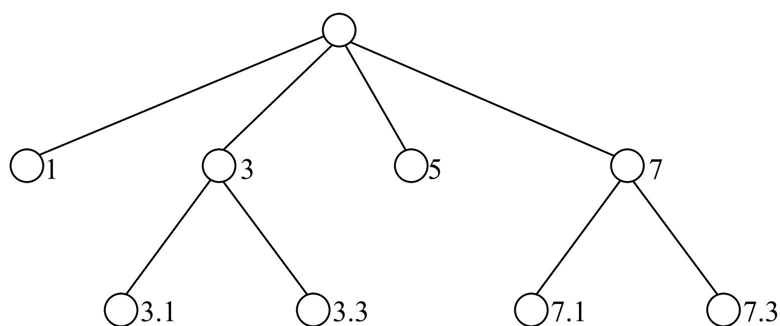


Figure 2.6: OrdPath prefix scheme

2.1.2.1 Deficiencies of the Prefix Schemes on Queries

In this section, we show the deficiency of the prefix scheme in XML queries.

From Example 2.2, we can see that the Prefix scheme can determine all the four basic relationships fast if the XML tree is shallow. However, it is very inefficient for the prefix scheme to determine all the four basic relationships if the XML tree is deep. For instance, to determine that "1.2.1.1.3.3.4.5" is a parent of "1.2.1.1.3.3.4.5.2", the prefix scheme needs to compare 8 pairs of numbers.

OrdPath also has the problem that the query performance will be decreased if the XML tree is deep. Besides this, OrdPath also has the following drawbacks in XML queries:

(1) It wastes half of the total numbers compared to DeweyID (wastes the even numbers; even after insertion, it still wastes the even number, e.g. "2.0" between "2.-1" and "2.1" will never be used after insertion), which will cause the storage increasing and accordingly the query performance decreasing.

(2) It can be seen from Example 2.3 that "1", "2.-1", "2.0.1", "2.1" and "3" are at the same level, i.e. they are siblings. OrdPath needs more time to determine this based on the even and odd numbers (the even number is not a level) which will decrease its query performance.

2.1.2.2 Deficiencies of the Prefix Schemes on Updates

Compared with the containment scheme, the prefix scheme (DeweyID and BinaryString) is dynamic to some extent. When a node is inserted into an XML tree, the prefix scheme can always put this node as the last sibling, then the existing nodes need not be re-labeled and we can determine the ancestor-descendant, parent-child and sibling relationships. However, the *ordering* relationship is not kept which may

break down the semantics of XML and make the order-sensitive queries unanswerable, i.e. some of the queries in XPath and XQuery can not be answered.

To keep the document order, the DeweyID and BinaryString prefix schemes need to re-label the sibling nodes after the inserted node and the descendants of these siblings (more details can be found in Example 4.11 of Chapter 4).

OrdPath can avoid re-labeling to some extent, but it greatly reduces the query performance (see Section 2.2.1) and its update cost is expensive.

(1) To some extent, OrdPath [64] can keep the document order without re-labeling the existing nodes. But because OrdPath stores the sizes of the labels to separate different labels, all the nodes should be re-labeled when the sizes of the labels overflow. We will further discuss the *overflow* problem in Example 5.1 of Chapter 5.

(2) OrdPath needs the addition and division operations to calculate the even number between two odd numbers which is expensive in updating. It is also possible that OrdPath only uses the addition operation to get the even number, but if there are many deletions, the calculation of the even number based only on the *addition operation* is bias and the label size will increase fast. Even if there is only the addition operation, the addition operation is also expensive.

2.1.3 Prime Labeling Scheme

Wu et al. [74] proposed an approach to label XML trees with prime numbers (we use *Prime* to refer to this scheme). Figure 2.7 shows Prime, in which the number above each node is the document order, the label is at the right side of each node, and the

two numbers below each label are its parent_label and self_label. The root node is labeled with "1" (integer). Then based on a top-down approach, each node is given a unique prime number (self_label) and the label of each node is the product of its parent node's label (parent_label) and its own self_label.

Example 2.4 *Prime uses a top-down approach to label the nodes (see Figure 2.7), i.e. label the root firstly, then all the child nodes of the root, then all the grandchild nodes, etc. The 0th node (the root node; 0th is the document order above the root node in Figure 2.7) is labeled with "1" (the right number). Then the 1st (the number above the node) node is labeled with "2" (the right number) which is the product of its parent_label "1" and its self_label, i.e. the prime number "2". The 2nd node is labeled with "3" which is the product of its parent_label "1" and the next available prime number (self_label) 3. Similarly the rest child nodes of the root are labeled with "5" and "7". Next Prime labels the grandchild nodes of the root. The 3rd (3rd is the document order above the node) node is labeled with "33" which is the product of its parent label "3" and the next available prime number (self_label) "11" (the prime number "7" has been used by the last child node of the root). Similarly the 4th , 7th and 8th nodes can be labeled.*

Although the document order of each node is explicitly shown in Figure 2.7, Prime does not store the document order. It uses the SC (Simultaneous Congruence) value in Chinese Remainder Theorem [7, 74] to decide the node order (see Appendix B for the calculation details of the SC value).

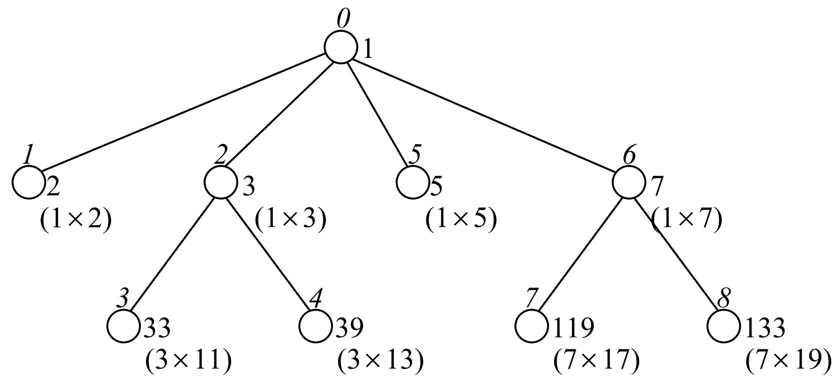


Figure 2.7: Prime scheme

Example 2.5 *The SC value for the 8 nodes (except the root) in Figure 2.7 is 8965025 (see Appendix B for the SC calculation steps). That is to say, 8965025 mod 2 = 1 (here 2 is the self_label and 1 is the document order), 8965025 mod 3 = 2, ···, 8965025 mod 17 = 7, and 8965025 mod 19 = 8. Prime only needs to store this SC value and the self_labels rather than store the document order.*

Next we show how the prime labeling scheme determine the four basic relationship in XML query processing. For any two nodes u and v, u is an ancestor of v iff label(v) mod label(u) = 0. Node u is a parent of node v iff label(v)/self_label(v) = label(u). Node u is a sibling of node v iff label(u)/self_label(u) = label(v)/self_label(v). Prime uses the SC (Simultaneous Congruence) values to decide the document order, i.e. SC mod self_label = document order, then it compares the document orders of two nodes. Example 2.6 is a concrete example to show how Prime determines the four basic relationships in XML queries.

Example 2.6 *See Figure 2.7 for the prime labeling scheme.*

Ancestor-Descendant determination: "33" is a descendant of the root because 33 mod 1 = 0.

Parent-Child determination: "33" (label(v)) is a child of "3" (label(u)) because label(v)/self_label(v) = 33/11 = 3 = label(u).

Sibling determination: "33" (label(v)) is a sibling of "39" (label(u)) because label(u)/self_label(u) = 39/13 = 3 = 33/11 = label(v)/self_label(v).

Ordering determination: label "39" is before (a preceding node of) label "119". Prime determine the order in this way. The SC value is 8965025, and the self_labels of "39" and "119" are "13" and "17" respectively. The document order of label "39" is SC mod self_label = 8965025 mod 13 = 4, the document order of label "119" is SC mod self_label = 8965025 mod 17 = 7. 4 is smaller than 7, therefore label "39" is before label "119" in document order.

Based on the SC value, Prime can solve the label update problem, which only needs to re-calculate the SC value [74].

Example 2.7 *When a new sibling node is inserted before the* 1st *node (the inserted node is now the first child of the root), the next available prime number is 23, then the label of the new inserted node is 23 (*1×23*). This new inserted node now becomes to the* 1st *node (document order), and the orders of the nodes after this inserted node should all be added with 1 (the old orders are calculated based on the old SC value). Prime calculates the new SC value for the new ordering, which is 28364406 such that* ***28364406 mod 23 = 1****, 28364406 mod 2 = 2, 28364406 mod 3 = 3, ···, 28364406 mod 17 = 8, and 28364406 mod 19 = 9.*

Theoretically the single SC value is very good which avoids the node re-labeling by only re-calculating the SC value. However in practice, the number of nodes in an XML tree can not be so small, thus the single SC value will be too large a number. Therefore Prime [74] calculates the SC values for every five (or other number) nodes.

Example 2.8 *The SC value for the first five (in document order) nodes in Figure 2.7 is 3215 (3215 mod 2 = 1, ···, 3215 mod 5 = 5) and the SC value for the next three nodes is 160 (160 mod 7 = 6, 160 mod 17 = 7, and 160 mod 19 = 8). When inserting the new node, the SC value for the first five nodes is 6648 (6648 mod 23 = 1, ···, 6648 mod 13 = 5) and the SC value for the rest four nodes is 161 (161 mod 5 = 6, ···, 161 mod 19 = 9).*

2.1.3.1 Deficiencies of the Prime Scheme on Queries

In this section, we show the deficiencies of the prime number labeling scheme in processing queries.

The prime scheme skips a lot of integers to get the prime number, and the label of a child is the product of the next available prime number and its parent's label, which both make the storage space for Prime labels very large. The large storage space requires more I/O time in XML query processing.

Besides the query performance decreasing caused by the large storage space, Prime employs the modular and division operations to determine the ancestor-descendant, parent-child, sibling and ordering relationships which are very expensive.

Therefore the query performance of the prime labeling scheme is very bad (see the experimental results in Section 4.5.2.2 of Chapter 4).

2.1.3.2 Deficiencies of the Prime Scheme on Updates

Although Prime is the only scheme which supports order-sensitive updates without any re-labeling of the existing nodes, it needs to re-calculate the SC values based on the new ordering of nodes. The SC values are very large numbers and the re-calculation is much more time consuming than re-labeling.

2.2 Encoding Approaches to Store the Labels of Labeling Schemes

The labels in the labeling schemes should be stored as binary numbers or other encodings in a computer. In this section, we discuss different encodings for the labels to solve different problems in labeling schemes.

2.2.1 Binary Number Encodings

The labels of the containment schemes are integers and float-point values. In a computer, these values are stored as binary numbers, e.g. decimal number 5 will be stored as binary number 101. Also the labels in the prime number labeling scheme are stored as binary numbers in a computer. We will further compare the binary number encoding and with our dynamic binary string encoding in Chapter 4. Because the binary number encoding is trivial, here we do not discuss further about the binary number encodings.

2.2.2 UTF8 Encoding

The UTF8 [78] encoding is used by the DeweyID prefix scheme to process the delimiters. As we know, DeweyID uses delimiter "." to separate different components of a label, e.g. separate "2" and "1" in "2.1". However, in practice, the delimiter "." can not be stored together with the numbers, therefore DeweyID uses UTF8 [78] encoding to process the delimiters.

In UTF8, a variable number of bytes are used to encode different integer values. If the integer value is smaller than $128=2^7$, it is encoded with one byte 0xxxxxxx where x represents the bits used for the integer value. If the integer value is between 2^7 and 2^{11}, it is encoded with 2 bytes 110xxxxx 10xxxxxx. See Table 2.1 for more details. To represent an entire Dewey path with UTF8, each component of the path is encoded in UTF8 and then concatenated together without the delimiter ".". The indicator bits "0", "110", "1110", etc in the first byte (see Table 2.1) determine how many bytes are used and separate different components.

Table 2.1: UTF8 encoding

Value	Physical representation of self_label	Number of bytes
$0 \leq F<128$ (2^7)	0xxxxxxx	1
$2^7 \leq F<2^{11}$	110xxxxx 10xxxxxx	2
$2^{11} \leq F<2^{16}$	1110xxxx 10xxxxxx 10xxxxxx	3
$2^{16} \leq F<2^{21}$	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	4
$2^{21} \leq F<2^{26}$	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx	5
$2^{26} \leq F<2^{31}$	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx	6

Example 2.9 *Consider a DeweyID label “1.129”. Since “1” is less than 128, the UTF8 code of “1” will be “0****0000001****”. Since 129 is larger than* 2^7 *and less than* 2^{11}*, the 11 bit binary encoding of 129 is “10000000001”; then the first five bits “10000” will be concatenated after “110”, and the rest six bits “000001” will be concatenated after “10” (see the third row (*$2^7 \leq F < 2^{11}$ *row) of Table 2.1). Thus the UTF8 code of 129 is “110****10000**** 10****000001****”. Finally, the DeweyID “1.129” will be “0****0000001****110****10000****10****000001****” in UTF8. Based on the indicators “0” and “110”, we know that the first component is stored with 1 byte, and the second component is stored with 2 bytes. In this way, DeweyID can separate different components without using the delimiter “.”.*

After processing the delimiters of DeweyID, we call it DeweyID(UTF8).

2.2.3 OrdPath Encodings

OrdPath [64] is a prefix labeling scheme which can be used to process updates. In addition, O’Neil et al. [64] also proposed two encoding approaches, called OrdPath1 and OrdPath2, which are improvements of the UTF8 [78] encoding. The OrdPath encodings are also used to process the delimiters in the prefix labeling schemes, and they are more compact encodings than UTF8 [78].

Tables 2.2 and 2.3 show the two kinds of encodings of OrdPath, OrdPath1 and OrdPath2 (OrdPath2 is more compact). Both OrdPath1 and OrdPath2 codes have variable lengths. We use an example to show how OrdPath1 (Table 2.2) works. It is similar for OrdPath2 (Table 2.3).

Table 2.2: OrdPath1 encoding

Indicator	Number of bits	Interval
0000001	48	[-2.8x10^{14}, -4.3x10^{9}]
0000010	32	[-4.3x10^{14}, -69977]
0000011	16	[-69976, -4441]
000010	12	[-4440, -345]
000011	8	[-344, -89]
00010	6	[-88, -25]
00011	4	[-24, -9]
001	3	[-8, -1]
01	3	[0, 7]
100	4	[8, 23]
101	6	[24, 87]
1100	8	[88, 343]
1101	12	[344, 4439]
11100	16	[4440, 69975]
11101	32	[69976, 4.3x10^{9}]
11110	48	[4.3x10^{9}, 2.8x10^{14}]

Table 2.3: OrdPath2 encoding

Indicator	Number of bits	Interval
000000001	20	[-1118485, -69910]
00000001	16	[-69909, -4374]
0000001	12	[-4373, -278]
000001	8	[-277, -22]
00001	4	[-21, -6]
0001	2	[-5, -2]
001	1	[-1, 0]
01	0	[1, 1]
10	1	[2, 3]
110	2	[4, 7]
1110	4	[8, 23]
11110	8	[24, 279]
111110	12	[280, 4375]
1111110	16	[4376, 69911]
11111110	20	[69912, 1118487]

Example 2.10 *Suppose that there is a label "1.19" for the OrdPath prefix labeling scheme. "1" falls in "[0,7]" (see the third column of Table 2.2), thus "1" should be stored with 3 bits (see the second column of Table 2.2) i.e. "001", and the indicator*

"01" (used to indicate that the code is stored with 3 bits; see ***the first column of Table 2.2****) should be concatenated before "001", i.e. the OrdPath1 code of "1" is "01****001****" ("****01****" is the* ***indicator****; "****001****" is the value for* ***number 1*** *which is represented with* ***3 bits****; see* ***"01 3 [0,7]" line of Table 2.2****). "19" falls in "[8,23]" (see the third column of Table 2.2), thus "19" should be stored with 4 bits (see the second column of Table 2.2). The four bits to store "19" should be "1011" corresponding to the number* ***11****=19-8 (8 is the start of interval [8,23]). Note that the binary representation of "19" is "10011" which is 5 bits but not 4 bits. The complete OrdPath1 code for "19" is "100****1011****". If OrdPath wants to get back number "19", it needs to decode "100****1011****" to number "11" firstly then add the start "8" of interval [8,23]. With OrdPath encodings, the delimiters also need not be stored which is like UTF8. Though OrdPath is a more compact encoding than UTF8, its decoding time is larger.*

Though OrdPath1 and OrdPath2 encodings (see Tables 2.2 and 2.3) can decrease the label size compared to UTF8 encoding, it is slow for OrdPath1 and OrdPath2 to get back the numbers, e.g. to get back number 19, OrdPath1 should interpret the OrdPath1 code to 11 firstly, then add 8. This will influence both the query and update performance of the OrdPath prefix labeling scheme.

2.2.4 Binary String and Quaternary String Encodings

Cohen et al. [23] propose the BinaryString prefix labeling scheme. The binary string is also an encoding approach. There are only two symbols "0" and "1" in the binary

string and each symbol is stored with 1 bit. The size of the binary string encoding in [23] is very large.

In this thesis, we also use the binary string encoding. Compared with the binary string encoding in [23], our binary string encoding is *dynamic* and *compact*, called Compact Dynamic Binary String, i.e. CDBS.

In addition, we propose the quaternary string encoding which is a new encoding approach. There are four symbols in the quaternary string encoding, i.e. "0", "1", "2" and "3", and each symbol is stored with two bits, i.e. "00", "01", "10" and "11". Our quaternary encoding is also dynamic and compact, called Compact Dynamic Quaternary String, i.e. CDQS.

2.3 Summary

Towards the query performance, the existing containment labeling schemes can determine the ancestor-descendant, parent-child and ordering relationships very fast, but it is very inefficient in determining the sibling relationship. The prefix labeling schemes can determine all the four basic relationships in XML queries fast if the XML tree is shallow. However, if the XML tree is deep, the query performance based on the prefix labeling schemes will be greatly decreased. The query performance of Prime is very bad. Therefore the first objective of this thesis is to overcome the deficiencies of the existing labeling schemes such that query efficiencies can be improved. We propose the *P-Containment scheme* in Chapter 3 which can determine all the four basic relationships efficiently no matter what XML structure is.

Towards the update performance, although Prime supports order-sensitive updates without any re-labeling of the existing nodes, it needs to re-calculate the SC values based on the new ordering of nodes. The re-calculation is very time consuming.

The main idea of other labeling schemes [6, 56] (except Prime) is to leave some unused values for the future insertions. When the unused values are used up later, they have to re-label the existing nodes, i.e. they can not completely avoid re-labeling in XML updates.

The DeweyID and BinaryString prefix schemes can not support the order-sensitive updates.

Though OrdPath [13] is dynamic to some extent to process the order-sensitive updates (will encounter the overflow problem; see Example 5.1), it needs to decode its codes and use the addition and division operations to calculate the even number between two odd numbers, which both make its update cost not so cheap.

In addition, the better update performance of OrdPath does not come without a cost. It wastes a lot of even numbers which makes its label size larger, and it needs more time to determine the prefix levels based on the even and odd numbers in XML query processing.

In this thesis, we propose a novel *Compact Dynamic* Binary String (CDBS) encoding (CDBS is completely different from the encoding in [23]; the only common point is that they both use binary strings). The size of CDBS is as small as the binary number encoding of ***consecutive*** decimal numbers. As we know, there is no unused values between two ***consecutive*** decimal numbers; that means CDBS is the most compact and it need not leave unused values for the future insertions, thus the query

performance will not be decreased. Yet CDBS supports that codes can be inserted between any two ***consecutive*** CDBS codes because the most important feature of CDBS encoding is that we compare codes based on the ***lexicographical order***. This is the most important benefit of CDBS over the previous approaches. In addition, CDBS encoding can be applied ***broadly*** to different labeling schemes to process updates. Also CDBS does not decrease the query performance. Moreover, to solve the overflow problem of CDBS, i.e. the fixed size length field will overflow (see Example 5.1 for the details about the ***overflow*** problem), we improve CDBS to a *Compact Dynamic Quaternary* String (CDQS) encoding which can ***completely*** avoid re-labeling in XML updates.

The comparisons between our approaches and the existing approaches on queries and updates are summarized in Tables 2.4 and 2.5 respectively.

When P-Containment scheme and CDBS or CDQS encoding are combined together, both queries and updates can be processed efficiently.

Table 2.4: Comparisons on queries

Relationships	**Ancestor-Descendant**	**Parent-Child**	**Sibling**	**Ordering**
Containment Scheme	Efficient	Not very Efficient	Very inefficient	Efficient
Prefix Scheme	Not efficient if XML tree is deep	Not efficient if XML tree is deep	Not efficient if XML tree is deep	Not efficient if XML tree is deep
Prime Scheme	Very inefficient	Very inefficient	Very inefficient	Very inefficient
P-Containment Scheme	Efficient	Efficient	Efficient	Efficient

Table 2.5: Comparisons on updates

Schemes	Method to process updates	Descriptions
Dietz's [26] and Zhang's [83] containment	No	Need to re-label all the ancestor nodes and all the nodes after the inserted node in document order
Li's [56] containment	Leave unused values	Need re-labeling when the unused values are used up
Float-point [6] containment	Use float point value	Can not avoid re-labeling due to the float-point precision
DeweyID [70] and BinaryString [23] prefix	No	Need to re-label the sibling nodes after the inserted node and all the descendants of the following siblings
OrdPath [64] prefix	Odd and even numbers	Decrease the query efficiencies; update cost is high; can not completely avoid re-labeling due to the overflow problem
Prime [74]	SC values	Need not re-label, but need to re-calculate the SC values which is very expensive; greatly decrease the query performance
CDBS encoding	Dynamic binary string	Most compact, cheapest update cost; query performance is very good; can not completely avoid re-labeling due to the overflow problem.
CDQS encoding	Dynamic quaternary string	Not as compact as CDBS, update cost not as cheap as CDBS, but can completely avoid re-labeling

Chapter 3

P-Containment Scheme

This chapter introduces P-Containment which can improve query efficiency.

From Chapter 2, we know that the structure of XML will influence the query efficiency of the prefix labeling scheme. However, the structure of XML will not influence the query efficiency of the containment scheme. The comparison of two containment labels is only related to the total number of nodes in an XML tree. Also we know that the prime number scheme is very inefficient to determine all four relationships. Therefore in this chapter, we propose the P-Containment scheme which is based on containment, hence it will not be influenced by the structure of XML, also P-Containment can remove the drawbacks of the containment scheme, i.e. P-Containment can determine the sibling relationship and the other three relationships very efficiently.

The rest of this chapter is organized as follows. In Section 3.1, we propose the P-Containment scheme which can determine the *sibling* relationship much faster than the existing containment labeling schemes [26, 56, 83], can determine the *parent-child* relationship faster, and can determine the ancestor-descendant and ordering relationships as efficient as the existing containment schemes. We summarize this chapter in Section 3.2.

3.1 A Node Labeling Scheme: P-Containment Scheme

We firstly illustrate what is P-Containment scheme.

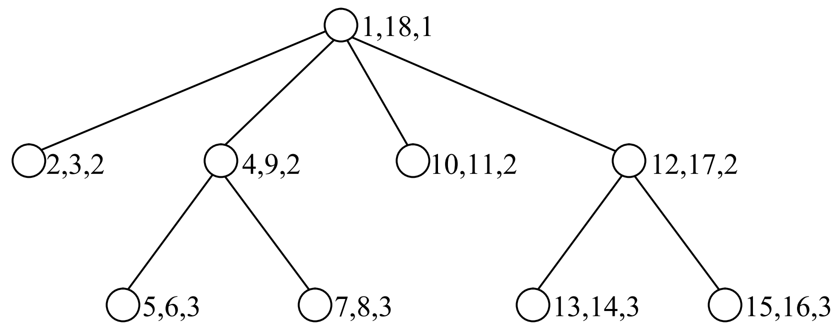
Rather than storing the "level" value in the existing containment scheme [83], P-Containment scheme stores the "parent_start" value, which is the "start" value of the parent of this node.

Example 3.1 *Figure 3.1(a) shows the existing containment scheme [83]; it can be seen that the existing containment scheme stores the "level" value. Figure 3.1(b) shows P-Containment scheme. Different from the existing containment scheme shown in Figure 3.1(a), P-Containment stores the "parent_start" value rather than the "level" value. In Figure 3.1(b), the "4" in "5,6,4" is the "parent-start" value, and it is equal to the "start" value of its parent, i.e. the "4" in "4,9,1".*

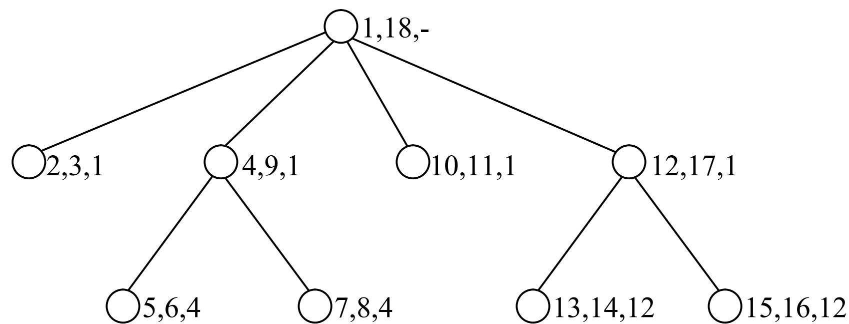
With the "parent_start", we can determine the *parent-child* and ***sibling*** relationships *faster*.

Property 3.1 *For two different nodes u and v, node u is a parent of node v iff the "**parent_start**" value of node v is equal to the "**start**" value of node u based on P-Containment.*

Property 3.2 *For two different nodes u and v which are not the root of an XML tree, node u is a sibling of node v iff the "**parent_start**" value of node u is equal to the "**parent_start**" value of node v based on P-Containment.*



(a) Zhang's containment scheme



(b) P-Containment scheme

Figure 3.1: The existing containment scheme and P-Containment scheme

Example 3.2 (Determine *parent-child* and *sibling* relationships based on *P-Containment* scheme) *Based on the P-Containment scheme shown in Figure 3.1(b), "4,9,1" is the parent of "5,6,4" because the "parent_start" value of "5,6,4" is 4 which is equal to the "start" value of "4,9,1". "5,6,4" is a sibling of "7,8,4" because their "parent_start" values are both equal to "4".*

Example 3.3 (*Comparison* between the *existing containment* scheme [83] and *P-*Containment scheme in determining the *parent-child* relationship) *Based on the*

existing containment scheme shown in Figure 3.1(a), "4,9,2" is the parent of "5,6,3" because 4 < 5, 6 < 9, and 3 – 2 = 1, i.e. the "start,end" interval of "5,6,3" should be contained in the "start,end" interval of "4,9,2", and the "level" value of "5,6,3" minus the "level" value of "4,9,2" should be equal to 1. It can be seen that the existing containment scheme needs 3 comparisons to determine the parent-child relationship. In contrast, P-Containment scheme only needs 1 comparison, i.e. the "parent_start" of one node is equal to the "start" of another node, thus P-Containment scheme is more efficient to determine the parent-child relationship.

Example 3.4 (*Comparison* between the *existing containment* scheme [83] and *P-*Containment scheme in determining the *sibling* relationship) *To determine the sibling relationship between "5,6,3" and "7,8,3" in Figure 3.1(a), the existing containment scheme needs to search the parent of "5,6,3", then decide whether "7,8,3" is the child of this parent. A lot of parent-child relationships should be determined in the searching of the parent of "5,6,3" which is very expensive. In contrast, P-Containment scheme only needs 1 comparison, i.e. the "parent_start" of one node is equal to the "parent_start" of another node, which is much cheaper.*

Therefore P-Containment scheme is more efficient to determine the parent-child and sibling relationships.

The following property shows that P-Containment scheme is still as efficient as the existing containment scheme to determine the ancestor-descendant and ordering relationships.

Property 3.3 *P-Containment scheme determines the ancestor-descendant and ordering relationships in the same way as the existing containment labeling scheme.*

Example 3.5 (Determine *ancetor-descendant* and *ordering* relationship based on *P-Containment* scheme) *Based on the P-Containment scheme shown in Figure 3.1(b), "5,6,4" is a descendant of "1,18,-" because 1 < 5 and 6 < 18. "7,8,4" is before "13,14,12" in document order because 7 < 13. The determinations of these two relationships based on P-Containment are in the same way as the existing containment scheme.*

Theorem 3.1 *P-Containment scheme requires that the "start" value of each node should be unique.*

Proof: If the "start" of P-Containment is not unique, P-Containment may determine the parent-child etc. relationships wrongly since more than one nodes have the same "start" and the "parent_start" of one node may be equal to the "start"s of many nodes. Therefore the "start" value should be unique.

To implement the P-Containment scheme, we only need to scan an XML tree once, then we can get the "start", "end" and "parent_start" values for all the nodes.

3.2 Summary

In this chapter, we propose the P-Containment scheme which can determine the sibling relationship much faster than the existing containment scheme, determine the

parent-child relationship faster, and determine the ancestor-descendant and ordering relationships as efficient as the existing containment scheme.

The P-Containment scheme is originally proposed by us to efficiently process the internal node updates (see Chapter 4); meanwhile we find that P-Containment can determine the sibling relationship much faster; hence we simply present the P-Containment scheme in this chapter. Now we find that in [32], the idea about storing the parent value is mentioned though [32] does not explicitly indicate that in this way, the sibling relationship can be processed much faster. [32] mainly focuses on processing the ancestor, descendant, preceding and following relationships in a coordinate plane by traversing an XML tree in preorder and postorder.

The novel and important contribution of this thesis is on processing of updates; see later chapters. The P-Containment scheme proposed here can be used to efficiently process internal node updates (see Chapter 4) and to completely avoid re-labeling (see Chapter 5). No one has ever studied that the "parent_start" value can be used to efficiently process internal updates and completely avoid re-labeling.

Chapter 4

CDBS Encoding of Node Labels to Efficiently Process XML Updates

To efficiently process XML updates, this chapter introduces a Compact Dynamic Binary String encoding, called CDBS. The features of this encoding are that (1) it supports order-sensitive insertions without re-encoding the existing binary string codes (*dynamic*), (2) it is as compact as the binary number encoding of consecutive numbers (*most compact*), and (3) it is orthogonal to specific labeling schemes, therefore it can be applied *broadly* to different labeling schemes to efficiently support XML updates.

The rest of this chapter is organized as follows. Section 4.1 indicates that the most important feature of our approach is that we compare codes (labels) based on the lexicographical order, also an algorithm is given in this section which can insert a binary string between two lexicographically ordered binary strings. Section 4.2 presents CDBS encoding which is very compact, yet it supports order-sensitive insertions efficiently. Section 4.3 discusses that CDBS encoding can be applied broadly to different labeling schemes. In Section 4.4, we discuss how to process XML updates based on CDBS encoding. Section 4.5 reports the experimental results. Finally, we summarize this chapter in Section 4.6.

4.1 Lexicographical Order for Binary Strings

The most important feature of our approach is that we compare labels based on the ***lexicographical order*** rather than the numerical order. In this section, we firstly introduce the definition of lexicographical order for binary strings (each symbol of the binary string is stored with 1 bit) and then propose an algorithm that can always insert a binary string between two lexicographically ordered binary strings. This algorithm is the *foundation* of this thesis which guarantees that we can update XML without re-labeling the existing nodes.

Definition 4.1 (Lexicographical order $\prec$) *Given two binary strings S_L and S_R (S_L represents the left binary string and S_R represents the right binary string), S_L is said to be lexicographically equal to S_R iff they are exactly the same. S_L is said to be lexicographically smaller than S_R ($S_L \prec S_R$) iff*

(a) the lexicographical comparison of S_L and S_R is bit by bit from left to right. If the current bit of S_L is 0 and the current bit of S_R is 1, then $S_L \prec S_R$ and stop the comparison, or

(b) S_L is a prefix of S_R.

Example 4.1 *Given two binary strings "0011" and "01", "0011" $\prec$ "01" lexicographically because the comparison is from left to right, and the 2nd bit of "0011" is "0", while the 2nd bit of "01" is "1". Given two binary strings "01" and "0101", "01" $\prec$ "0101" lexicographically because "01" is a prefix of "0101".*

Algorithm 4.1: AssignMiddleBinaryString(S_L, S_R)

Input: $S_L \prec S_R$; S_L and S_R are both ***ended with "1"***

Output S_M (ended with 1) such that $S_L \prec S_M \prec S_R$ lexicographically

Description:

1: **if** size(S_L) ≥ size(S_R) **then** //***Case (a)***

2: $S_M = S_L \oplus$ "1" //⊕ means concatenation

3: **else if** size(S_L) < size(S_R) **then** //***Case (b)***

4: $S_M = S_R$ with the last bit "1" changed to "01"

5: **end if**

6: **return** S_M

Next based on Algorithm 4.1, Theorem 4.1 and Example 4.2, we illustrate how to insert a binary string S_M (S_M represents the middle binary string) between two lexicographically ordered binary strings S_L and S_R (S_L represents the left binary string and S_R represents the right binary string) such that $S_L \prec S_M \prec S_R$ lexicographically.

Theorem 4.1 *Given any two binary strings S_L and S_R both of which end with "1" and $S_L \prec S_R$, we can always find a binary string S_M based on Algorithm 4.1 such that $S_L \prec S_M \prec S_R$ lexicographically.*

Proof:

Case (a): If size(S_L) ≥ size(S_R), we process S_M based on lines 1 and 2 in Algorithm 4.1, i.e. $S_M = S_L \oplus$ "1".

(a1): S_M is that S_L concatenates one more "1", thus S_L is a prefix of S_M. According to condition (b) in Definition 4.1, $\boldsymbol{S_L \prec S_M}$ lexicographically.

(a2): Since size(S_L) ≥ size(S_R) and $S_L \prec S_R$, condition (a) in Definition 4.1 must be satisfied. That means there is a position; the bit of S_L at this position is "0", and the bit of S_R at this position is "1". Therefore when we concatenate one more "1"

after S_L i.e. S_M, S_M is still smaller than S_R lexicographically (the lexicographical comparison is from left to right), i.e. $\boldsymbol{S_M \prec S_R}$.

Based on (a1) and (a2), $\boldsymbol{S_L \prec S_M \prec S_R}$ lexicographically when **size(*S_L*)** $\geq$ **size(*S_R*)**.

Case (b): If size(S_L) < size(S_R), we process S_M based on lines 3 and 4 in Algorithm 4.1, i.e. $S_M = S_R$ with the last bit "1" changed to "01".

(b1): If the first (size(S_R)-1) bits of S_R are larger than S_L lexicographically, $S_L \prec S_M$ because S_M is the first (size(S_R)-1) bits of $S_R \oplus$ "01". If the first (size(S_R)-1) bits of S_R are exactly the same as the S_L, $S_L \prec S_M$ because S_M is $S_L \oplus$ "01" (S_L is the same as the first (size(S_R)-1) bits of S_R; S_L is a prefix of S_M). Note that the first (size(S_R)-1) bits of S_R can not be smaller than S_L lexicographically, otherwise S_L will be larger than S_R lexicographically (conflict to the condition in Theorem 4.1). Therefore $\boldsymbol{S_L \prec S_M}$.

(b2): If we do not consider the last two bits "01" of S_M and the last bit "1" of S_R, S_M is exactly the same as S_R, and "01" $\prec$ "1" lexicographically. Therefore $\boldsymbol{S_M \prec S_R}$.

Based on (b1) and (b2), $\boldsymbol{S_L \prec S_M \prec S_R}$ lexicographically when **size(*S_L*) < size(*S_R*)**.

Therefore Theorem 4.1 holds.

Example 4.2 *To insert a binary string between "0011" and "01", the size of "0011" is 4 bits which is larger than the size 2 bits of "01", therefore we directly concatenate one more "1" after "0011" (see lines 1 and 2 in Algorithm 4.1). The inserted binary string is "0011**1**", and "0011" $\prec$ "00111" $\prec$ "01" lexicographically. To insert a binary string between "01" and "0101", the size of "01" is 2 bits which is smaller*

*than the size 4 bits of "0101", therefore we change the last bit "1" of "010**1**" to "**01**", i.e. the inserted binary string is "010**01**" (see lines 3 and 4 in Algorithm 4.1); obviously "01" $\prec$ "01001" $\prec$ "0101" lexicographically.*

Next we use an example to show why we require the last bit of the binary string to be "1".

Example 4.3 *Suppose there are two binary strings "0" and "00". "0" $\prec$ "00" lexicographically because "0" is a prefix of "00" (see Definition 4.1), but we can not insert a binary string S_M between "0" and "00" such that "0" $\prec$ S_M $\prec$ "00". Accordingly we require the binary strings to end with "1".*

Algorithm 4.1 is the ***foundation*** of this thesis which can help to process updates efficiently.

When the labeling scheme is a *prefix* scheme, based on *Theorem 4.1*, we can insert one label between two labels without re-labeling the existing nodes. When the labeling scheme is a *containment* scheme, we may need to insert the "start" and "end" two values at one place. The following *Corollary 3.3* guarantees that two labels can be inserted between two labels without re-labeling.

Lemma 4.2 *The S_M in Theorem 4.1 returned by Algorithm 4.1 ends with "1".*

Proof: This is obvious when we check Algorithm 4.1. Lines 1 and 2 indicate that the end bit of S_M is "1" when size(S_L) $\geq$ size(S_R), and lines 3 and 4 indicate that the end bit of S_M is "1" when size(S_L) < size(S_R). The case at lines 1 and 2 and the case at lines 3 and 4 are complete, therefore S_M ends with "1".

Corollary 4.3 *Given any two binary strings S_L and S_R which are both ended with "1" and $S_L \prec S_R$, we can always find two binary strings S_{M1} and S_{M2} such that $S_L \prec S_{M1} \prec S_{M2} \prec S_R$ lexicographically.*

Proof: Based on Theorem 4.1, we can insert a binary string S_M between S_L and S_R. Based on Lemma 4.2, we know that S_M is also ended with "1". Therefore based on Theorem 4.1, we can insert another binary string between S_L and S_M, or between S_M and S_R. Therefore Corollary 4.3 holds.

We can further insert binary strings among S_L, S_{M1}, S_{M2} and S_R.

Theorem 4.1 and Corollary 4.3 guarantee that we have low update cost in XML updating.

Algorithm 4.1 proposed in this thesis is *dynamic* and can be applied to any two ordered binary strings (ended with "1") for insertions. On the other hand, to maintain the high query performance, we should not increase the label size when reducing the update cost. In Section 4.2 we further propose a ***Compact*** Dynamic Binary String encoding, called CDBS. All the codes (binary strings) of CDBS are ended with "1" and CDBS encoding is as compact as the existing binary number encoding of consecutive numbers (see Section 4.2).

4.2 The Compact Dynamic Binary String Encoding (CDBS)

In this section, we propose a ***Compact*** Dynamic Binary String encoding (CDBS), and based on Algorithm 4.1, CDBS supports *updates* efficiently.

Table 4.1: Binary and CDBS encodings

Decimal number	V-Binary	V-CDBS	F-Binary	F-CDBS
1	1	00001	00001	00001
2	10	0001	00010	00010
3	11	001	00011	00100
4	100	0011	00100	00110
5	101	01	00101	01000
6	110	01001	00110	01001
7	111	0101	00111	01010
8	1000	011	01000	01100
9	1001	0111	01001	01110
10	1010	1	01010	10000
11	1011	10001	01011	10001
12	1100	1001	01100	10010
13	1101	101	01101	10100
14	1110	1011	01110	10110
15	1111	11	01111	11000
16	10000	1101	10000	11010
17	10001	111	10001	11100
18	10010	1111	10010	11110
Total size (bits)	64	64	90	90

We firstly use an example to illustrate how CDBS encodes a set of numbers, and use examples to simply analyze the total size of the CDBS codes. Next the formal encoding algorithm in Section 4.2.1 and the formal size analysis in Section 4.2.2 will be easier to understand.

Table 4.1 shows the binary number encoding (V-Binary and F-Binary) and CDBS (V-CDBS and F-CDBS) encoding of 18 numbers. We choose 18 as an example because the total "start" and "end" values in Figure 2.3 are 18. In fact, CDBS can encode any number (not only 18; see the formal algorithm in Section 4.2.1).

When encoding 18 decimal numbers in binary, they are shown in Column 2 (V-Binary Column) of Table 4.1 which have Variable lengths, called V-Binary. For example, Binary number 101 is equal to decimal number 5.

Now let us discuss how to encode the 18 decimal numbers based on CDBS encoding. Column 3 (V-CDBS Column) of Table 4.1 shows CDBS, which is called V-CDBS because it is encoded with Variable length binary strings. The following steps show the details of how to get the V-CDBS codes (binary strings) and these steps are examples for the formal algorithm in Section 4.2.1.

Step 1: In the encoding of the 18 numbers, we suppose that there is one more number before number 1, say number **0**, and one more number after number 18, say number **19**.

Step 2: We firstly encode the middle number with binary string "***1***". The middle number is **10** where 10 is calculated in this way, 10 = round(0+(19–0)/2). The V-CDBS code of number 10 is "1" (see Table 4.1).

Step 3: Next we encode the middle number between 0 and 10, and between 10 and 19. The middle number between 0 and 10 is 5 (5=round(0+(10-0)/2)) and the middle number between 10 and 19 is 15 (15=round(10+(19-10)/2)).

Step 4: To encode number 5, the code size of number 0 is 0 (the V-CDBS code of number 0 *corresponding to* $\boldsymbol{S_L}$ *in Algorithm 4.1* is empty now), and the code size of number 10 is 1 (the V-CDBS code of number 10 *corresponding to* $\boldsymbol{S_R}$ *in Algorithm 4.1* is "1" now with size 1 bit). This is the **Case (b) where size(**$\boldsymbol{S_L}$**) < size(**$\boldsymbol{S_R}$**)** (see *Algorithm 4.1*). Thus based on lines 3 and 4 in Algorithm 4.1, the V-CDBS code of number **5** is "***01***" ("1" → "01").

Step 5: To encode number 15, the 10th code (S_L) is "1" now with size 1 bit, and the 19th code (S_R) is empty now with size 0. This is the **Case (a) where size(S_L) ≥ size(S_R)** (see *Algorithm 4.1*). Therefore based on lines 1 and 2 in Algorithm 4.1, the V-CDBS code of number **15** is "***11***" ("1" ⊕ "1" → "11").

Step 6: Next we encode the middle numbers between 0 and 5, between 5 and 10, between 10 and 15, and between 15 and 19, which are numbers 3, 8, 13 and 17 respectively. The encodings of these numbers are still based on Case (a) or Case (b) in Algorithm 4.1.

In this way, all the numbers except 0 will be encoded because the *round* function will reach the larger value (divided by 2), and we need to discard the V-CDBS code for number 19 since number 19 does not exist actually.

There are two methods to make the encoding the most symmetric.

(1) With Step 1, the total code size of V-CDBS is always equal to the total code size of V-Binary (without Step 1, but the other steps are the same, then their total sizes are not always equal).

(2) If there is no Step 1, we should process the middle numbers in this way. Based on the number 1 and number 18, the middle number is number 10 (10=round((1+18)/2)). Next we should calculate the middle numbers between number 1 and number **9**, and between number **11** and number 18, i.e. number 10 should not be used to calculate the middle numbers. In this way, the middle number between number 1 and number **9** is 5 (5=round((1+9)/2)), and the middle number between number **11** and number 18 is 15 (15=round((11+18)/2)). Next we calculate the middle

numbers between 1 and **4**, between **6** and 9, between 11 and **14**, between **16** and 18. Finally the code of number 1 is that we change the last bit "1" of the code of number 2 to "01" since the round function will not reach number 1.

Both of these two methods calculate the middle numbers in the *most symmetric* way. The larger size codes are used only after the smaller size codes are used up, therefore both of these two methods can guarantee that the total code size of V-CDBS is equal to the total code size of V-Binary.

Also we can encode the decimal numbers 1-18 with Fixed length binary numbers, called F-Binary (see F-Binary Column of Table 4.1). Since 18 needs 5 bits to store, zero or more "0"s should be concatenated ***before*** each code of V-Binary. On the other hand, when representing CDBS using Fixed length, called F-CDBS, we concatenate "0"s ***after*** the V-CDBS codes (see F-CDBS Column of Table 4.1).

With Step 1 to Step 6 above, the formal encoding algorithm in Section 4.2.1 will be easier to understand, and with the following example illustrations for the total code size, the formal size analysis in Section 4.2.2 will be easier to understand.

Example 4.4 *It can be seen from Table 4.1 that V-Binary has one code "1" with size 1 bit, two codes "10" and "11" with sizes 2 bits, four codes "100", "101", "110" and "111" with sizes 3 bits, etc., and the total size of V-Binary is 64 bits. Also we can see that V-CDBS has one code "1" with size 1 bit, two codes "01" and "11" with sizes 2 bits, four codes "001", "011", "101" and "111" with sizes 3 bits, etc., and the total size of V-CDBS is also 64 bits. This means that V-CDBS is as compact as the existing binary number encoding of consecutive numbers. It is similar for F-Binary and F-CDBS (they both have size 90 bits).*

Example 4.5 *Table 4.1 shows that V-Binary has smaller total code size than F-Binary. However, we also need to store the length of each V-Binary code, the maximal length for a code is 5, e.g. the length of "10010" is 5. We need to store this 5 using fixed length of bits ("101"; 3 bits). The lengths of other codes should also be stored using fixed length of bits (3 bits), therefore the total code size for V-Binary is* $\boldsymbol{3} \times 18+64=118$ *bits which is larger than the bits required by F-Binary. It is similar for V-CDBS (118 bits) and F-CDBS (90 bits).*

In the later parts of this thesis, we mainly focus on V-CDBS to introduce the theorems and properties; these properties can be applied to F-CDBS also.

4.2.1 CDBS Encoding Algorithm

Because F-CDBS is that some "0"s are concatenated after the V-CDBS codes, we focus on V-CDBS to introduce the algorithm.

Algorithm 4.2 is the V-CDBS encoding algorithm. We use the procedure V-CDBS_SubEncoding to get all the codes of the numbers. *Finally number 0 and number (TN+1) should be discarded since they do not exist actually.*

V-CDBS_SubEncoding is a recursive procedure, the input of which is an array *codeArr*, the left position "P_L" and the right position "P_R" in the array *codeArr*. This procedure assigns codeArr[P_M] (corresponding to S_M in Algorithm 4.1) using the AssignMiddleBinaryString algorithm (Algorithm 4.1), then it uses the new left and right positions to call the V-CDBS_SubEncoding procedure itself, until each (except the 0th) element of the array *codeArr* has a value.

Algorithm 4.2: V-CDBS Encoding (*TN*)
Input: A positive integer *TN*
Output: The *V-CDBS codes* for numbers 1 to *TN*

Description:
1: suppose there is one more number before the first number, called number **0**, and one more number after the last number, called number **(*TN*+1)**
2: Define an array *codeArr[0,TN+1]* //the size of codeArr is //TN+2; each element of the codeArr is empty at the beginning
3: **V-CDBS_SubEncoding**(*codeArr*, 1, *TN*)
4: **discard** the 0th and $(TN+1)^{th}$ elements of the *codeArr*

***Procedure* V-CDBS_SubEncoding** (*codeArr*, P_L, P_R)
/*V-CDBS_SubEncoding is a recursive procedure; *codeArr* is an array, P_L is the left position, and P_R is the right position*/
1: P_M = round((P_L+P_R)/2)
2: **if** $P_L+1<P_R$ **then**
3: *codeArr*[P_M]= assignMiddleBinaryString(*codeArr*[P_L], *codeArr*[P_R])
4: V-CDBS_SubEncoding(*codeArr*, P_L, P_M)
5: V-CDBS_SubEncoding(*codeArr*, P_M, P_R)
6: **end if**

Note that S_L and S_R in the input of Algorithm 4.1 can be empty when Algorithm 4.1 is called by V-CDBS_SubEncoding here. If S_L and S_R are both empty, their sizes are both equal to 0, and S_M is "1" based on lines 1 and 2 in Algorithm 4.1. If S_L is empty and S_R is not empty, size(S_L) < size(S_R), and we process S_M based on lines 3 and 4 in Algorithm 4.1 ($S_M \prec S_R$). If S_L is not empty and S_R is empty, size(S_L) > size(S_R), and we process S_M based on lines 1 and 2 in Algorithm 4.1 ($S_L \prec S_M$).

Given a positive integer TN, Algorithm 4.2 can encode all the numbers between 1 and TN with V-CDBS codes.

The V-CDBS encoding is like the binary search. As we know, the binary search will not miss any values in the search, therefore Algorithm 4.2 can encode each

number without missing. This property is very important to guarantee that our approach can completely encode all the numbers.

Lemma 4.4 *All the V-CDBS codes are ended with "1".*

Proof: Lemma 4.2 guarantees that Lemma 4.4 holds.

Theorem 4.5 *All the V-CDBS codes are lexicographically ordered.*

Proof: Algorithm 4.2 is about the insertion at different places, and Algorithm 4.1 guarantees that all the insertions at different places are lexicographically ordered, and the total lexicographical order is also kept.

Example 4.6 *The V-CDBS codes in Table 4.1 are lexicographically ordered from top to bottom.*

Lemma 4.4 and Theorem 4.5 guarantee that the ***conditions*** in Theorem 4.1 and Corollary 4.3 are satisfied, therefore we can ***insert without re-labeling*** in updates based on ***V-CDBS***.

4.2.2 Size Analysis

In this section, we analyze the sizes required by different encodings.

V-Binary For V-Binary, one number ("1"; see Table 4.1) is stored with one bit, two numbers ("10" and "11") are stored with 2 bits, four numbers ("100", "101", "110" and "111") are stored with 3 bits, ⋯, therefore the total size of V-Binary is

$$1 \times 1 + 2 \times 2 + 2^2 \times 3 + 2^3 \times 4 + \cdots + 2^n \times (n+1)$$

$$= n \times 2^{n+1} + 1 \quad (4.1)$$

(see Appendix C1 for how to get formula (4.1))

Suppose the total number of codes is *N*, which should be equal to $2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1$. Thus formula (4.1) becomes to

$$N \log(N+1) - N + \log(N+1) \quad (4.2)$$

V-CDBS When considering V-CDBS, it has one code ("1") stored with one bit, two codes ("01" and "11") stored with two bits, four codes ("001", "011", "101" and "111") stored with three bits, ···, therefore V-CDBS has the same code size as V-Binary (see Formula (4.2)).

In addition, since V-Binary and V-CDBS have variable lengths, we need to store the size of each code. A fixed-length number of bits are used to store the length of the codes. The maximal length for a code is $\log(N)$. To store this length, the bits required are $\log(\log(N))$, and the total bits required to store the lengths of all the variable codes are $N \log(\log(N))$. When taking formula (4.2) into account, the total sizes of V-Binary and V-CDBS are both

$$N \log(N+1) + N \log(\log(N)) - N + \log(N+1) \quad (4.3)$$

F-Binary To store *N* numbers with fixed lengths, the size required is

$$N \log(N) \quad (4.4)$$

The length of the F-Binary code also needs to be stored, but needs to be stored ***only once***, which needs size $\log(\log(N))$. Therefore the total size for F-Binary is

$$N\log(N) + \log(\log(N)) \tag{4.5}$$

F-CDBS has the same total code size as formula (4.5).

Note that for simplicity, we omit the *ceiling* functions on the *log* functions in all the formulas.

Theorem 4.6 *V-CDBS and F-CDBS are the most compact variable and fixed length binary string encodings which support updates efficiently.*

Proof: As we know, the V-Binary and F-Binary are encodings for the ***consecutive*** decimal numbers and there are no gaps between any two ***consecutive*** numbers, thus V-Binary and F-Binary are the most compact encodings. In addition, from the above size analysis, we know that V-CDBS and F-CDBS have the same total sizes as V-Binary and F-Binary respectively[2]. Furthermore, based on Lemma 4.4, Theorem 4.5 and Theorem 4.1, we can insert a binary string between any two ***consecutive*** V-CDBS or F-CDBS codes without re-encoding the existing numbers. Therefore, V-CDBS and F-CDBS are the most compact dynamic encodings.

4.3 Applying CDBS to Different Labeling Schemes

In this section, we mainly illustrate how V-CDBS can be applied to different labeling schemes. F-CDBS is similar since it is that some zeros are concatenated after the V-CDBS codes.

---

[2] We assume the consecutive numbers starting from 1. If the consecutive numbers start from 0, our approach can use "0" as one code in the encoding, then our approach still has the same size as Binary, but each time when we want to insert a code before "0", we need to insert a code before the second code, and always put "0" as the first code.

We firstly describe a property which is the *second foundation* of this thesis (the first one is Algorithm 4.1).

Property 4.1 *V-CDBS is orthogonal to specific labeling schemes, thus it can be applied to different labeling schemes or other applications which need to maintain the order in updates.*

Property 4.1 states that V-CDBS can be broadly applied to different labeling schemes.

When we replace the "start" and "end" values 1-18 of the containment scheme [83] (similar for other containment schemes [3, 26, 56, 80]) in Figure 2.3 with the V-CDBS codes in Table 4.1 and based on the lexicographical comparison, a V-CDBS based containment labeling scheme is formed, called *V-CDBS-Containment*.

Example 4.7 *Figure 4.1 shows the V-CDBS-Containment scheme. The "start" and "end" values are replaced with V-CDBS codes. The "level" values are still the same as the decimal numbers in Figure 2.3 which can be used to calculate the "level" difference for the parent-child determination. Note that the decimal numbers are stored in binary numbers in the implementation. Based on the lexicographical order, we can compare the "start" and "end" values for the ancestor-descendant etc. determinations. V-CDBS-Containment has the same total label size as the existing containment scheme, therefore it will not decrease the query performance. More important, based on V-CDBS-Containment, we can process updates efficiently (see Section 4.4)*

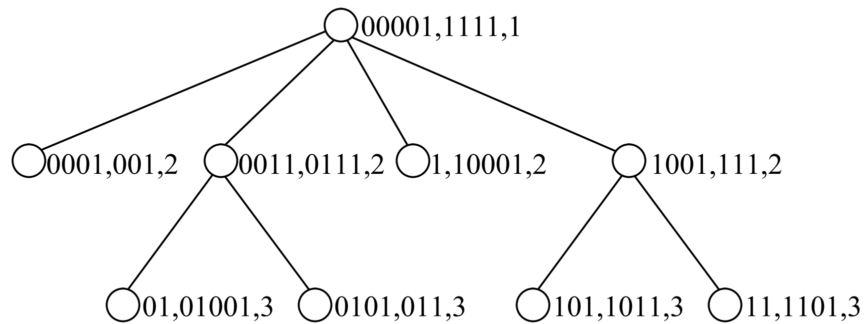


Figure 4.1: V-CDBS-Containment scheme

Similarly, we can replace the decimal numbers in the prefix labeling scheme [70] (see Figure 2.4) with V-CDBS codes, then a V-CDBS based prefix labeling scheme is formed, called *V-CDBS-Prefix*. We use the following example to show *V-CDBS-Prefix*.

Example 4.8 *From Figure 2.4, we can see that the root has 4 children. To encode 4 numbers based on Algorithm 4.2, the V-CDBS codes will be "001", "01", "1" and "11". Similarly if there are two siblings, their self_labels are "01" and "1" based on Algorithm 4.2. Figure 4.2 shows V-CDBS-Prefix scheme.*

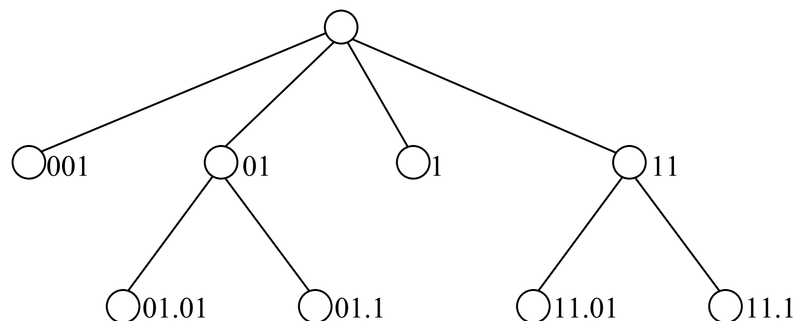


Figure 4.2: V-CDBS-Prefix scheme (for Figure 2.4)

Similarly we can apply V-CDBS to the prime labeling scheme to record the document order rather than calculate the SC values (see Section 2.3 of Chapter 2 for the prime labeling scheme and the SC value calculation). Based on V-CDBS encoding, Prime can also maintain the orders with very cheap cost. However because Prime employs the modular and division operations to determine the ancestor-descendant etc. relationships, its query efficiency is quite bad (see Section 4.5.2.2 for experimental results). Thus we do not discuss in detail how V-CDBS is applied to Prime.

For the containment and prime number scheme, we only need to know the total number of nodes of an XML tree, then we can replace the decimal numbers with CDBS encoding which is very efficient in initial labeling. However, for the prefix scheme, we need to know the number of siblings of each first child node. If the size of the XML tree is small, it is not a problem to get the number of siblings of a node, however if the size of the XML tree is very large, it will be slow to get the number of siblings for each first child node.

It may be argued that V-CDBS only has the orders but does not have the exact position of each code, which is a deficiency when compared to the V-Binary codes. For example, from a V-Binary code "110", we can immediately know that "110" corresponds to the decimal number 6. However, if we delete the V-Binary codes "100" and "101", "110" is now not the 6th number but the 4th number in order. In this thesis, we focus on the dynamic XML data in which there are a lot of deletions and insertions, therefore ***V-Binary does NOT have merits over V-CDBS in processing the nth position label***. V-Binary and V-CDBS both need to sort and get the position in the dynamic environment of XML data.

In addition, it is not the case that V-CDBS can not immediately get the exact position in the static environment of XML data. Based on an inverse processing of Algorithm 4.2, we can get the exact position of each V-CDBS code by calculations only (see Appendix D). However, if an XML tree is static, we can directly use V-Binary rather than V-CDBS. If XML is dynamic, no encoding can calculate the positions immediately.

4.4 Processing of XML Updates Based on Different Labeling Schemes Encoded with CDBS

Based on CDBS, in this section, we discuss how to efficiently process different XML updates. Algorithm 4.1 is the foundation to efficiently process XML updates. Before we start the discussion of this section, we review the idea of Algorithm 4.1: given two lexicographically ordered binary strings ended with "1", we can find a binary string lexicographically between the given two binary strings. If the size (bit number) of the left binary string is larger than or equal to the size of the right binary string, the inserted binary string is that we concatenate one more "1" at the end of the left binary string. If the size of the left binary string is smaller than the size of the right binary string, the inserted binary string is that we change the last bit "1" of the right binary string to "01". In this way, the inserted binary string is lexicographically between the left binary string and the right binary string.

Section 4.4.1 discusses how to process the leaf node updates. We discuss how to process the internal node updates in Section 4.4.2. When a subtree is inserted into

XML, Section 4.4.3 describes how to make the label size of the inserted subtree increase slowly. Section 4.4.4 discusses the uniformly and skewed insertions.

4.4.1 Leaf Node Updates

The deletion of a leaf node will not affect the *relative* orders of the nodes in XML, hence we mainly discuss how to process the insertions based on V-CDBS.

In this section, we use examples to show how to process the leaf node insertion based on V-CDBS-Prefix (see Figure 4.2) and V-CDBS-Containment (see Figure 4.1).

Example 4.9 *If we want to insert a sibling node before "01.01" in Figure 4.3, the self_label of the inserted node is "001" (see lines 3 and 4 in Algorithm 4.1; the left binary string is empty and the right binary string is the self_label "01" of "01.01"); the complete label of the inserted node is "01.001". Theorem 4.1 guarantees that we need not re-label the existing nodes but we can keep the orders. The insertions at other places also need not re-label the existing nodes.*

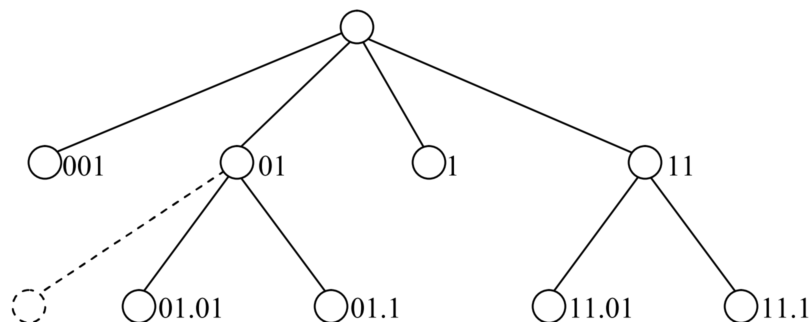


Figure 4.3: Leaf node insertions based on V-CDBS-Prefix scheme

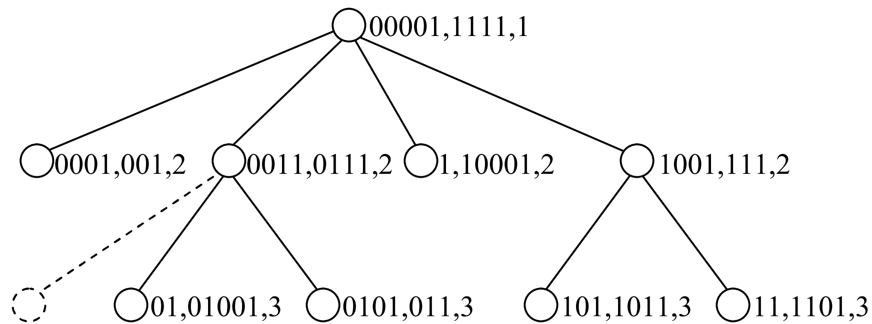


Figure 4.4: Leaf node insertions based on V-CDBS-Containment scheme

Example 4.10 *Similarly if we insert a sibling node before "01,01001,3" in Figure 4.4, we should insert two values ("start" and "end") between the start of "0011,0111,2" i.e. "0011" and the start of "01,01001,3" i.e. "01". Corollary 4.3 guarantees that we can insert two binary strings between "0011" and "01" with the orders kept. Based on Algorithm 4.1, the two inserted binary strings are "00111" and "001111". The complete label of the inserted node is "00111,001111,3". Obviously "0011" $\prec$ "00111" $\prec$ "001111" $\prec$ "01" lexicographically. We need not re-label the existing nodes, but we can keep the containment scheme working correctly to determine all the relationships.*

After insertion, we can further insert other nodes before the inserted node without re-labeling the existing nodes and with the orders kept.

Next we use examples to show how *inefficient* the existing prefix [70] and containment [83] schemes process the updates.

Example 4.11 *If we want to insert a sibling node before "2.1" in Figure 4.5 based on the existing prefix scheme, the label of the inserted node is "2.1" and the existing*

"2.1" and "2.2" should be changed to "2.2" and "2.3". If the existing "2.1" and "2.2" have descendants, the labels of these descendants should be changed also.

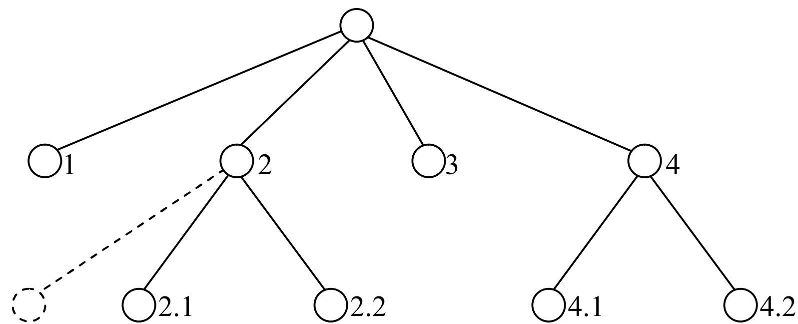


Figure 4.5: Leaf node insertions based on the existing prefix scheme

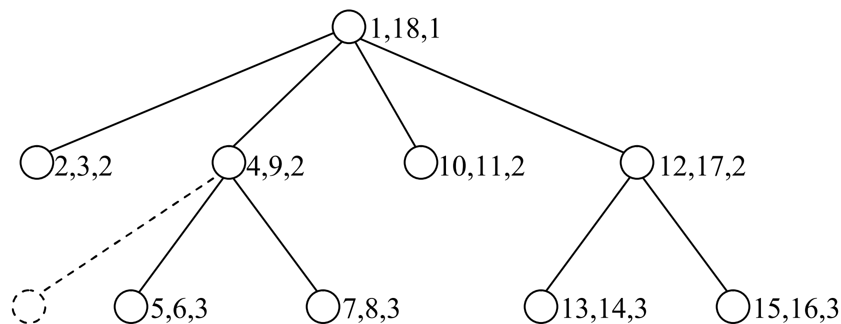


Figure 4.6: Leaf node insertions based on the existing containment scheme

Example 4.12 *If we want to insert a sibling node before "5,6,3" in Figure 4.6 based on the existing containment scheme, the label of the inserted node is "5,6,3" and the existing labels except "2,3,2" should all be changed. The end values of "1,18,1" and "4,9,2" should be added with 2; the new labels are "1,20,1" and "4,11,2". The start and end values of all the other labels except the first three (in document order) should be added with 2; for instance, label "10,11,2" will be changed to "12,13,2". It can be*

seen that the existing containment scheme needs to re-label many nodes when a node is inserted into the XML tree which is very inefficient.

Prime needs to re-calculate the SC values in updates which is very expensive (see Section 4.5 for the experimental results).

Sometimes Float-point [6] and OrdPath [64] also need not re-label the existing nodes. The update performance differences among Float-point, OrdPath and our approaches can be seen in Section 5.5.2 of Chapter 5.

CDBS encoding can be applied to the P-Containment scheme introduced in Chapter 3 to efficiently process the leaf node updates also.

4.4.2 Internal Node Updates

In [74], the internal node update problem has been studied which shows that all the existing labeling schemes have expensive internal node update cost.

When inserting an internal node, the existing containment scheme needs to re-label all the nodes after this inserted node in document order (similar to Example 4.12), all prefix schemes need to re-label the descendant nodes of the inserted node (the prefixes of all the descendants should be changed), and Prime also needs to re-label all the descendant nodes with the new inserted label multiplying all the labels of the descendants, in addition Prime needs to re-calculate the SC values.

Furthermore, when deleting an internal node from an XML tree, all the containment, prefix and prime labeling schemes should re-label all the descendant nodes.

That is to say, all the existing labeling schemes are not appropriate to process the internal node updates. When V-CDBS is applied to the existing containment scheme, V-CDBS-Containment can process the "start" and "end" values efficiently, but because the level values of all the descendants should be increased by 1, the update cost is not so cheap. Furthermore, when V-CDBS is applied to the existing prefix scheme, V-CDBS-Prefix can not process the internal node updates efficiently since the prefixes of all the descendants should be changed when an internal node is inserted into or deleted from an XML tree. This is the drawback of the existing labeling schemes, but not the drawback of CDBS encoding.

Based on P-Containment scheme introduced in Chapter 3 and V-CDBS encoding, we can decrease internal node update cost.

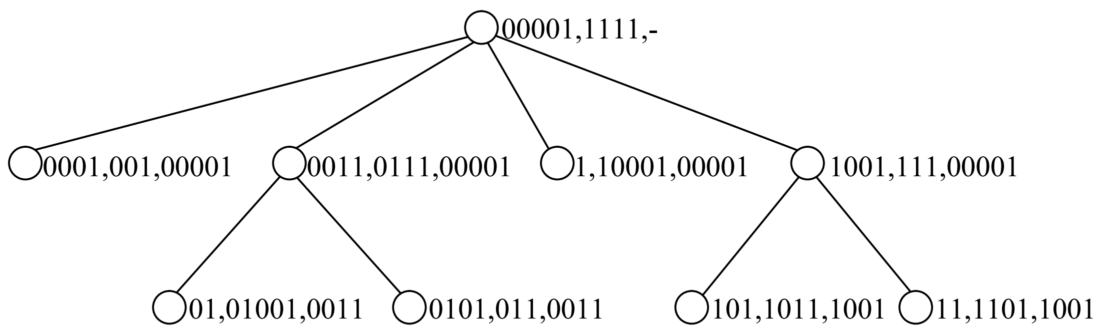


Figure 4.7: V-CDBS-P-Containment scheme

We firstly review P-Containment scheme (see Figure 3.1(b)). In P-Containment scheme, we store the "parent_start" value rather than the "level" value of the existing containment scheme. If two nodes have the same "parent_start" value, they are siblings. If the "start" value of one node is equal to the "parent_start" value

of another node, the first node is the parent of the second node. When we apply V-CDBS encoding to P-Containment scheme, Figure 4.7 shows V-CDBS-P-Containment scheme.

The following Properties 4.2 and 4.3 show that ***V-CDBS-P-Containment*** has much cheaper internal node update cost.

Property 4.2 *Based on **V-CDBS-P-Containment**, when an internal node is inserted into an XML tree, the "parent_start" of the inserted internal node should refer to the "start" of the parent of this internal node, the "parent_start"s of the children of the inserted internal node should be modified to refer to the "start" of the inserted internal node, and the "parent_start"s of all the descendants of the inserted internal node except the children need not be changed.*

Example 4.13 *Figure 4.8 shows that we insert an internal node "u" based on V-CDBS-P-Containment scheme. The "start" of the inserted node "u" should be a binary string between the "start" of the root and the "start" of "0001,001,00001", i.e. between "00001" and "0001". Based on Algorithm 4.1, the "start" of node "u" will be "**000011**" (size("00001") > size("0001"); "**000011**" = "00001" ⊕ "1"). Similarly the "end" of the inserted node "u" should be between the "end" of "1,10001,00001" and the "start" of "1001,111,00001", i.e. between "10001" and "1001". Based on Algorithm 4.1, the "end" of node "u" will be "**100011**" (size("10001") > size("1001"); "**100011**" = "10001" ⊕ "1"). The "parent_start" value of the inserted node "u" should be equal to the "start" value of the root, i.e. "**00001**". The "parent_start" values of "0001,001,00001", "0011,0111,00001" and "1,10001,00001" should be modified to refer to the "start" value of node "u", i.e.*

change "00001" to "000011". The "start", "end" and "parent_start" values of the "01,01001,0011" and "0101,011,0011" (they are the descendant nodes of the children of node "u") need not be changed.

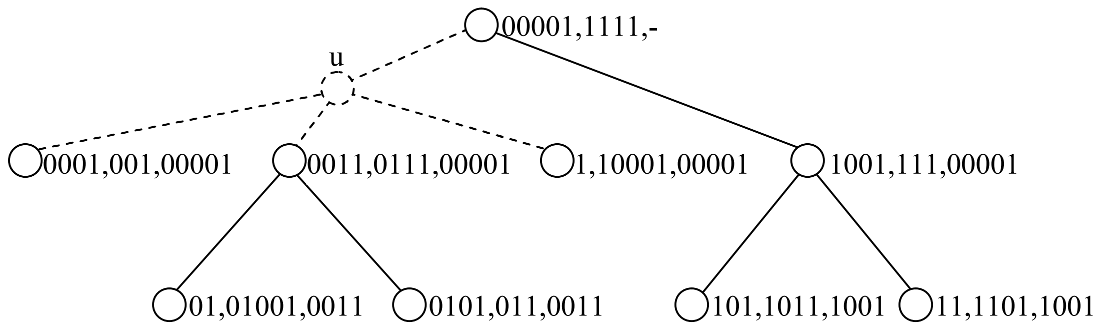


Figure 4.8: Internal node insertions based on V-CDBS-P-Containment scheme

Theorem 4.7 *The P-Containment shown in Figure 3.1(b) can not decrease the internal node insertion cost when the decimal numbers in Figure 3.1(b) are stored with V-Binary or F-Binary encodings.*

Proof: The "**start**" values of the descendants based on ***V-Binary*** and ***F-Binary*** need to be changed when inserting an internal node, therefore if we use the "start" of the parent as the "parent_start" of the child, we still need to change the "parent_start" values. The insertion cost will not be decreased.

Only V-CDBS-P-Containment (or F-CDBS-P-Containment) is efficient to process the internal node insertion.

The following property shows that V-CDBS-P-Containment has cheaper internal node deletion cost.

Property 4.3 *When an internal node is deleted from an XML tree, **V-CDBS-P-Containment** only needs to modify the "parent_start" values of the child nodes of the deleted node to refer to the "start" value of the parent of the deleted node, but need not modify the "parent_start" values of the descendant nodes of these child nodes.*

Though internal node insertions and deletions do not happen so often in practice, the V-CDBS-P-Containment technique can help to reduce the internal node update cost if the internal node updates happen. In addition, the "parent_start" introduced in P-Containment scheme can help to determine the parent-child relationship, especially the *sibling* relationship very fast. Moreover, the "parent_start" is useful later in Chapter 5 to completely avoid re-labeling.

It is not intuitive to improve the prime scheme to process the internal node updates efficiently since the labels of all the descendants need to be modified. It is easy to understand that the internal node updates for the existing containment and prefix schemes need to re-label all the descendant nodes of the inserted or deleted node, therefore we do not repeat how they process the internal node updates. Here we use an example to show how the prime scheme process the internal node updates.

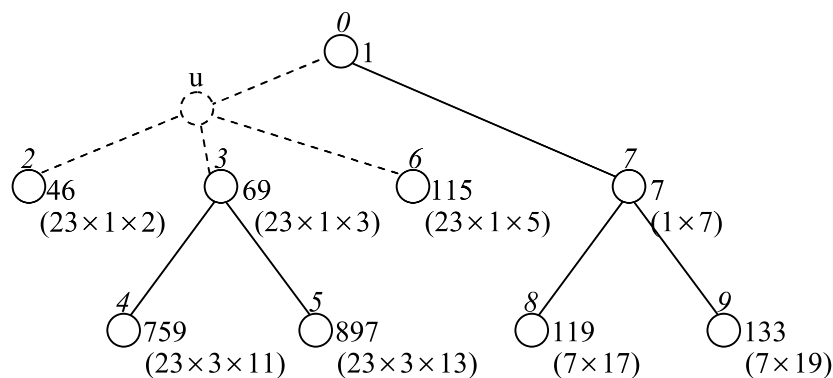


Figure 4.9: Internal node insertions based on the prime number scheme

Example 4.14 *Figure 4.9 shows that we insert an internal node "u" based on the prime scheme. The next available prime number is 23, thus the label of node "u" is "23" ("1×23"). The labels of all the descendant nodes of the inserted node "u" should be multiplied by the label "23" of node "u" (see Figure 4.9). Thus the internal node update cost based on Prime is very expensive. In addition to that, the orders of all the nodes after this inserted internal node should be added with 1 (see Figure 4.9), and Prime needs to re-calculate the SC values based on the new orders which is also very expensive. Therefore Prime can not process internal node updates efficiently.*

4.4.3 Subtree Updates

The deletion of a subtree will not affect the *relative* orders of the rest nodes in XML, hence we mainly discuss how to process the insertion of a subtree based on V-CDBS.

When a subtree is inserted into XML, we can process the insertion of this subtree as the insertion of nodes one by one. However, this kind of insertion will make the label size increase fast (see Section 4.4.4 for more details). That is not what we expected. We use the following method to process the insertion of a subtree.

Example 4.15 *Figure 4.10 shows that a subtree is inserted into the XML tree based on V-CDBS-Prefix. The label of the root of the subtree is an insertion between "01" and "1". Based on Algorithm 4.1, the inserted label is "**011**" (see lines 1 and 2 of Algorithm 4.1; "011" = "01" ⊕ "1"). Based on Algorithm 4.2, the self_labels of the three child nodes of the subtree are "01", "1" and "11", and their complete labels are "011.01", "011.1" and "011.11". If the subtree is inserted node by node, their labels are "011", "011.1", "011.11" and "011.111" with larger total size.*

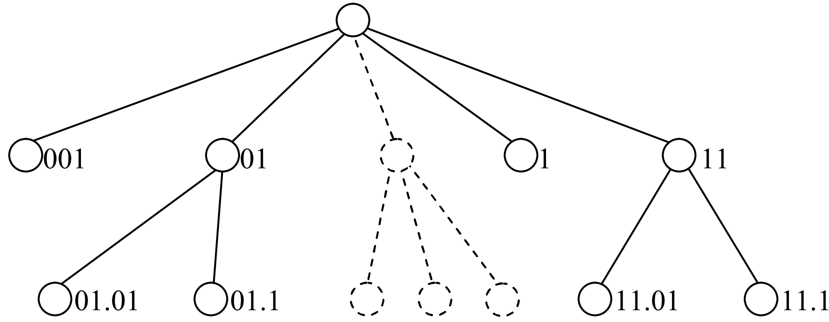


Figure 4.10: Subtree insertion based on V-CDBS-Prefix scheme

Example 4.16 *Figure 4.11 shows that a subtree is inserted into the XML tree based on V-CDBS-P-Containment. For the subtree, we need to insert 8 binary strings (4 nodes; 8 "start" and "end" values) between the V-CDBS codes "**0111**" (the "end" of "0011,0111,00001) and "**1**" (the "start" of "1,10001,00001") in Figure 4.11. We use Algorithm 4.2 to process the insertion of the 8 binary strings, and "**0111**" and "**1**" can be thought as the V_CDBS codes for number **0** and number **(TN+1)=(8+1)=9** in Algorithm 4.2. The middle number is the 5th number where 5= round(0+(9-0)/2). The $\boldsymbol{S_L}$ is "**0111**" with size 4 bits, and the $\boldsymbol{S_R}$ is "**1**" with size 1 bit, therefore according to lines 1 and 2 in Algorithm 4.1 (called by Algorithm 4.2), the V-CDBS code of the 5th number is "**01111**" (see lines 1 and 2 of Algorithm 4.1; "01111" = "0111" ⊕ "1"). Similarly we can insert the V-CDBS codes for the rest 7 numbers. Finally the V-CDBS codes for the 8 numbers are "01110001", "0111001", "011101", "0111011", "01111", "0111101", "011111" and "0111111". They are lexicographically ordered between "0111" and "1". The "start", "end" and "parent_start" values of the four nodes of the inserted subtree are "**01110001, 0111111, 00001**", "**0111001, 011101, 01110001**", "**0111011, 01111, 01110001**" and "**0111101, 011111,***

01110001". *If the scheme is the existing containment scheme, it is not a problem to get the "level" value for each node of the inserted subtree compared with P-Containment.*

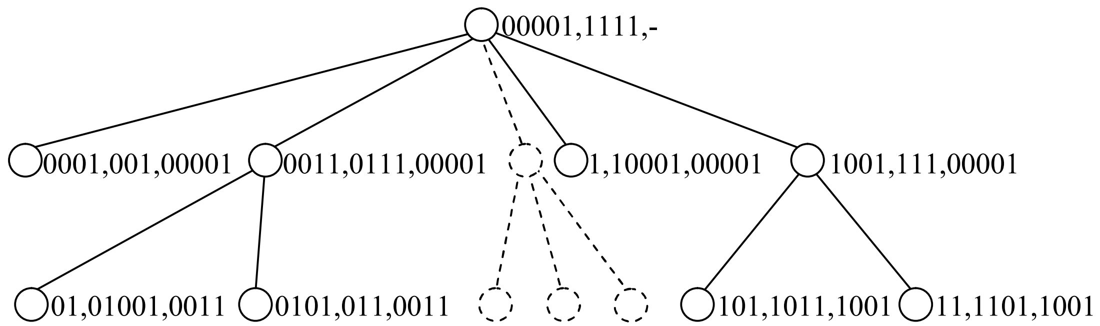


Figure 4.11: Subtree insertion based on V-CDBS-P-Containment scheme

In this way, the total label size of the inserted subtree is smaller than the size that we repeat the insertion node by node if not necessary (see Section 4.5.3.3 for the experimental results).

The insertion of a subtree will make the existing containment and prefix schemes re-label the existing nodes, and because a subtree contains many nodes, it is easier to lead the Float-point [6] and OrdPath [64] to re-labeling.

4.4.4 Uniformly and Skewed Frequent Updates

The size analysis in Section 4.2.2 is based on the initial encoding. Algorithm 4.2 shows that our encoding algorithm is step by step insertions of nodes evenly at different places. Therefore if a sequence of nodes are inserted randomly at different places of XML, the size analysis in Section 4.2.2 is still valid, and the *query performance will **not** be decreased*.

For the case where nodes are always inserted at a fixed place (we call this kind of insertion *skewed insertion*) of XML, the size of V-CDBS increases fast. [23] proves that any deterministic labeling scheme which does not re-label nodes must in the worst case assign *one* label with size *O(N)*. V-CDBS can not escape from this claim also, i.e. the label size of V-CDBS increases linearly in the *worst case*. *O(N)* is the *upper bound* of the size of V-CDBS. OrdPath [64] also has this skewed insertion problem. [68] uses B-tree to balance the update and lookup performance.

[13] studies that the insertions in XML are often segments e.g. subtrees, and the insertion of single node seldom happens. As we can see from Section 4.4.3, the insertion of a *subtree* will ***not*** cause the label size ***increase*** fast. The above analysis also shows that CDBS at least ***work very well*** when the insertions are *randomly at different places* of XML. Even in the *skewed insertion* environment, CDBS ***still works the best*** to answer queries since we dramatically decrease the update time, and with the saved time, we can answer queries faster than other labeling schemes (other labeling schemes need re-labeling which needs a lot of time; see the experimental results in Section 4.5.3 of this chapter and in Section 5.5.2 of Chapter 5).

4.5 Experimental Evaluation and Comparisons

4.5.1 Experimental Setup

The experimental setup here is used at all the experiments in this thesis whereas there are other special explanations.

We evaluate and compare the performance of different labeling schemes. The schemes containing a “CDBS” or “CDQS” are all schemes proposed in this thesis; all the others are prior schemes. The schemes with a “-Prefix” at the end of the scheme names are prefix schemes, and with a “-Containment” at the end of the scheme names are containment schemes.

All the schemes are implemented in Java and all the experiments are carried out on a 3.0 GHz Pentium 4 processor with 1 GB RAM running Windows XP Professional.

Table 4.2 shows the characteristics of the test datasets. D1 is from [63], D3 and D4 are from [71], and all of them are real-world XML data. D2 is a benchmark generated by XMark [76]. We choose these datasets because they have different characteristics and they are widely used in different papers for XML performance study. We also test our approaches on other datasets from [63] and [71] and similar results are found; here we focus on D1-D4 to report all the experimental results in this thesis.

Table 4.2: Test datasets

Datasets	Topics	# of files	Max/average fan-out for a file	Max/average depth for a file	Total # of nodes for each dataset	Size (MB)
D1	Shakespeare’s play	37	434/48	6/5	179689	7.53
D2	XMark	1	25500/3242	12/6	1666315	82
D3	Treebank	1	56384/1623	36/8	2437666	111
D4	DBLP	1	328858/65930	6/3	3332130	127

4.5.2 Performance Study on Static XML Data

From now on, we mainly study how the existing binary encoding and CDBS encodings (see Table 4.1 for different encodings) are applied to the existing containment and prefix labeling schemes to process the queries and updates.

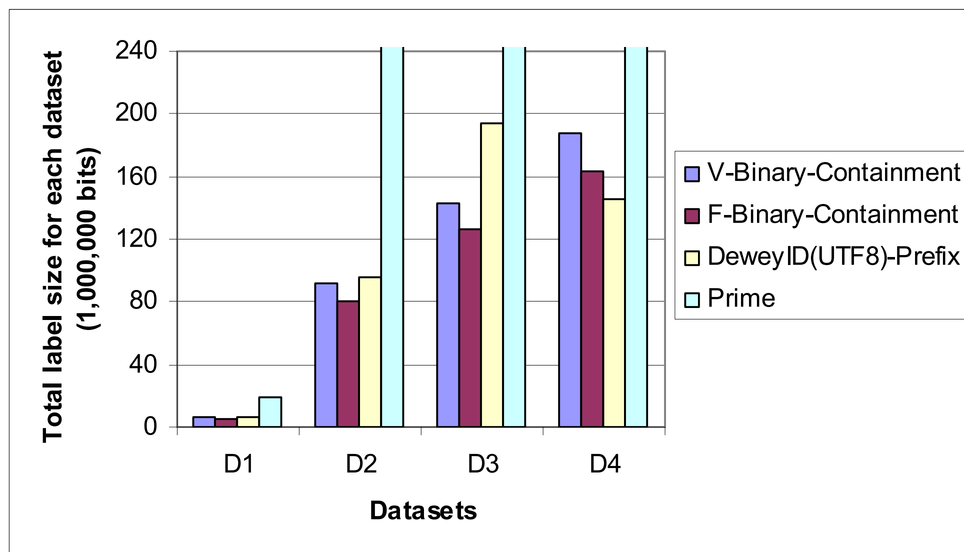
Static XML is not the emphasis of this thesis, thus we only compare the *label size* and *query performance* of different encodings in this section.

4.5.2.1 Storage Requirement

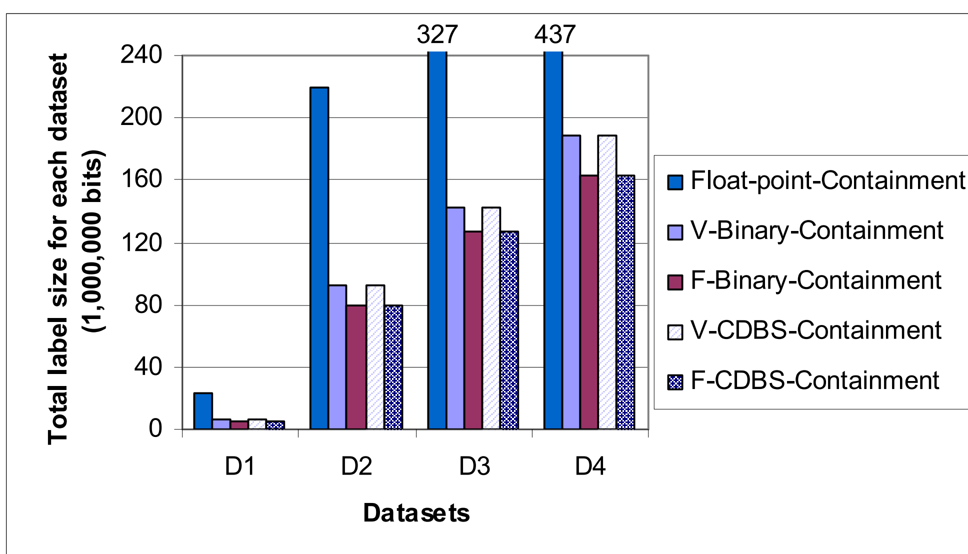
Figure 4.12(a) shows the label sizes of the existing containment, prefix and prime labeling schemes for the four datasets shown in Table 4.2. Prime [74] labeling scheme has larger label size than the containment and prefix schemes because it skips a lot of integer numbers to get the prime numbers and it uses the multiplications of the numbers for the labels which both make its label size very large. If the XML tree is deep (see the characteristics of different datasets in Table 4.2), the prefix scheme has larger label size than the containment scheme (see the label sizes for D2 and D3); if the XML tree is shallow, the prefix scheme has smaller label size than the containment scheme (see the label sizes for D4).

Figure 4.12(b) is the comparison between the existing containment schemes and CDBS containment scheme. Float-point-Containment [6] has larger label size than other containment labeling schemes. V-CDBS-Containment has the same label size as V-Binary-Containment, and F-CDBS-Containment has the same label size as F-Binary-Containment. These show that V-CDBS and F-CDBS are the most compact variable and fixed length encodings.

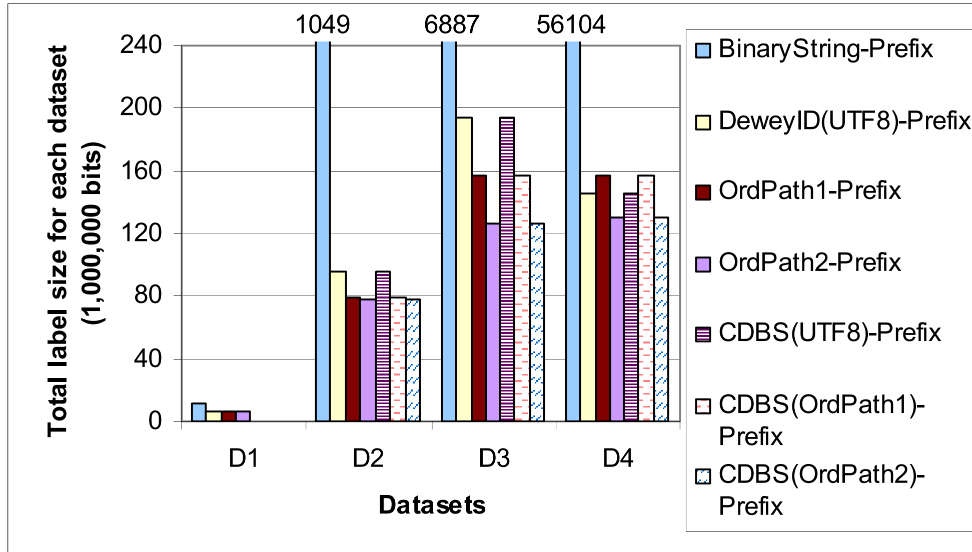
When V-Binary, F-Binary, V-CDBS, and F-CDBS are applied to the P-Containment scheme, V-CDBS-P-Containment still has the same label size as V-Binary-P-Containment, and F-CDBS-P-Containment has the same size as F-Binary-P-Containment. Here we do not show them in Figure 4.12(b).



(a) Label sizes of different schemes



(b) Label sizes of containment schemes



(c) Label sizes of prefix schemes

Figure 4.12: Label sizes of different labeling schemes

For the prefix schemes, based on the size (length) of each code of V-CDBS (similar for F-CDBS), we can use the UTF8 [78] or OrdPath [64] encoding to process the delimiters. If we use UTF8 to process the delimiters, V-CDBS(UTF8)-Prefix has the same label size as DeweyID(UTF8)-Prefix. If we use OrdPath encodings to process the delimiters, V-CDBS(OrdPath)-Prefix has smaller label size than OrdPath-Prefix since we do not waste the even numbers. The UTF8 and OrdPath encodings are existing techniques, In Section 5.2 of Chapter 5, we will show how to process the delimiters based on CD*Q*S (see Example 5.6). It can be seen from Figure 4.12(c) that BinaryString-Prefix [23] has much larger label size than other prefix labeling schemes. Generally OrdPath1-Prefix and OrdPath2-Prefix have smaller label size than DeweyID(UTF8)-Prefix though OrdPath1-Prefix and OrdPath2-Prefix waste a lot of even numbers. This is because the encodings of OrdPath1 and OrdPath2 are more compact. However, though OrdPath has smaller label size, its query performance is

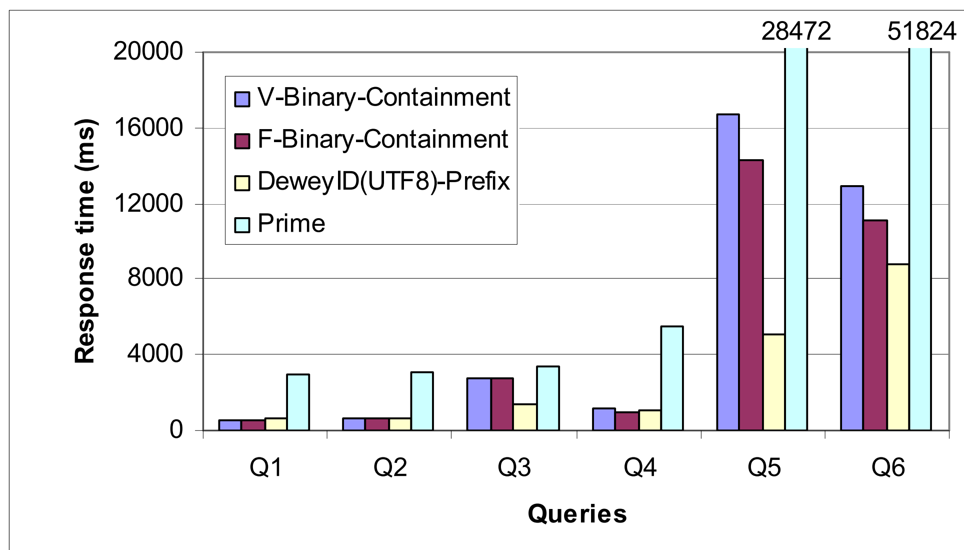
worse because it needs more time to decode its encodings and needs more time to determine the levels based on the odd and even numbers.

4.5.2.2 Query Performance

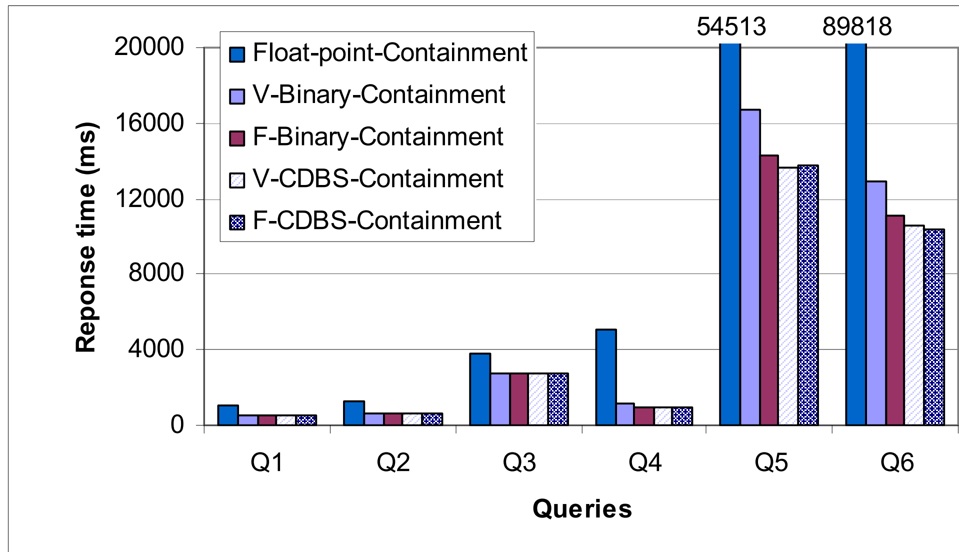
We test the query performance based on all XML files in the Shakespeare's play dataset (D1) (see Table 4.2) and for a more sizeable data workload we scaled up (replicate) D1 10 times as described in [70]. The ordered and un-ordered queries and the number of nodes retrieved are shown in Table 4.3.

Table 4.3: Test queries on the scaled D1

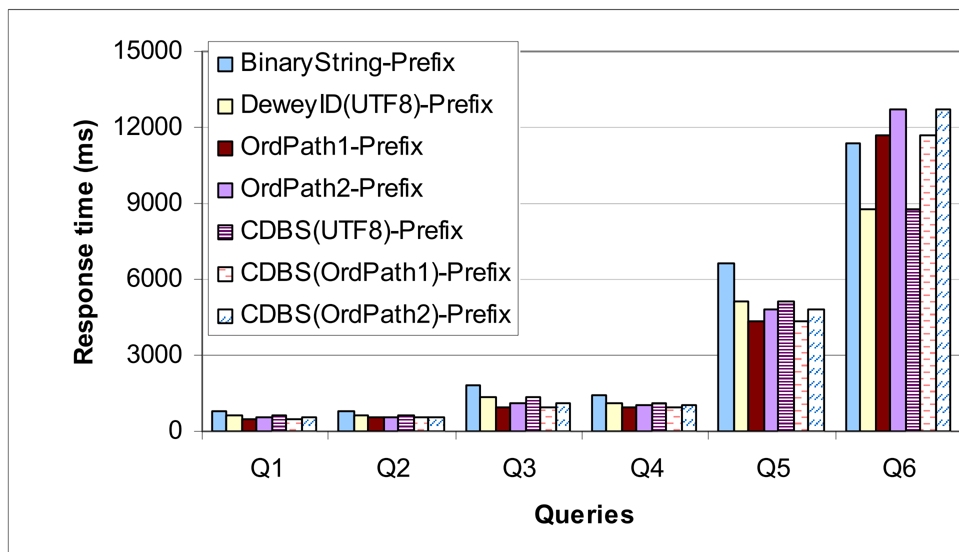
Queries		# of nodes Retrived
Q1	/play/act[4]	370
Q2	/play//personae[./title]/pgroup[.//grpdescr]/persona	2690
Q3	/play/personae/persona[12]/preceding-sib ling::*	4240
Q4	/play//act[2]/following::speaker	184060
Q5	/play/act/scene/speech	309330
Q6	/play//line	1078330



(a) Response time of different schemes



(b) Response time of containment schemes



(c) Response time of prefix schemes

Figure 4.13: Query performance of different labeling schemes

Different structural join algorithms [5, 12, 13, 20, 39, 72] have been proposed to process XML queries. To do a fair comparison of different labeling schemes, in the

implementation, except the part which must be different, we use the same join method to test the queries for all the labeling schemes. Figure 4.13 shows the response time (CPU time + I/O time) of the 6 queries in Table 4.3.

Figure 4.13(a) shows the response time of the containment, prefix and prime labeling schemes. Prime [74] has much larger response time because it has larger label size and it employs the modular and division operations to determine the ancestor-descendant, parent-child etc. relationships which are very expensive. We compare containment scheme and prefix scheme fairly. Note that it is unfair if prefix labels are stored as strings, but containment labels are stored as integers.

Figure 4.13(b) shows the response time of different containment schemes. Float-point-Containment [6] has much larger response time due to its very large label size. Our CDBS-Containment (“V-” and “F-”) has smaller response time than Binary-Containment (“V-” and “F-”) because our encodings can directly compare labels from left to right no matter the labels have variable lengths or fixed lengths, but V-Binary can not directly compare labels from left to right.

Finally Figure 4.13(c) shows the response time of different prefix schemes. BinaryString-Prefix [23] has larger response time due to its larger label size on D1. Though OrdPath1-Prefix and OrdPath2-Prefix have smaller label size than DeweyID(UTF8)-Prefix, their query performance is worse than DeweyID(UTF8)-Prefix because it is slow for them to decode the OrdPath1 and OrdPath2 codes and slow to separate the prefix levels (OrdPath2 even slower). CDBS(UTF8)-Prefix, CDBS(OrdPath1)-Prefix, CDBS(OrdPath2)-Prefix have the similar response time as DeweyID(UTF8)-Prefix, OrdPath1-Prefix and OrdPath2-Prefix respectively.

4.5.3 Performance Study on Intermittent Updates in Dynamic XML Data

Section 4.5.3.1 discusses how to process the leaf node updates. Section 4.5.3.2 is about the internal node updates. Section 4.5.3.3 describes the performance when a subtree is inserted into an XML tree.

4.5.3.1 Leaf Node Updates

The deletion of a leaf node will not require re-labeling of the existing nodes, therefore in this section we only compare the update performance when leaf nodes are inserted into XML.

Same as [74], we select one XML file *Hamlet* in dataset D1 to test the update performance (it is similar for other XML files). *Hamlet* has 5 *act* elements. We test the following *5 cases* (see Table 4.4 and Figure 4.14): inserting an *act* element before *act[1]*, inserting an act element before *act[2]*, ⋯, and inserting an act element before *act[5]*.

Table 4.4 shows the number of nodes to re-label when applying different labeling schemes. V-Binary-Containment and F-Binary-Containment need to re-label many nodes (Hamlet has totally 6636 nodes) in the 5 cases. Though V-Binary-Containment and F-Binary-Containment are very compact, they need to re-label the existing nodes when a node is inserted into XML.

Also BinaryString-Prefix and DeweyID(UTF8)-Prefix need to re-label many nodes in the five insertion cases. It should be noted that V-Binary-Containment and F-Binary-Containment have one more node than BinaryString-Prefix and DeweyID(UTF8)-Prefix to re-label because *act* elements are the children of the root

and the containment schemes need to re-label the root also (modify the "end" value of the root).

Table 4.4: Number of nodes to re-label in leaf node updates

Labeling schemes	Number of nodes to re-label (5 cases)				
	1	**2**	**3**	**4**	**5**
Float-point-Containment	0	0	0	0	0
V-Binary-Containment	6596	5121	3932	2431	1300
F-Binary-Containment	6596	5121	3932	2431	1300
V-CDBS-Containment	0	0	0	0	0
F-CDBS-Containment	0	0	0	0	0
BinaryString-Prefix	6595	5120	3931	2430	1299
DeweyID(UTF8)-Prefix	6595	5120	3931	2430	1299
OrdPath1-Prefix	0	0	0	0	0
OrdPath2-Prefix	0	0	0	0	0
CDBS(UTF8)-Prefix	0	0	0	0	0
CDBS(OrdPath1)-Prefix	0	0	0	0	0
CDBS(OrdPath2)-Prefix	0	0	0	0	0
Prime	1320	1025	787	487	261

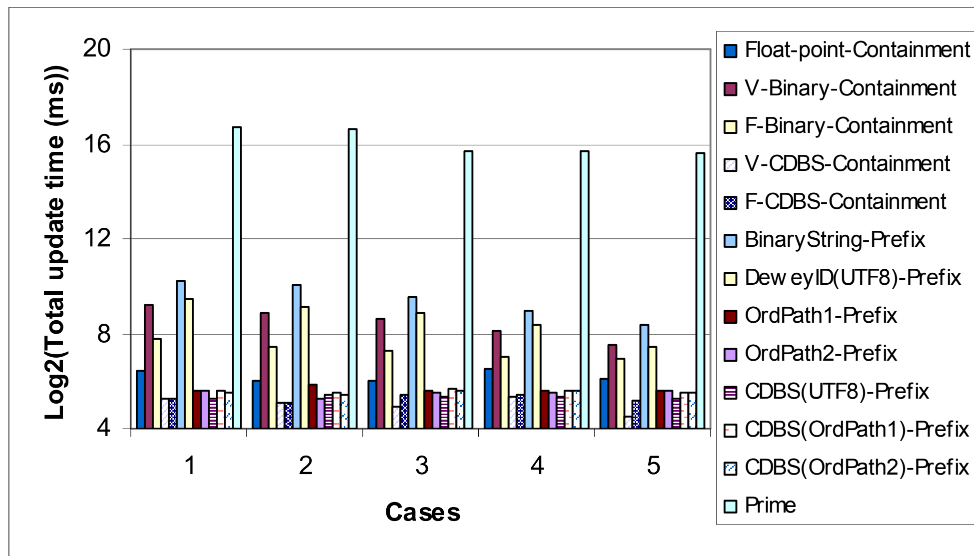


Figure 4.14: Log2 of total time (CPU time + I/O time) for leaf node updates

For Prime, the number of SC values that are required to re-calculate is counted in Table 4.4. Because Prime uses each SC value for every five nodes [74], the number of SC values required to re-calculate is 1/5 of the number of nodes required by V-Binary-Containment and F-Binary-Containment to re-label. Note that it is impossible to use a single SC value for all the nodes in the XML tree since the SC value will be too large a number.

In the five cases, Float-point-Containment (less than 18 nodes at a single place), V-CDBS-Containment (without overflow; see Example 5.1 of Chapter 5 for the overflow problem), F-CDBS-Containment (without overflow), OrdPath1-Prefix (without overflow), OrdPath2-Prefix (without overflow), CDBS(UTF8)-Prefix (without overflow), CDBS(OrdPath1)-Prefix (without overflow), and CDBS(OrdPath2)-Prefix (without overflow) need not re-label any existing nodes. Compared with V-Binary-Containment and F-Binary-Containment, V-CDBS-Containment and F-CDBS-Containment are also the most compact, yet they need *not* re-label the existing nodes in intermittent updates.

Next we study the total time (CPU time + I/O time) for updates. Figure 4.14 shows the ***LOG_2*** *of the total leaf node update time (ms) (Y-axis).* The total time required by *Prime* to re-calculate the SC values is much larger (at least 80 times; sum time of Case 1 to Case 5) than the time required by *Binary-Containment* ("V-" and "F-") to re-label the nodes. Prime theoretically is a good scheme to process updates, but it is not practicable. The update time of BinaryString-Prefix [23] and DeweyID(UTF8) [70] is larger than the update time of *Binary-Containment* ("V-" and "F-"). In contrast, the total update time of V-CDBS-Containment, F-CDBS-

Containment, CDBS(UTF8)-Prefix, CDBS(OrdPath1)-Prefix, and CDBS(OrdPath2)-Prefix is 1/12 to 1/3 of the time of *Binary-Containment*. This is because these approaches need not re-label the existing nodes.

It can be seen from Figure 4.14 that the update performance differences among Float-point, OrdPath and our approach are not very large though our approach is still ***better***. This is because only several nodes are inserted into the XML tree and the main part of the update time of Float-point, OrdPath and our approach is the I/O time. When considering the CPU time only, our approach is much better than Float-point and OrdPath. Their wide update differences can be seen in Section 5.5.2 of Chapter 5 where frequent insertions are executed.

4.5.3.2 Internal Node Updates

No matter an internal node is inserted into or deleted from an XML tree, the nodes should be re-labeled before the labeling schemes can work correctly to answer queries. Table 4.5 shows the number of nodes to re-label when *inserting* a node *acts* as the parent of the five act nodes of the Hamlet file and when *deleting* this internal node *acts* from the Hamlet file.

It can be seen from Table 4.5 that all the labeling schemes except V-CDQS-P-Containment and F-CDBS-P-Containment need to re-label many nodes in internal node updates. Though V-CDQS-P-Containment and F-CDBS-P-Containment also need to re-label the child nodes of the inserted or deleted node, it need not re-label the other descendant nodes of the inserted or deleted node. It only needs to re-label 5 nodes which is much better than the other labeling schemes.

Table 4.5: Number of nodes to re-label for internal node updates

Labeling schemes	Number of nodes to re-label	
	Insertion	Deletion
Float-point-Containment	6595	6595
V-Binary-Containment	6596	6595
F-Binary-Containment	6596	6595
V-CDBS-Containment	6595	6595
F-CDBS-Containment	6595	6595
V-CDBS-P-Containment	5	5
F-CDBS-P-Containment	5	5
BinaryString-Prefix	6595	6595
DeweyID(UTF8)-Prefix	6595	6595
OrdPath1-Prefix	6595	6595
OrdPath2-Prefix	6595	6595
CDBS(UTF8)-Prefix	6595	6595
CDBS(OrdPath1)-Prefix	6595	6595
CDBS(OrdPath2)-Prefix	6595	6595
Prime	6595	6595

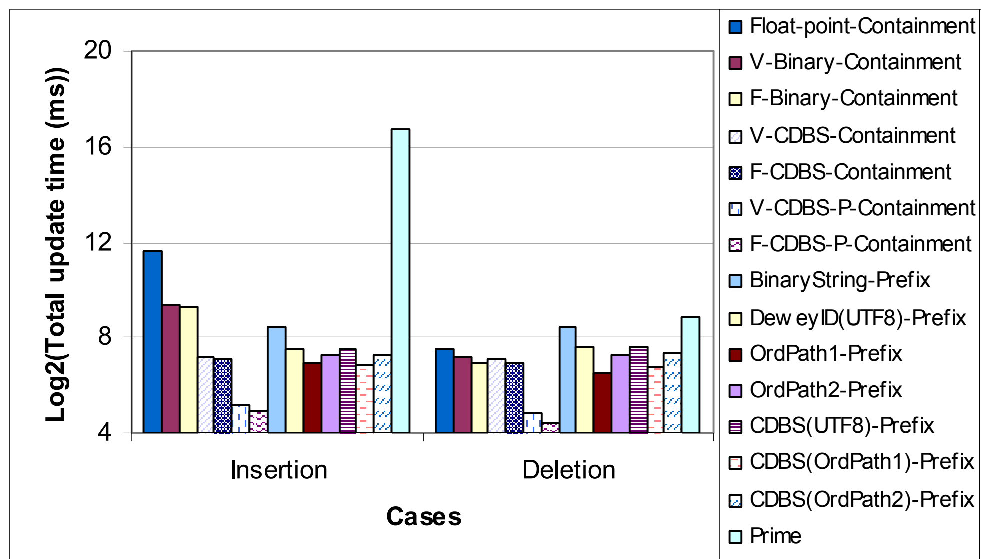


Figure 4.15: Log2 of total time (CPU time + I/O time) for internal node updates

Figure 4.15 shows the ***LOG*$_2$ *of the total internal node update time (ms) (Y-axis)***. For V-Binary-Containment and F-Binary-Containment, the deletion of an

internal node needs less update time than the insertion of an internal node, because the deletion only needs to modify the "level" values, but the insertion needs to modify the "start", "end" and "level" values.

V-CDBS-Containment and F-CDBS-Containment only need to modify the "level" values, but need not modify the "start" and "end" values even in *insertions*, therefore their insertion time is smaller. The update time of Float-point-Containment is larger because its label size is larger which needs more I/O time. ***In contrast, V-CDBS-P-Containment and F-CDBS-P-Containment need much less update time because they need to re-label much less nodes (5 vs 6595 or 6596).***

All the prefix labeling schemes including CDBS encoding based prefix labeling schemes need to re-label all the descendant nodes when an internal node is inserted or deleted.

When an internal node is updated, Prime needs to re-label all the descendant nodes of the inserted node. When an internal node is inserted, all the labels of the descendant nodes should multiply the label of the inserted node (see Example 4.14 and Figure 4.9). When an internal node is deleted, all the labels of the descendant nodes should divide the label of the deleted node. In addition, Prime needs to re-calculate the SC values to maintain the document order in insertions. Therefore the insertion time of Prime is much larger which can be seen from Figure 4.15.

4.5.3.3 Subtree Updates

In this section, we discuss how to insert a subtree. If we insert the nodes of the subtree one by one, the label size will increase fast. If we insert the nodes of the subtree based on the method introduced in Section 4.4.3, the label size increases slowly.

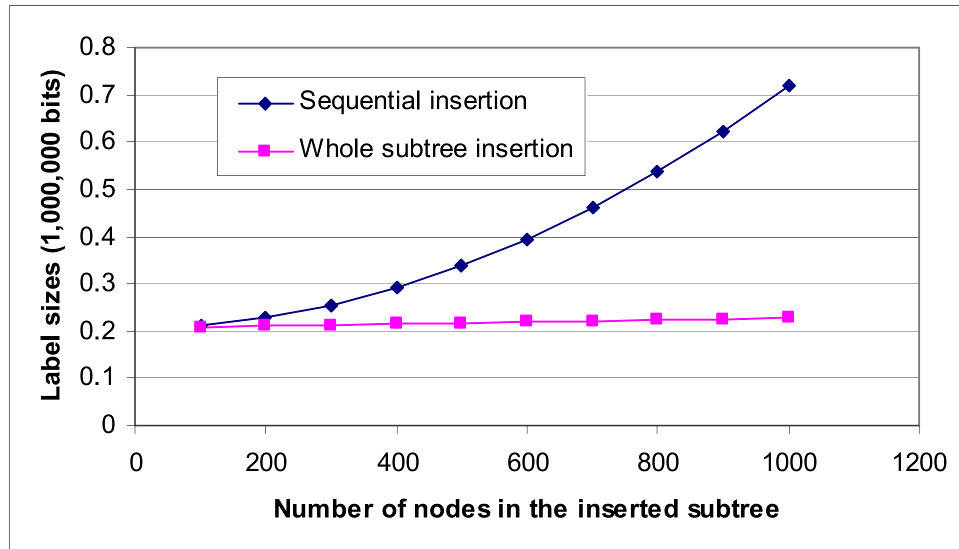


Figure 4.16: Label size increasing speed when inserting subtrees

Figure 4.16 shows the label size increasing speed of these two methods when inserting subtrees with different number of nodes. It can be seen from Figure 4.16 that the label size based on the method introduced in Section 4.4.3 increases much slower than the method of insertions of subtrees node by node.

4.5.4 Summary of Experimental Results

Because skewed frequent updates are easy to lead to re-labeling, we propose another Quaternary String encoding approach in Chapter 5 which can completely avoid re-labeling. We will compare the frequent update performance of different approaches in Section 5.5.2 of Chapter 5.

If XML is static, CDBS encodings work quite well in considering either the storage or the query performance. CDBS encoding is as compact as the binary number encoding of consecutive numbers.

Towards the intermittent updates, CDBS based labeling schemes need not re-label the existing nodes when a leaf node is inserted into an XML tree. CDBS-P-Containment can process the internal node updates much more efficiently than other labeling schemes since it only needs to modify the labels of the children of the inserted or deleted internal node rather than all the descendants. When a subtree is inserted into XML, the experimental result shows that the method introduced in Section 4.4.3 will make the label size increase slowly.

4.6 Summary

In this chapter, we firstly illustrate that the most important feature of our approach is that we compare codes (labels) based on the lexicographical order. Based on the lexicographical order, we propose Algorithm 4.1 which can always insert a binary string between two lexicographically ordered binary strings ended with "1". Algorithm 4.1 is the foundation of this thesis which can help to process XML updates efficiently.

Furthermore, we describe CDBS encoding. CDBS is as *compact* as the binary number encoding of *consecutive* numbers; there are no gaps between any two consecutive numbers, therefore CDBS is the most compact. In addition, based on Algorithm 4.1, CDBS supports order-sensitive insertions between any two *consecutive* CDBS codes *without re-encoding* the existing numbers.

We show that CDBS encoding is *orthogonal* to specific labeling schemes, thus it can be applied *broadly* to different labeling schemes. When CDBS is applied to

different labeling schemes, it will not increase the label size and will not decrease the query performance, and it supports updates efficiently.

V-CDBS encoding can efficiently process the leaf node updates. We need not re-label the existing nodes when a leaf node is inserted into an XML tree.

To efficiently process the internal node updates, we apply V-CDBS and F-CDBS encodings to P-Containment scheme introduced in Chapter 3. Based on V-CDBS-P-Containment and F-CDBS-P-Containment, we only need to modify the "parent_start" values of the children of the inserted or deleted internal node, but need not modify the "parent_start" values of the descendants of the children of the inserted or deleted internal node. This is cheaper than the existing containment, prefix and prime schemes since they need to re-label all the descendant nodes of the inserted or deleted internal node. Also it should be noted that only the P-Containment itself can not decrease the internal node update cost; the P-Containment scheme should be combined together with V-CDBS or F-CDBS encoding to efficiently process the internal node updates (see Theorem 4.7).

We also discuss how to make the label size increase slowly if a subtree is inserted into XML. It is an insertion of all the binary strings between the left and right binary strings but not one by one insertions.

Furthermore we discuss the uniform and skewed insertions. We will further discuss how to process the skewed insertion problem in Section 6.2 of Chapter 6.

Finally we conduct experiments which show that the methods proposed in this chapter can efficiently process different updates; meanwhile the encodings proposed in this chapter is very compact.

Chapter 5

CDQS Encoding of Node Labels to Completely Avoid Re-labeling

CDQS represents the Compact Dynamic *Quaternary* String encoding.

The CDBS encoding proposed in Chapter 4 still can *not* completely avoid re-labeling in XML updates. Here we use an example to show the reason.

Example 5.1 *The length of each V-CDBS code is stored with fixed length (e.g. 3; see Example 4.5). If many nodes are inserted into the XML tree, the size of the length field (e.g. 3) is not enough for the new labels, then we have to re-label all the existing nodes. Even if we increase the size of the length field to a larger number, it still can not completely avoid re-labeling, and it will waste the storage space. This is called the* ***overflow*** *problem in this thesis. Similarly F-CDBS (each code of F-CDBS is fixed length, therefore F-CDBS will encounter the overflow also) and OrdPath [64] will encounter the overflow problem also (O'Neil et al. do not mention this overflow problem in OrdPath [64]).*

To solve the overflow problem, we have the following observation. We observe that the size of V-CDBS is used only to separate different V-CDBS codes. After separation, we can directly compare the V-CDBS codes from left to right. Therefore to solve the overflow problem, the way is to find a ***separator*** which can

separate different V-CDBS codes; meanwhile this separator will not encounter the overflow problem. In binary string, there are only two symbols "0" and "1"; if we use "0" or "1" as the separator, only one symbol is left and CDBS will not be dynamic. Therefore we design a Compact Dynamic Quaternary String (CDQS) encoding which can help to completely avoid re-labeling in XML updates.

The rest of this chapter is organized as follows. In Section 5.1 we describe CDQS encoding. Section 5.2 depicts how to apply CDQS to different labeling schemes. Based on CDQS, we discuss how to completely avoid re-labeling in XML updates in Section 5.3. We report the experimental results in Section 5.5. Finally we summarize this chapter in Section 5.6.

5.1 The Compact Dynamic Quaternary String Encoding (CDQS) for Node Labels

Four symbols "0", "1", "2" and "3" are used in the quaternary string and each symbol is stored with two bits, i.e. "00", "01", "10" and "11".

Now we illustrate our **Compact Dynamic *Quaternary* String (CD*Q*S) code)**: *CDQS code is a special quaternary string; the "0" is used as the separator and only "1", "2" and "3" are used in the CDQS code itself.*

Because we use "0" as the separator, it is not appropriate to concatenate "0"s for the fixed length CDQS, i.e. F-CDQS. In this thesis, when we talk about ***CDQS***, it is ***equivalent to V-CDQS***.

Still based on the 18 numbers in Table 4.1, we use examples to show how CDQS works (see Table 5.1).

Table 5.1: CDQS encoding

Decimal number	CDQS
1	112
2	12
3	122
4	13
5	132
6	2
7	212
8	22
9	222
10	223
11	23
12	232
13	3
14	312
15	32
16	322
17	33
18	332
Total size (bits)	88

Step 1: In the encoding of the 18 numbers based on CD*Q*S, we suppose there is one more number before number 1, say number **0**, and one more number after number 18, say number **19**.

Step 2: The **(1/3)th** number is encoded with "***2***", and the **(2/3)th** number is encoded with "***3***". The (1/3)th number is number **6**, which is calculated in this way, 6 = round(0+(19–0)/3). The (2/3)th number is number **13** (13 = round(0+(19–0)×2/3)). It can be seen from Table 5.1 that the CDQS code for number 6 is "2" and the CDQS code for number 13 is "3".

Step 3: The (1/3)th and (2/3)th numbers between number 0 and number 6 are number 2 (2 = round(0+(6–0)/3)) and number 4 (4 = round(0+(6–0)×2/3)). The CDQS code of number 0 (S_L) is now empty with size 0 bit and the CDQS code of number 6 (S_R) is

now “2” with size 2 bits. This is **Case (b) where size(S_L) < size(S_R)**. In this case, the $(1/3)^{th}$ code is that we change the last symbol “2” of S_R to “12”, i.e. the code of number **2** is “***12***” (“2” → “12”), and the $(2/3)^{th}$ code is that we change the last symbol “2” of S_R to “13”, i.e. the code of number **4** is “***13***” (“2” → “13”). Note that in the initial encoding, if *size(S_L) < size(S_R)*, S_R can only be ended with “2” (can not be ended with “3”).

Step 4: The $(1/3)^{th}$ and $(2/3)^{th}$ numbers between numbers 6 and 13 are numbers 8 (8 = round(6+(13–6)/3)) and 11 (9 = round(6+(13–6)×2/3)). The CDQS code of number 6 (S_L) is “2” with size 2 bits and the code of number 13 (S_R) is “3” with size 2 bits. This is **Case (a) where size(S_L) ≥ size(S_R)**. In this case, the $(1/3)^{th}$ code is that we directly concatenate one more “2” after the S_L, i.e. the code of number **8** is “***22***” (“2” ⊕ “2”→“22”), and the $(2/3)^{th}$ code is that we directly concatenate one more “3” after the S_L, i.e. the code of number **11** is “***23***” (“2” ⊕ “3” → “23”).

Step 5: The $(1/3)^{th}$ and $(2/3)^{th}$ numbers between numbers 13 and 19 are numbers 15 (15 = round(13+(19–13)/3)) and 17 (17 = round(13+(19–13)×2/3)). The code of number 13 (S_L) is “3” with size 2 bits and the code of number 19 (S_R) is empty now with size 0 bit. This is still **Case (a)**. Therefore the CDQS code of number **15** is “***32***” (“3” ⊕ “2” → “32”), and the code of number **17** is “***33***” (“3” ⊕ “3” → “33”).

In this way, all the numbers will be encoded with CDQS codes. Finally we need to discard the codes for numbers 0 and 19 since they do not exist actually. It should be noted that if the $(2/3)^{th}$ number exactly refers to the $(1/3)^{th}$ number, the code

for the $(2/3)^{th}$ number will not appear since this number has already been encoded with the $(1/3)^{th}$ code. Table 5.1 shows the CDQS codes for all the 18 numbers.

5.1.1 CDQS Encoding Algorithm

The formal algorithms of CDQS (Algorithms 5.1 and 5.2) are similar to the V-CDBS algorithms (Algorithms 4.1 and 4.2). The difference is that CDQS is based on the $\mathbf{(1/3)^{th}}$ and $\mathbf{(2/3)^{th}}$ positions rather the $(1/2)^{th}$ position in V-CDBS. The above Step 1 to Step 5 are illustrations of the formal algorithms (Algorithms 5.1 and 5.2) for CDQS.

When we note that the quaternary strings "0" $\prec$ "1" $\prec$ "2" $\prec$ "3" lexicographically, we have the following theorem 5.2.

Lemma 5.1 *All the CDQS codes are ended with either "2" or "3".*

Proof: "1" can not appear at the end of a CDQS code (see Algorithms 5.1 and 5.2, or see Step 1 to Step 5), thus Lemma 5.1 holds.

Theorem 5.2 *All the CDQS codes are lexicographically ordered.*

Proof: The CDQS algorithm guarantee that the $(1/3)^{th}$ and $(2/3)^{th}$ CDQS codes are lexicographically ordered between S_L and S_R. By recursively applying the encoding of the $(1/3)^{th}$ and $(2/3)^{th}$ CDQS codes, the global lexicographical order of all the CDQS codes are maintained. Therefore Theorem 5.2 holds.

Example 5.2 *The CDQS codes in Table 5.1 are lexicographically ordered from top to bottom, e.g. "112" $\prec$ "12"* ***lexicographically*** *since the second symbol of "112" is "1" while the second symbol of "12" is "2".*

Algorithm 5.1: AssignOneThirdAndTwoThirdCodes(S_L, S_R)

Input: $S_L \prec S_R$; S_L and S_R are ***ended with either "2" or "3"***
Output: S_{M1} and S_{M2} (ended with 1)such that $S_L \prec S_{M1} \prec S_{M2} \prec S_R$ lexicographically. S_{M1} is the quaternary string at the $(1/3)^{th}$ position, and S_{M2} is the quaternary string at the $(2/3)^{th}$ position.

Description:

1: **if** S_L and S_R are both empty **then**
2: S_{M1} = "2"
3: S_{M2} = "3"
4: **else**
5: **if** size(S_L) $\geq$ size(S_R) **then**
6: $S_{M1} = S_L \oplus$ "2"
7: $S_{M2} = S_L \oplus$ "3"
8: **else if** size(S_L) $<$ size(Right_Code) **then**
9: Temp_Code = S_R with the last symbol changed to "1"
10: S_{M1} = Temp_Code $\oplus$ "2"
11: S_{M2} = Temp_Code $\oplus$ "3"

Algorithm 5.2: CDQS Encoding(*TN*)

Input: A positive integer *TN*
Output: The *CDQS codes* for numbers 1 to *TN*

Description:

1: suppose there is one more number before the first number, called number **0**, and one more number after the last number, called number **(*TN*+1)**
2: Define an array *codeArr[0,TN+1]* //the size of codeArr is //TN+2; each element of the codeArr is empty at the beginning
3: **CDQS_SubEncoding**(*codeArr*, 1, *TN*)
4: **discard** the 0th and $(TN+1)^{th}$ elements of the *codeArr*

Procedure **CDQS_SubEncoding** (*codeArr*, P_L, P_R)
/*CDQS_SubEncoding is a recursive procedure; *codeArr* is an array, P_L is the left position, and P_R is the right position*/

1: $P_{M1} = P_L$+round((P_R-P_L)/3) (P_{M1} is the $(1/3)^{th}$ position)
2: P_{M2} = P_L+round((P_R-P_L)$\times$2/3) (P_{M2} is the $(2/3)^{th}$ position)
3: **if** $P_L \neq P_R$ **then**
4: AssignOneThirdAndTwoThirdCodes(*codeArr*[P_L], *codeArr*[P_R])
5: **if** $P_{M1} \neq P_L$ and $P_{M1} \neq P_R$ **then**
6: *codeArr*[P_{M1}]= S_{M1} //returned by line 4 in CDQS_SubEncoding
7: **if** $P_{M2} \neq P_{M1}$ and $P_{M2} \neq P_R$ **then**
8: *codeArr*[P_{M2}]= S_{M2} //returned by line 4 in CDQS_SubEncoding
9: **if** ($P_{M1} \neq P_L$ and $P_{M1} \neq P_R$) or ($P_{M2} \neq P_L$ and $P_{M2} \neq P_R$) **then**
10: CDQS_SubEncoding(*codeArr*, P_L, P_{M1})
11: CDQS_SubEncoding(*codeArr*, P_{M1}, P_{M2})
12: CDQS_SubEncoding(*codeArr*, P_{M2}, P_R)

CDQS is the most compact encoding with three symbols. When there are only 2 symbols "0" and "1", we know that V-CDBS is the most compact from Theorem 4.6. When we use three symbols "1", "2" and "3", the $(1/3)^{th}$ and $(2/3)^{th}$ positions can guarantee that CDQS is the most compact encoding with 3 symbols. Note that the symbol "0" is used as the separator.

Example 5.3 *It can be seen from Table 5.1 that the total size of CDQS is 88 bits, also we need to count the size of the separators (the separator "0" is stored with size 2 bits). Therefore the size of CDQS is* ***2****×18+88=124 bits. Compared with the total size 118 bits of V-CDBS (see Example 4.5 in Chapter 4), the total size of CDQS is a little larger. However, based on CDQS we can completely avoid re-labeling in XML updates.*

5.1.2 Size Analysis

Below is the size analysis of CDQS.

CDQS CDQS has two numbers 6 and 13 stored with size 1×2 bits, 6 numbers 2, 4, 8, 11, 15 and 17 stored with size 2×2 bits, ···, therefore the total size of CDQS is:

$$(2\times 3^0)\times(1\times 2)+(2\times 3^1)\times(2\times 2)+(2\times 3^2)\times(3\times 2)+\cdots+(2\times 3^n)\times((n+1)\times 2)\ \text{(bits)}$$

$$=(2n+1)\times 3^{n+1}+1 \qquad (5.1)$$

(see Appendix C2 for how to get formula (5.1))

Suppose the total number is *N*, which should be equal to $(2\times 3^0)+(2\times 3^1)+\cdots+(2\times 3^n)=3^{n+1}-1$. Thus formula (5.1) becomes to

$$2N\log_3(N+1)-N+2\log_3(N+1) \quad (5.2)$$

When taking the separator ("0") size $N\times 2=2N$ into account, the total size of CDQS is:

$$2N\log_3(N+1)+N+2\log_3(N+1) \quad (5.3)$$

Compared with the total size $N\log(N+1)+N\log(\log(N))-N+\log(N+1)$ of V-CDBS shown in Formula (4.3), the total size of CDQS is larger. When N=2, the size of CDQS is 2.90 times of that of V-CDBS; when $N\in[3,49]$, the multiples are 1.14 to 1.87; when $N\in[50,100000000]$, the multiples are between 1.10 and 1.14. Thus the size of CDQS is a little larger than the size of V-CDBS. However, CDQS can completely avoid re-labeling (see Section 5.3 of this chapter).

5.2 Applying CDQS to Different Labeling Schemes

We can apply CDQS to different labeling schemes. For the containment scheme, since the "level" value will encounter the *overflow* problem, we only discuss how to apply CDQS to the P-Containment scheme (see Section 3.1 of Chapter 3 for the P-Containment scheme). When replacing the decimal numbers 1-18 of the "start", "end" and "parent_start" values of the P-Containment scheme in Figure 3.1(b) with CDQS codes in Table 5.1, a CDQS-P-Containment scheme is formed. Based on the separator

"0", we can separate the "start", "end" and "parent_start" values, and every three values form a group of "start, end, parent_start".

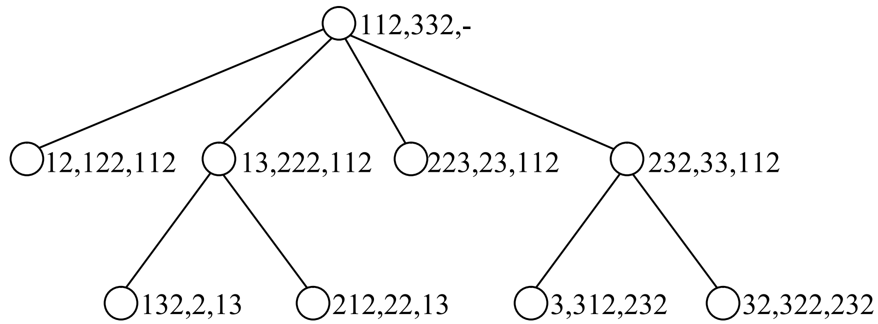


Figure 5.1: CDQS-P-Containment scheme

Example 5.4 *Figure 5.1 shows CDQS-P-Containment scheme. For the labels "112,332,-", "12,122,112" and "13,222,112" of the first three nodes of the CDQS-P-Containment scheme shown in Figure 5.1, we store them consecutively in the hard disk as "112033201201220112013022201120". Based on the separator "0", we can separate them as "112", "332", "12", "122", "112", "13", "222" and "112", the first two are a group of "start, end" which is the label of the root. It should be noted that the root does not have the "parent_start" value. The next three are a group of "start, end, parent_start" which is the label of the next node after the root. The rest three are another group of "start, end, parent_start" which is the label of the third node. The labels for the 4th, 5th, etc. nodes can be similarly stored after the first three labels. Different from the V-CDBS codes which use the lengths to separate the "start", "end" and "parent_start", CDQS uses the separator "0" to separate the "start", "end" and "parent_start" which will never encounter the overflow problem.*

In this way, we can completely avoid re-labeling in XML updates. Note that in the implementation, each quaternary number is stored with two bits e.g. "2" is stored as "10" (two bits).

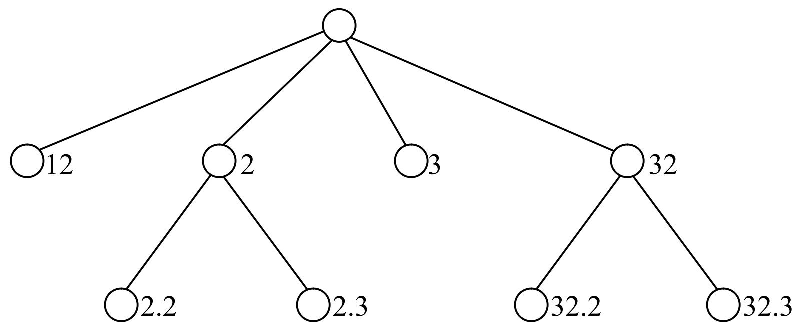


Figure 5.2: CDQS-Prefix scheme

Example 5.5 *Figure 5.2 shows that we apply CDQS to the prefix scheme. The root has 4 children. To encode 4 numbers based on CDQS, the codes will be "12", "2", "3" and "32". Similarly if there are two siblings, their self_labels are "2" and "3".*

For the prefix scheme, the delimiter "." can not be stored together with the numbers in the implementation to separate different components. We can use the UTF8 [78] encoding or OrdPath encoding [64] to process the delimiters for the V-CDBS encoding.

For CDQS encoding, we use the following approach to process the delimiters. We use one *separator* "0" as the ***delimiter*** to *separate different components of a label* (*e.g.* separate "2" and "3" in "2.3"; the *separator* "0" is equivalent to the "." in Figure

5.2), and use *two consecutive separator* "00" as the ***separator*** to *separate different labels* (*e.g.* separate "2.2" and "2.3").

Example 5.6 *To store the first three labels "12", "2" and "2.2" in Figure 5.2 (except the root which is empty), they are stored as "120020020200" in the hard disk. Based on the separator "00", we can separate the three labels "12", "2" and "202", and if necessary, we can separate different components of a label, e.g. separate "2" and "2" in "202" based on the delimiter "0".*

It may be asked why we choose "0" but rather than any other number "1", "2" or "3" as the delimiter? It is because in this way, we can directly compare two labels symbol by symbol from left to right to determine the document order. See the following example for more details.

Example 5.7 *Suppose that there is one more* ***sibling*** *node inserted between "2" and "3" in Figure 5.2. Based on Algorithm 5.3 (****in the next section; Section 5.3****), the label of the inserted node is "22". We know that "2.3" is before "22" (the label of the inserted node) in document order. "2.3" is stored as "203" with delimiter "0". We can directly compare "203" and "22" from left to right to get the relative orders of these two labels. If we use any number of "1", "2" or "3" as the delimiter, we can not directly compare the labels from left to right to get the document order.*

5.3 Completely Avoiding Re-labeling in XML Updates

Algorithm 5.3 shows how to insert a quaternary string between two CDQS codes (two quaternary strings). Algorithm 5.3 considers the case that there are only insertions which is similar to Algorithm 4.1. If there are only insertions and size(S_L) < size(S_R), then S_R can only be ended with "2". We use examples to show how Algorithm 5.3 works.

Algorithm 5.3: AssignInsertedQuaternaryString(S_L, S_R)

Input: $S_L \prec S_R$; S_L and S_R are ***ended with either "2" or "3"***
Output S_M such that $S_L \prec S_M \prec S_R$ lexicographically

Description:

```
 1: if size(S_L) > size(S_R) then
 2:    if the last symbol of S_L is "2" then
 3:       S_M = S_L with the last symbol changed from "2" to "3"
 4:    else if the last symbol of S_L is "3" then
 5:       S_M = S_L ⊕ "2"    //⊕ means concatenation
 6:    end if
 7: else if size(S_L) == size(S_R) then
 8:       S_M = S_L ⊕ "2"
 9: else if size(S_L) < size(S_R) then
10:       S_M = S_R with the last symbol "2" changed to "12"
11: end if
12: return S_M
```

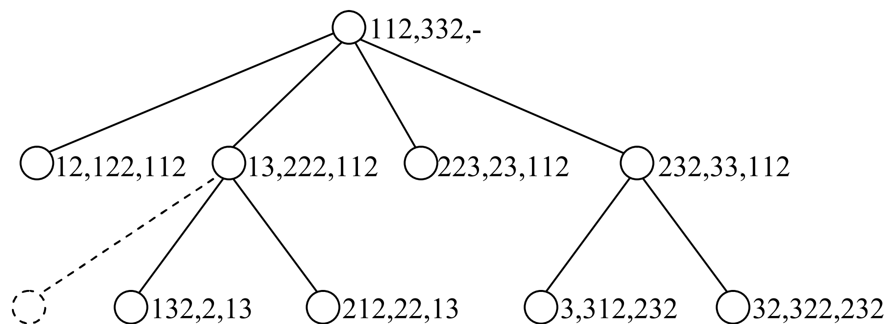


Figure 5.3: Insertions based on CDQS-P-Containment scheme

Example 5.8 *If we want to insert a sibling before "132,2,13" in Figure 5.3, the "start" and "end" of this inserted node should be lexicographically between the "start" of "13,222,112" and the "start" of "132,2,13", i.e. between "13" and "132". Based on Algorithm 5.3, we insert a quaternary string between "13" and "132", then the "start" value of the inserted node is "**1312**" (see lines 9-10 of Algorithm 5.3). The "end" value of the inserted node is an insertion between the new "start" value "1312" and "132" (the "start" of "132,2,13"). The "end" value of the inserted node will be "**1313**" (see lines 1-3 of Algorithm 5.3). Obviously, "13"$\prec$ "**1312**" $\prec$ "**1313**" $\prec$ "132" lexicographically. The "parent_start" value of the inserted node is "13" which is the "start" of its parent. CDQS will never encounter the overflow problem, therefore we need not re-label any existing nodes no matter how many nodes are inserted, but we can keep the containment scheme work correctly.*

Example 5.9 *Similarly if we want to insert a sibling node before "202" in Figure 5.4 ("202" is equivalent to the "2.2" in Figure 5.2), the self_label of the inserted node is "12" (see lines 9-10 in Algorithm 5.3; note that S_L is empty); the complete label of the inserted node is "2012". CDQS will never encounter the overflow problem, therefore we need not re-label any existing nodes based on the CDQS-Prefix scheme when nodes are inserted into an XML tree.*

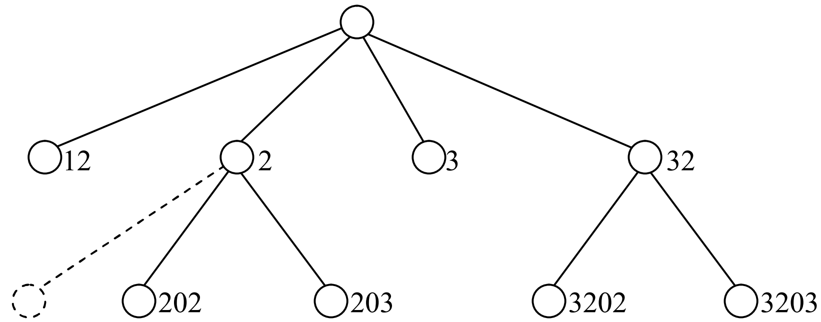


Figure 5.4: Insertions based on CDQS-Prefix scheme

Theorem 5.3 *Algorithm 5.3 guarantees that a quaternary string can be inserted between two consecutive CDQS codes with the orders kept and without re-encoding any existing numbers.*

Proof: When we check Algorithm 5.3, all the conditions can guarantee that $S_L \prec S_M \prec S_R$ lexicographically, therefore Theorem 5.3 holds.

Corollary 5.4 *Algorithm 5.3 guarantees that infinite number of quaternary strings can be inserted between any two consecutive CDQS codes.*

Proof: When recursively using Algorithm 5.3 for the insertions, Corollary 5.4 holds.

Theorem 5.5 *CDQS can* ***completely*** *avoid the re-encoding of the existing numbers.*

Proof: We use “0” as the *separator* to separate different CDQS codes, and “0” will never encounter the overflow problem. Also Corollary 5.4 guarantees that infinite number of quaternary strings can be inserted between any two consecutive CDQS codes. Therefore Theorem 5.5 holds.

Section 4.4.2 shows that we can efficiently process the internal node updates though we can not completely avoid re-labeling in internal node updates; this is the drawback of the existing labeling schemes, but not the drawback of CDQS encoding.

5.4 Extensions of CDQS

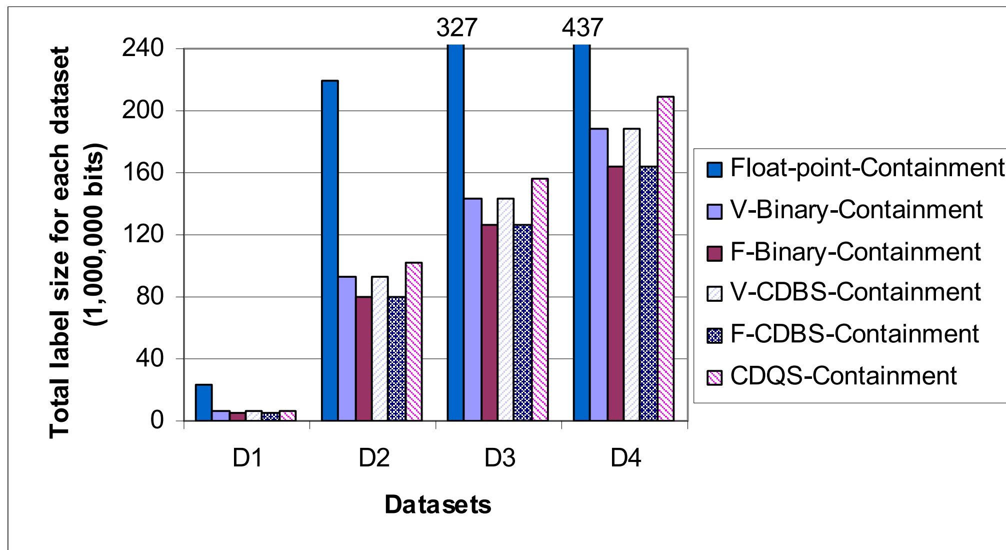
By further extending CDQS, we can use octal and hex string encodings to process updates, called CDOS and CDHS respectively. It can be seen from previous sections that CDQS waste 1/4 of numbers for the separator. If we use CDOS and CDHS encodings, only 1/8 and 1/16 of the total numbers are wasted. Thus the CDOS and CDHS encodings will be more compact when the total number is large. On the other hand, the separator sizes of CDOS and CDHS encodings are 3 bits and 4 bits respectively which makes CDOS and CDHS not as compact as expected. See Section 5.5.3 for the experimental results and more details about CDOS and CDHS.

5.5 Experimental Evaluation and Comparisons

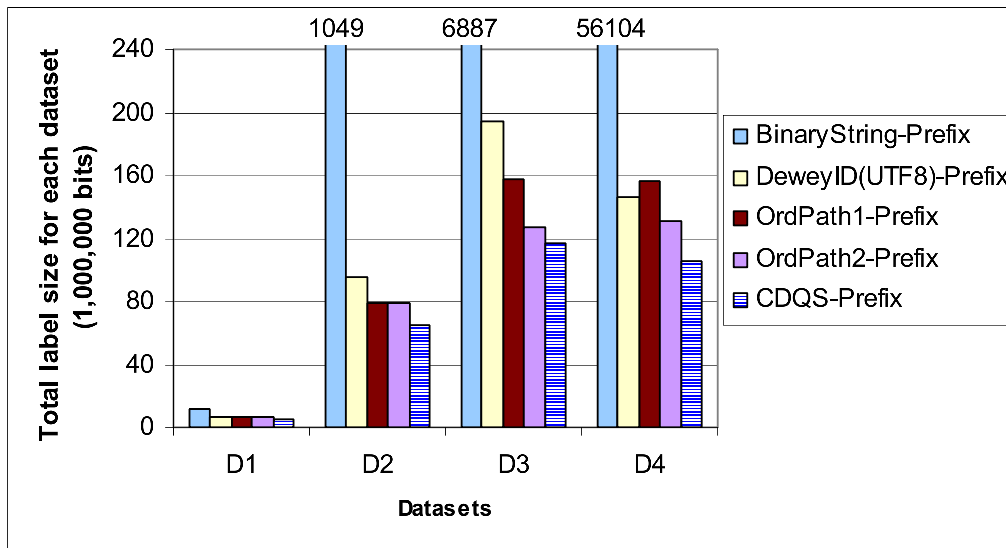
5.5.1 Performance Study on Static XML Data

We firstly discuss the label size. Figure 5.5(a) shows that CDQS encoding is applied to the containment scheme. The label size of CDQS-Containment (equivalent to V-CDQS-Containment; see the third paragraph of Section 5.1 of this chapter for more details) is a little larger (10% around) than the label size of V-CDBS-Containment because the separator “0” can not appear in the CDQS code itself which is a waste (see the formal size analysis in Section 5.1.2 of this chapter also). Though the label

size of CDQS-Containment is a little larger than the label size of V-CDBS-Containment, CDQS-Containment can completely avoid re-labeling in XML updates.

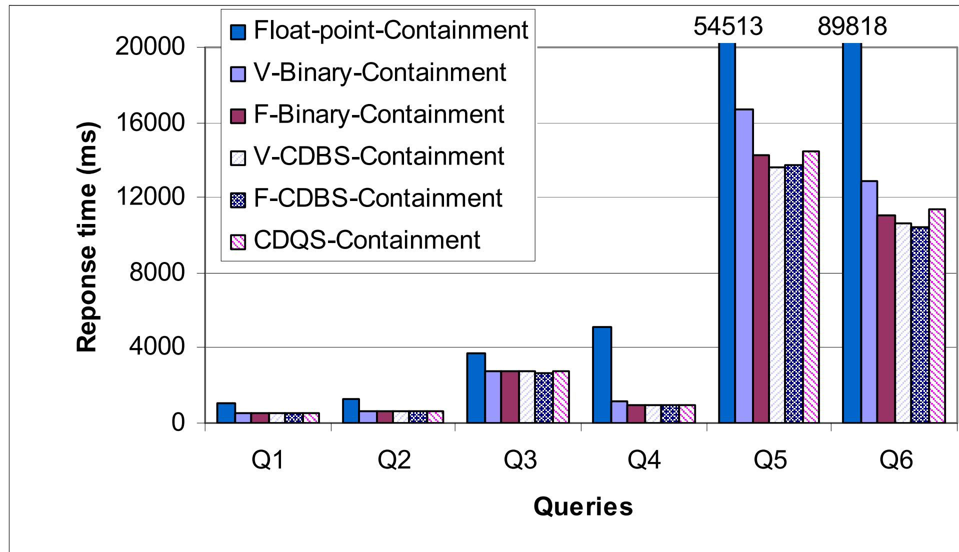


(a) Label sizes of containment schemes

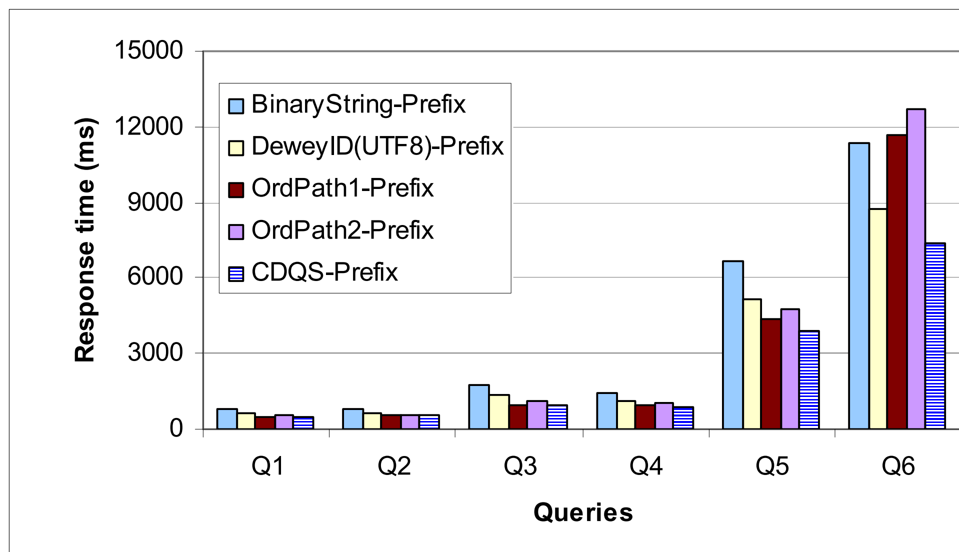


(b) Label sizes of prefix schemes

Figure 5.5: Label sizes of different labeling schemes



(a) Response time of queries based on containment schemes



(b) Response time of queries based on prefix schemes

Figure 5.6: Response time of different queries based on different labeling schemes

Moreover, from Figure 5.5(b), we can see that CDQS-Prefix has the smallest label sizes in all the four datasets (D1-D4). CDQS-Prefix is the most compact

compared to the existing *prefix* labeling schemes, and CDQS-Prefix can completely avoid re-labeling in XML updates (except internal node updates). Note that we separate different labels of DeweyID(UTF8) and OrdPath based on their label sizes.

In addition, Figure 5.6(a) shows that the response time of CD*Q*S-Containment is a little larger than the response time of V-CDBS-Containment, and Figure 5.6(b) shows that CD*Q*S-Prefix has smaller response time on different queries since it has the smaller label size.

5.5.2 Performance Study on Frequent Updates in Dynamic XML Data

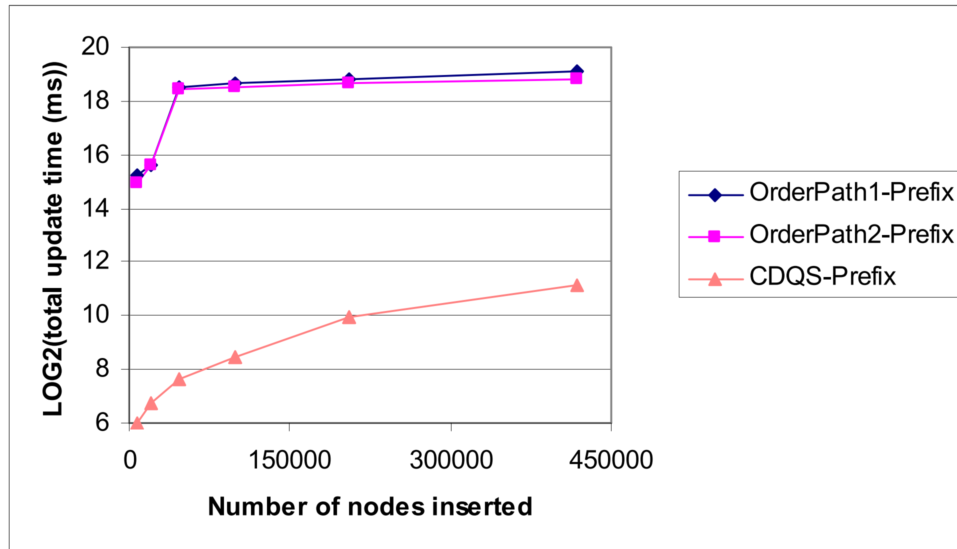
When intermittent nodes are inserted into XML, V-Binary-Containment, F-Binary-Containment, BinaryString-Prefix, DeweyID(UTF8)-Prefix and Prime have much larger update time, thus it will be a disaster for them to update XML with frequent and tiny insertions, which makes them impossible to answer any queries in either the uniformly frequent or skewed frequent insertion environment. In this section, we mainly compare the update performance between OrdPath-Prefix (OrdPath1-Prefix and OrdPath2-Prefix) and CDQS-Prefix, and between Float-point-Containment and CDQS-Containment. We compare CDQS with the existing labeling schemes because frequent updates are easy to lead to the overflow, and CDQS can completely avoid re-labeling in XML updates (CDQS will not encounter the overflow problem). Section 5.5.2.1 discusses the uniformly frequent insertions and Section 5.5.2.2 discusses the skewed frequent insertions.

5.5.2.1 Uniformly Frequent Updates

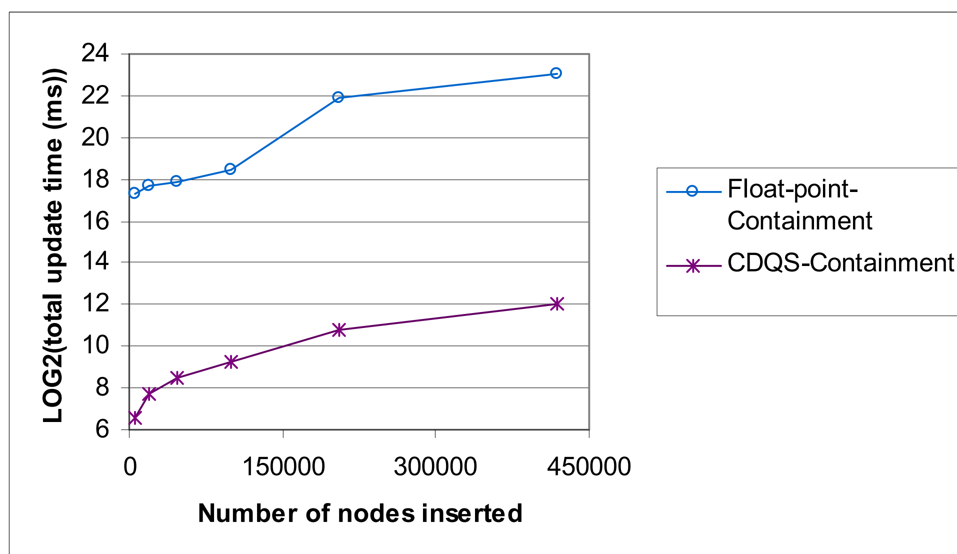
In this section, we test the uniformly distributed frequent insertions, i.e the insertions are randomly at different places of XML. The Hamlet file has totally 6636 nodes. We insert 6635 nodes between every two consecutive nodes of the 6636 nodes. Based on the new file after insertion, we insert another 13270 nodes between any two consecutive nodes. We repeat this kind of insertion 6 times. After the 6th time insertion, the node number in the XML data is 424641 which is 63.99 times of the original node number.

Figures 5.7(a) and 5.7(b) show the ***LOG*₂** *of the total update time (ms) (Y-axis)* of prefix schemes (OrdPath-Prefix [64] vs CDQS-Prefix) and containment schemes (Float-point-Containment [6] vs CDQS-Containment) respectively. In frequent updates, the main part of the total update time is the *CPU time* since we can read the file *at one time* and write back all the updates at different places to the hard disk *at one time*. Even in *frequent writing back*, our approach still can *save a lot of update time* because the label size of CDQS-Prefix is smaller than the label size of OrdPath-Prefix and the label size of CDQS-Containment is smaller than the label size of Float-point-Containment.

Even if the overflow is not encountered, i.e. without re-labeling, the update time of OrdPath-Prefix is still at least 207 ($2^{18.8-11.1} = 2^{7.7}$) times of that of CDQS-Prefix (see Figure *5.7(a)*). OrdPath needs to decode its codes [64] and needs the addition and division operations to get the numbers between two numbers which are both expensive. CDQS-Prefix only needs to modify the last *2* bits of the neighbor label to get the inserted label which is cheaper.



(a) OrdPath-Prefix (1&2) vs CDQS-Prefix



(b) Float-point-Containment vs CDQS-Containment

Figure 5.7: Uniformly frequent updates

Even if the overflow is not encountered (less than 18 nodes at a fixed place), i.e. without re-labeling, the update time of Float-point-Containment (need to insert two values "start" and "end"; the calculation is expensive) is still at least 548 ($2^{9.1}$) times of that of CDQS-Containment (see Figure *5.7(b)*).

When there is overflow, the update time of OrdPath-Prefix and Float-point-Containment is even larger.

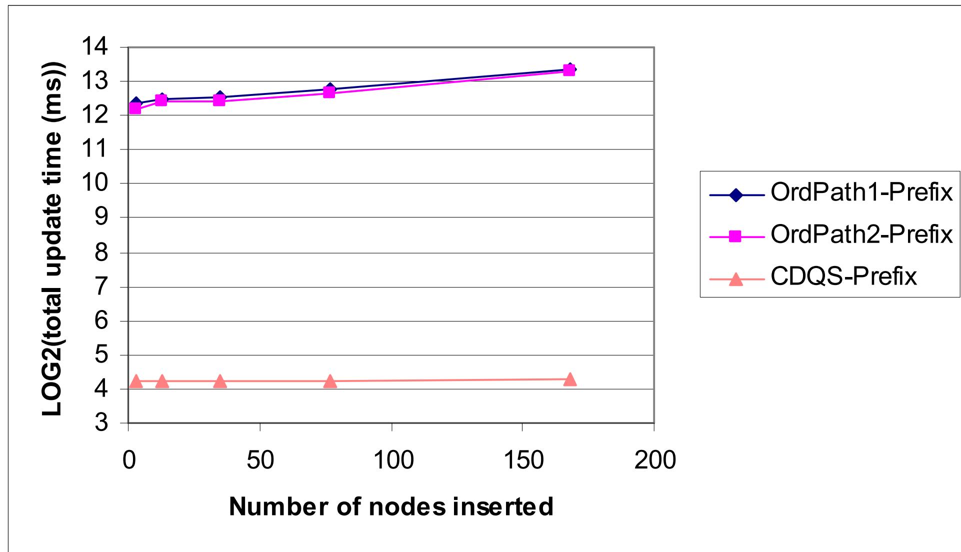
If we can increase the length field of V-CDBS code a little larger, the uniformly frequent updates will not be so easy to lead V-CDBS to re-labeling. In addition, because V-CDBS only needs to modify the last 1 bit of the neighbor label to get the inserted label, its update cost is smaller than the update cost of CDQS which needs to modify the last 2 bits of the neighbor label. Therefore V-CDBS can process the *uniformly* frequent updates more efficiently compared to CDQS if there is no overflow. Note that the update costs of OrdPath-Prefix and Float-point-Containment are much more expensive than V-CDBS and CDQS even if there is no overflow.

5.5.2.2 Skewed Frequent Updates

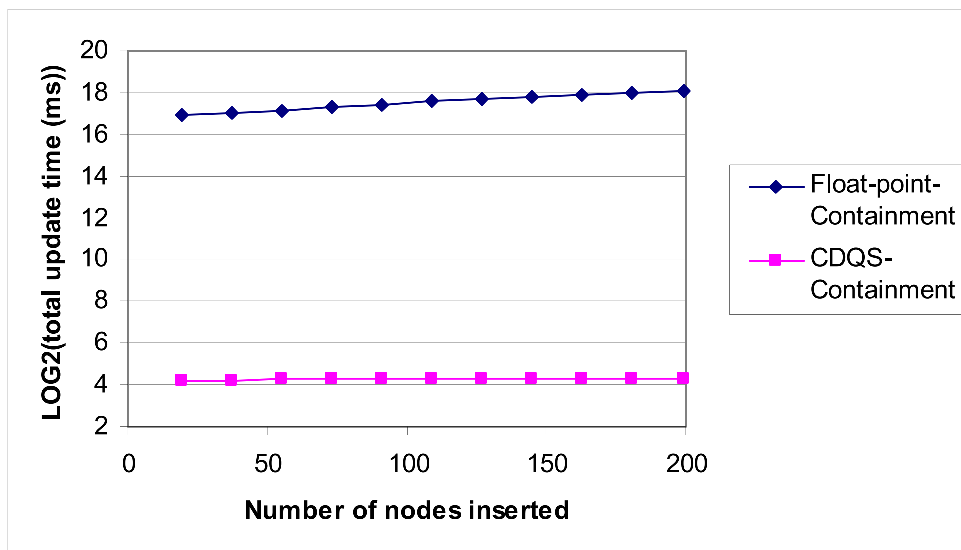
In this section, we test the case that the nodes are always inserted at a fixed place of the XML file Hamlet. The skewed insertion is easy to lead to the overflow, therefore V-CDBS is not appropriate to process the skewed insertion. In this section, we only compare CDQS encoding with the existing approaches.

When nodes are always inserted at a fixed place, it is much easier to lead OrdPath-Prefix and Float-point-Containment to re-labeling.

Figures 5.8(a) shows that the update time of OrdPath-Prefix is at least 1000 times of that of CDQS-Prefix, and the update time of Float-point-Containment is at least 2000 times of that of CDQS-Containment in skewed insertions. Thus CDQS is much better than OrdPath and Float-point in processing skewed frequent updates.



(a) OrdPath-Prefix vs CDQS-Prefix



(b) Float-point_Containment vs CDQS-Containment

Figure 5.8: Skewed frequent updates

The very large update time and the larger label sizes make OrdPath-Prefix and Float-point-Containment unsuitable to answer queries in the frequent (*uniformly and*

skewed) insertion environment. This analysis together with the analysis in the first paragraph of Section 5.5.2.2 indicate that CDQS will work the best to answer queries in the frequent insertion environment even if we do not use any techniques to process the *skewed* insertion problem. Even so, we still propose some techniques to process the skewed insertion in Section 6.3.2.

5.5.3 Performance Study on CDOS and CDHS

When the total number is between 2^0 and 2^{20}, Figure 5.9 shows the sizes of CDQS, CDOS, and CDHS. In Figure 5.9, we suppose that there is one separator for each code. When the total number is smaller than or equal to 2^8, CDQS is the most compact; when the total number is between 2^{10} and 2^{20}, CDOS is the most compact; and when the total number is larger than or equal to 2^{16}, CDHS has smaller size than CDQS.

Though with the increasing of total number, the total size of CDOS and CDHS will be smaller than CDQS, the encoding time of CDOS and CDHS is averagely 2.1 and 5.5 times of that of CDQS. That is to say, CDOS and CDHS are slower in encoding.

That also shows that CDOS and CDHS have more expensive update costs than CDQS. CDQS only needs to modify the last 2 bits of the neighbor codes, while CDOS and CDHS need to modify the last 3 and 4 bits respectively. More important, CDQS only needs to consider the neighbor code that is ended with “2” or “3” besides the sizes of the neighbor codes, while CDOS and CDHS need to consider many more cases to make the label size increase logarithmically, thus the update cost of CDOS and CDHS are not cheap; otherwise the size of CDOS and CDHS will increase very

fast which makes the advantage of CDOS and CDHS not an advantage, i.e. not more compact than CDQS.

In conclusion, CDBS and CDQS are the cheapest two approaches to process updates, as well their sizes are not large.

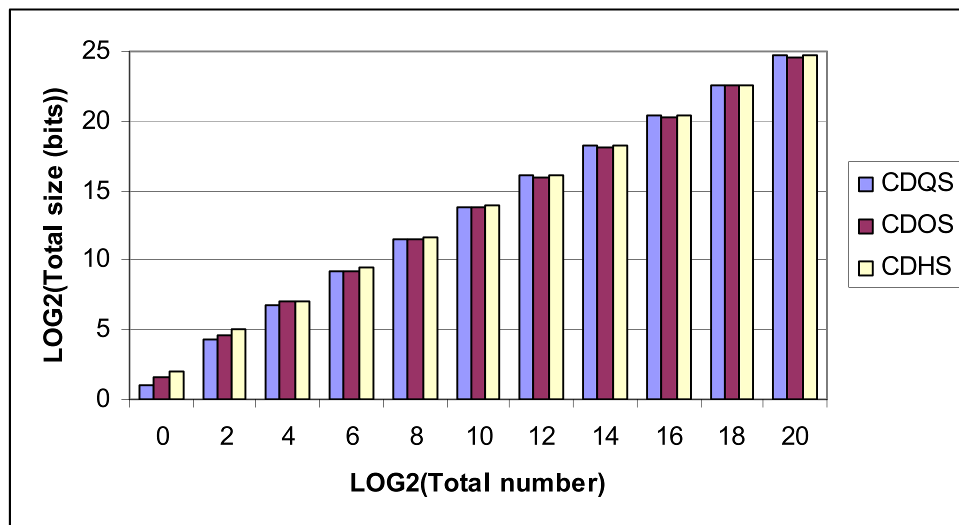


Figure 5.9: Label sizes of different labeling schemes

5.6 Summary

Because the CDBS encoding will encounter the overflow problem which can not completely avoid re-labeling in XML updates, we design the CDQS encoding in this Chapter. Four quaternary strings “0”, “1”, “2” and “3” are used in CDQS, and “0” is used as the separator. CDQS will never encounter the overflow problem, yet it

supports node insertions with the orders kept and without any re-labeling of the existing nodes. Therefore CDQS can *completely* avoid re-labeling in XML updates.

Compared with V-CDBS, the total code size of CDQS is larger and the update cost is larger, i.e. modify the last 2 bits rather than the last 1 bit, but on the other hand, CDQS can completely avoid re-labeling in XML updates.

In summary, V-CDBS is the most compact, and it can process the intermittent and uniformly frequent updates more efficiently if there is no overflow. On the other hand, CDQS can completely avoid re-labeling in XML updates.

We conduct experiments which show that CDQS encoding can completely avoid re-labeling, and it is the only approach to process skewed frequent updates efficiently.

Chapter 6

Controlling the Increase in Label Size

If there are only insertions, Algorithm 4.1 guarantees that the inserted binary string between two consecutive CDBS codes has the smallest size, and Algorithm 5.1 guarantees that the two inserted quaternary strings between two consecutive CDQS codes have the smallest size. In real life, there are many applications which have only insertions but have no deletions. For example, the DBLP inserts the new publications everyday into its XML database, but it will not delete the previous data. The stock XML data also have only insertions but no deletions.

On the other hand, if there are deletions, Algorithm 4.1 and Algorithm 5.1 can not guarantee that the inserted binary string has the smallest size. If we still use Algorithm 4.1 and Algorithm 5.1 to process updates with both insertions and deletions, the label size will increase not so slow. Thus we need to find new algorithms to control the label size increasing speed; meanwhile the new algorithms should also have the ability to keep the orders. Because CDBS is easier to understand, we introduce the new algorithms still based on CDBS. These algorithms can be easily extended for CDQS.

In Section 6.1, we use examples to show, in the update environment with both insertions and deletions, how to find the binary string with the smallest size between two binary strings and with the orders kept. Next in Section 6.2, we discuss a method to process the *skewed insertion* problem (see Section 4.4.4) though our approach works the best to answer queries in skewed insertion environment. The experimental results are reported in Section 6.3, and Section 6.4 summarizes this chapter.

6.1 Finding the Codes with the Smallest Size between Two Codes

Because the examples in this chapter will frequently refer to the V-CDBS codes in Table 4.1, we directly copy the V-CDBS codes in Table 4.1 to Table 6.1. Thus the V-CDBS codes can be easily referred when reading the following examples.

Table 6.1: V-CDBS encoding

	V-CDBS
1	00001
2	0001
3	001
4	0011
5	01
6	01001
7	0101
8	011
9	0111
10	1
11	10001
12	1001
13	101
14	1011
15	11
16	1101
17	111
18	1111

We firstly use an example to show why Algorithm 4.1 can not guarantee that inserted binary string has the smallest size if there are deletions.

Example 6.1 *For the first three V-CDBS codes "00001", "0001" and "001" in Table 6.1, if we use Algorithm 4.1 to insert a binary string between "00001" and "0001", the inserted binary string is "000011". We can not find any other binary strings which are ended with "1", are between "00001" and "0001" lexicographically, and have sizes smaller than or equal to 6 bits, i.e. the size of "000011". That is to say, if there are only insertions, Algorithm 4.1 guarantees that the inserted binary string is always with the smallest size. On the other hand, if there are deletions also, Algorithm 4.1 can not guarantee that the inserted binary string has the smallest size. Suppose that we delete the "0001" between "00001" (S_L) and "001" (S_R). Now if we want to insert a binary string between "00001" and "001", the inserted binary is "000011" based on Algorithm 4.1. Obviously "000011" is not the binary string with the smallest size between "00001" and "001" because "0001" is between "00001" and "001" and its size is smaller than the size of "000011". Therefore we design a new algorithm (Algorithm 6.1) to find the binary string with the smallest size between two binary strings in the update environment with both insertions and deletions.*

The main idea of Algorithm 6.1 is that we compare S_L and S_R bit by bit from left to right to find S_M such that S_M is ended with "1", and S_M has the smallest size in all the codes between S_L and S_R lexicographically.

Algorithm 6.1: AssignMiddleBinaryStringWithSmallestSize(S_L, S_R)

Input: $S_L \prec S_R$; S_L and S_R are both ***ended with "1"***

Output S_M such that $S_L \prec S_M \prec S_R$ lexicographically, and S_M has the smallest size

1: **Case 1** S_L is empty but S_R is NOT empty, i.e. ***insert a code before the first code.***
2: denote the position of the firstly encountered "1" in S_R as P //there must be a "1" in S_R
3: S_T = labeling(S_R, 1, P) //S_T is the Temporarily inserted binary string
4: **if** $S_T \prec S_R$ lexicographically **then** **//Case 1(a)**
5: $S_M = S_T$
6: **else** **//Case 1(b)**
7: S_M = labeling(S_R, 1, P-1) $\oplus$ "01" //change the firstly encountered "1" to "01"
8: **end if**

9: **Case 2** S_L is NOT empty but S_R is empty, i.e. ***insert a code after the last code.***
10: **if** all the bits of S_L are "1" **then** **//Case 2(a)**
11: $S_M = S_L \oplus$ "1"
12: **else** **//Case 2(b)**
13: denote the position of the firstly encountered "0" in S_L as P
14: S_M = labeling(S_L, 1, P-1) $\oplus$ "1" //change the firstly encountered "0" to "1"
15: **end if**

16: **Case 3** S_L is a prefix of S_R. ***Insert a code between two codes.***
17: S_T = labeling(S_R, length(S_L)+1, length(S_R)) //S_T is the Temporarily inserted binary
//string when removing S_L from the left side of S_R
18: denote the position of the firstly encountered "1" in S_T as P //there must be a "1" in S_T
19: S_{T2} = labeling(S_T, 1, P) //S_{T2} is another Temporarily inserted binary string
20: **if** $S_{T2} \prec S_T$ lexicographically **then** **//Case 3(a)**
21: $S_M = S_L \oplus S_{T2}$
22: **else** **//Case 3(b)**
23: $S_M = S_L \oplus$ labeling(S_T, 1, P-1) $\oplus$ "01" //change the firstly encountered "1" to "01"
24: **end if**

25: **Case 4** S_L is not a prefix of S_R. ***Insert a code between two codes.***
26: denote the first difference position of S_L and S_R as P;
27: S_T = labeling(S_L, 1, P-1) //S_T is the Temporarily inserted binary before the first
//different position in S_L and S_R, i.e. $S_L = S_T \oplus$ "0" $\oplus$ "***", and $S_R = S_T \oplus$ "1"
//$\oplus$ "***". Note that "***" is the rest binary string symbols.
28: **if** length(S_R) > P **then** **//Case 4(a)** the P here is the P at line 26
$S_M = S_T \oplus$ "1"
29: **else** //i.e. length(S_R) = P; note that length(S_R) can not be smaller than P
30: S_{T2} = labeling(S_L, P+1, length(S_L)) //S_{T2} is the Temporarily inserted binary string
//from position P+1 to the end position of S_L
31: **if** all the bits of S_{T2} are "1" **then** **//Case 4(b)**
32: $S_M = S_L \oplus$ "0" $\oplus S_{T2} \oplus$ "1"
33: **else** **//Case 4(c)**
34: denote the position of the firstly encountered "0" in S_{T2} as $P2$
35: : $S_M = S_T \oplus$ "0" $\oplus$ labeling(S_{T2}, 1, $P2$-1) $\oplus$ "1"
//change the firstly encountered "0" in S_{T2} to "1"
36: **end if**
37: **end if**

Now we use some intuitive examples to illustrate the different cases in Algorithm 6.1.

Case 1 in Algorithm 6.1

Case 1 is used to insert a code before the first code. The following intuitive example shows how Case 1 works.

Example 6.2 *Case 1(a), suppose we delete the first three V-CDBS codes in Table 6.1, and want to insert a binary string before the current first code "0011". The firstly encountered "1" in "0011" is at the third position; thus S_T = "001", and because $S_T \prec S_R$, $S_M = S_T$ = "001". "001" is the binary string with the smallest size which is smaller than "0011" lexicographically. Case 1(b): suppose we delete the first V-CDBS code in Table 6.1 and want to insert a binary string before the current first code "0001". The firstly encountered "1" in "0001" is at the fourth position; thus S_T = "0001", but because S_T is not lexicographically smaller than S_R, i.e. the first code "0001", we have to change the last "1" in S_T to "01" as the final inserted binary string, i.e. the S_M = "00001" ("0001" $\rightarrow$ "00001"). "00001" is the binary string with the smallest size which is smaller than "0001" lexicographically.*

(II) Case 2 in Algorithm 6.1

Case 2 is used to insert a code after the last code. The following intuitive example shows how Case 2 works.

Example 6.3 *Case 2(a): suppose we delete the last V-CDBS code "1111" in Table 6.1 and want to insert a binary string after the current last code "111". Because all*

the bits of "111" are "1"s, $S_M = S_L \oplus$ "1" = "1111". It can be seen that "1111" is the binary string with the smallest size which is large than "111" lexicographically. Case 2(b): suppose we delete the 13th to 18th V-CDBS codes in Table 6.1, and want to insert a binary string after the current last code "1001". We change the firstly encountered "0" to "1". The firstly encountered "0" in "1001" is at the second bit; we change this "0" to "1", and the inserted binary string is the first two bits of "1001" with "0" changed to "1", i.e. S_M = "11". In this way, we guarantee that the inserted binary string is lexicographically larger than the last code and has the smallest size.

(III) Case 3 in Algorithm 6.1

Case 3 is used to insert a code between two codes. In Case 3, S_L is a prefix of S_R. The following intuitive example shows how Case 3 works.

Example 6.4 *Case 3(a): suppose we delete the two V-CDBS codes between "11" (S_L) and "1111" (S_R) in Table 6.1, and want to insert a new binary string between S_L "11" and S_R "1111". "11" $\prec$ "1111" lexicographically because "11" is a prefix of "1111", therefore this is Case 3. S_T = "11", i.e. the last two bits of S_R "1111". The firstly encountered "1" in S_T is at the first position; thus S_{T2} = "1" i.e. we assume that the temporarily inserted binary string is the first bit of "11". $S_{T2} \prec S_T$, thus $S_M = S_L \oplus S_{T2}$ = "11" $\oplus$ "1" = "111". Obviously "111" is the binary string with the smallest size between "11" and "1111" lexicographically. Similarly Case 3(b) can be processed following the steps for Case 3(b) in Algorithm 6.1; here we do not repeat these steps in Algorithm 6.1.*

(IV) Case 4 in Algorithm 6.1

Case 4 is still used to insert a code between two codes. In Case 4, S_L is not a prefix of S_R. The following intuitive example shows how Case 4 works.

Example 6.5 *Case 4(c), suppose we delete the second code between the first code "00001" (S_L) and the third code "001" (S_R) in Table 6.1, and want to insert a binary string between S_L "00001" and S_R "001". "00001" $\prec$ "001" lexicographically because the third bit of "00001" is "0", while the third bit of "001" is "1", therefore this is Case 4. Because the first difference bit between "00001" and "001" is at position 3, thus S_T = "00". Because length(S_R) = 3 which is not larger than the first difference position between S_L and S_R, S_{T2} = "01", i.e. the last two bits of S_L "00001". Because not all the bits of S_{T2} are "1"s, this is Case 4(c). Finally $S_M = S_T \oplus$ "0" $\oplus$ subString(S_{T2}, 1, P2-1) $\oplus$ "1" = "00" $\oplus$ "0" $\oplus$ "" $\oplus$ "1" = "0001". Obviously S_M "0001" is lexicographically between "00001" and "001" and it has the smallest size, i.e. there are no any other binary strings which are ended with "1", are lexicographically between "00001" and "001", and have smaller or equal sizes as the inserted binary string "0001". Similarly Case 4(a) and Case 4 (b) can be processed following the steps for Case 4(a) and Case 4(b) in Algorithm 6.1; here we do not repeat these steps in Algorithm 6.1.*

Though Cases 3 and 4 are both used to insert a code between two codes, their processing methods are different.

6.2 Handling Insertion Skew

In this section, we introduce a method to process the skewed insertion problem presented in Section 4.4.4 of Chapter 4. Though the experimental results in Section 5.5.2 of Chapter 5 shows that our encoding still works the best to answer queries in the skewed insertion environment because we dramatically decrease update time, here we further discuss one method to control the label size increasing speed in the skewed insertion environment.

Still based on V-CDBS, we introduce the skewness processing method because V-CDBS is easier to understand. This skewness processing method can be easily extended for CDQS.

Skewness Processing Method (SPM) Estimate (based on the characteristics of XML data or probing test) the number of nodes that will be inserted at the fixed place. Based on the estimated number, *pre-calculate* the labels, and assign these labels to the inserted nodes.

Example 6.6 *Suppose that there are 127 codes that are required to be inserted one by one before the first V-CDBS code "00001" (see Table 4.1), then each insertion requires that one more bit should be added for the new inserted code, i.e. the new code will be "000001", "0000001", "00000001" etc. Therefore the code size will increase fast; after inserting 127 codes, the total size for these 127 new codes will be (6 + 132) × 127 / 2 = 8763. It can be seen that without any Skewness Processing Methods (SPM), the label size increases fast in the skewed insertion. On the other hand, if we employ the Skewness Processing Method (SPM), we can pre-calculate the*

*codes for the 127 inserted codes at the beginning. Note that we pre-calculate the codes now, and assign the codes to the inserted nodes only when they are really inserted. The (1/2)th number of the 127 numbers is encoded with "000001" ("0000**1**" → "0000**01**"), the (1/4)th number of the 127 numbers is encoded with "0000001" ("00000**1**" → "00000**01**"), and the (3/4)th number of the 127 numbers is encoded with "0000011" ("00000**1**" ⊕ "1" → "00000**11**"). Similarly we can encode the (1/8)th, (3/8)th, (5/8)th and (7/8)th numbers of the 127 numbers. These steps are similar to the steps in Algorithm 4.2; the difference is that for this example, we know the most right code "00001", but for Algorithm 4.2, both the most left and most right codes are empty at the beginning. In this way, the total size of the new inserted codes is* $N \times \log(N+1) + 4N + \log(N+1)$ = $127 \times \log(127+1) + 4 \times 127 + \log(127+1)$ = *1404; here N is the total number of inserted codes; this formula is only appropriate for this insertion case. It can be seen that 1404 is smaller than 8763, therefore SPI can efficiently process the skewed insertion problem.*

The method in Examples 6.3 can be used for the skewed insertions at other places, and not restricted to the insertions before the first code.

6.3 Experimental Evaluation

In Section 6.3.1, we test Algorithm 6.1 which can find the smallest size binary string between two binary strings in the update environment with both insertions and deletions. In Section 6.3.2, we test the Skewness Processing Method (SPM).

6.3.1 Comparisons of Algorithm 4.1 and Algorithm 6.1

We test the case that nodes are deleted and inserted at the *odd* positions of *Hamlet* file in Shakespeare's play dataset (D1) (see Table 4.2); it is similar for other files in other datasets. After the deletions and insertions, we call this new Hamlet file *Hamlet2*, and this is case 1. Secondly we test that the nodes are deleted and inserted at the *even* positions of *Hamlet2*; we call this new Hamlet file *Hamlet3*, and this is case 2. Thirdly we test that the nodes are deleted and inserted at the *odd* positions of *Hamlet3*; we call this new Hamlet file *Hamlet4*, and this is case 3. We do the similar deletions and insertions till case 10.

We compare the performance of Algorithm 4.1 and Algorithm 6.1 in the update environment with both insertions and deletions. Figure 6.1 shows that the label size of Algorithm 6.1 does not increase in all the 10 cases (since we can find the smallest labels, i.e. reuse the deleted labels in these 10 cases). On the other hand, the label size of Algorithm 4.1 increases linearly (for these 10 cases) which is fast. Note if there are only insertions (no deletions) at different places of XML, the label size of Algorithm 4.1 increases logarithmically but not linearly.

The experimental results confirm that Algorithm 6.1 can efficiently control the increase of the label size. Meanwhile, Algorithm 6.1 can keep the document order without re-labeling also.

Algorithm 6.1 is more appropriate to efficiently process the updates with both insertions and deletions, and Algorithm 4.1 is more appropriate for the updates with insertions only because the cost of Algorithm 4.1 is much smaller, i.e. it only needs to modify the last 1 bit of the neighbor code to get the inserted code.

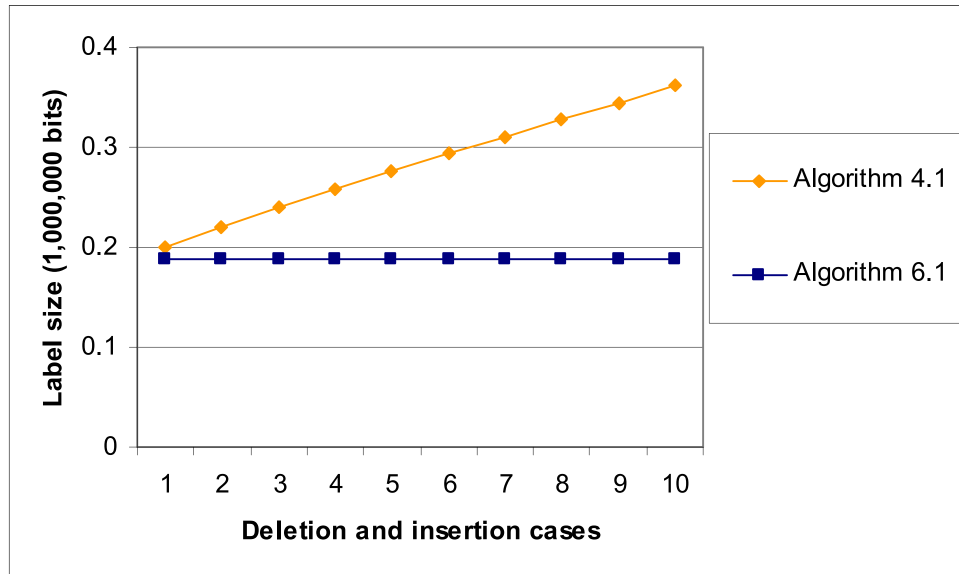


Figure 6.1: Comparison of Algorithm 4.1 and Algorithm 6.1 for CDBS in the update environment with both insertions and deletions

6.3.2 Processing the Skewed Insertion

Now we test the skewness processing method introduced in Section 6.2. Based on the *Hamlet* file of dataset D1 in Table 4.2, we always insert nodes as the first child of the root. Figures 6.2 shows the ***LOG₂** of the total label size (bits) (Y-axis)*. The X-axis of Figure 6.2 shows different number of inserted nodes at a fixed place; note that the Hamlet file originally has totally 6636 nodes. If there is no Skewness Processing Methods (NoSPM), it can be seen from Figure 6.2 that the label size increases very fast. When the Skewness Processing Method (SPM) (see Section 6.2 for the details) is applied to process the skewed insertion problem, the label size increases much slower; see Figure 6.2.

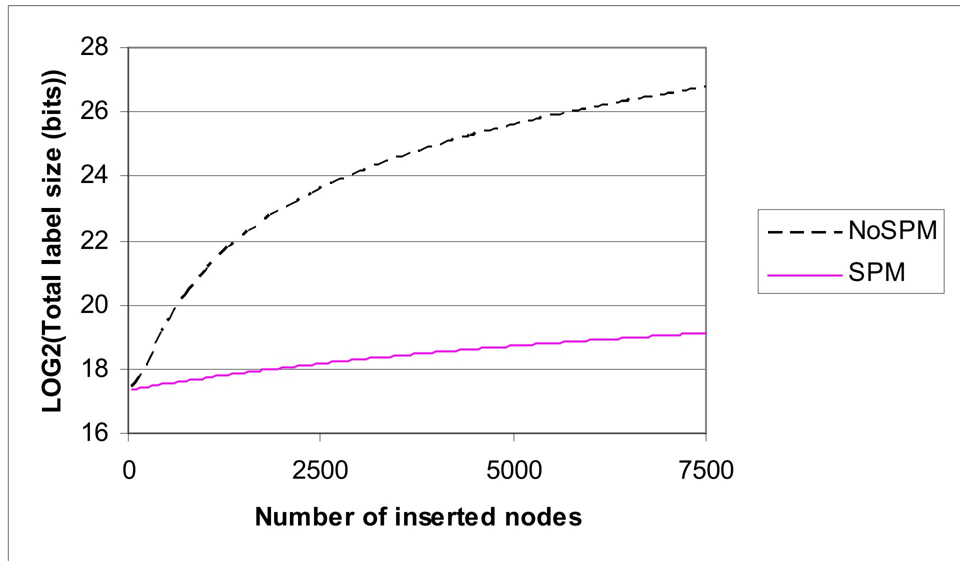


Figure 6.2: Processing of skewed insertions

6.4 Summary

If there are only insertions, Algorithm 4.1 can guarantee that the inserted binary string has the smallest size. If there are both insertions and deletions, Algorithm 4.1 can not guarantee that the inserted binary string has the smallest size. Therefore in this chapter, we firstly designed an algorithm (Algorithm 6.1) which can find the smallest size binary string between two binary strings. In this way, the label size will increase slowly. Accordingly we can keep XML query performance un-decreased. Furthermore, Algorithm 6.1 also supports order-sensitive insertions without re-encoding the existing numbers. In summary, Algorithm 6.1 is more appropriate to efficiently process the updates with both insertions and deletions, and Algorithm 4.1 is more appropriate for the updates with insertions only because the cost of Algorithm

4.1 is much smaller, i.e. it only needs to modify the last 1 bit of the labeling code to get the inserted code.

In addition, to address the skewed insertion problem, we introduce the skewness processing method which can control the label size increasing speed even if the nodes are always inserted at a fixed place of XML. It should be noted that even if we do not use the skewness processing techniques, our approach still works the best to answer queries in the dynamic environment of XML data because our approach saves a lot of time in updating.

The experimental results show that both Algorithm 6.1 in Section 6.1 and the skewness processing method in Section 6.2 can efficiently control the label size increasing speed.

Chapter 7

Conclusion

In this chapter, we summarize the contributions of this thesis and discuss the future works.

7.1 Summary of Contributions

(1) P-Containment Scheme to Improve the Query Efficiency

The core operations in XML query are determining the following four basic relationships, i.e. ancestor-descendant, parent-child, sibling and ordering relationships. The existing labeling schemes are not efficient to determine all the four relationships. Therefore we propose the ***P-Containment*** scheme which can determine all the four basic relationships efficiently no matter what the XML structure is. More important, the P-Containment scheme is used to efficiently process internal node updates and completely avoid re-labeling.

(2) CDBS Encoding to Efficiently Process Updates

One more important problem is how to efficiently process XML updates. The most important contribution of this thesis is that we propose novel techniques which

can efficiently process the updates. The most important feature of CDBS encoding is that our comparison is based on the ***lexicographical order***. Based on the lexicographical order, we have the following *theorem*: given two lexicographically ordered binary strings which are both ended with "1", we can always insert a binary string between the two given binary strings with the orders kept. We proposed the *algorithm* to insert binary strings between two ordered binary strings. This algorithm is the *foundation* of this thesis which supports that order-sensitive updates can be processed efficiently. Also CDBS encoding is the *most compact*, i.e. it is as compact as the binary number encoding of consecutive decimal numbers (there is no gap). The update cost of V-CDBS is the cheapest, i.e. it only needs to modify the last 1 bit of the neighbor code to get the inserted code.

(3) CDQS Encoding to Completely Avoid Re-labeling

On the other hand, CDBS uses the fixed size length field to separate different labels. The fixed size length field will encounter the overflow problem when a lot nodes are inserted into an XML tree. When the size overflows, all the nodes should be re-labeled. In order to solve the overflow problem, we propose the Compact Dynamic Quaternary String (CDQS) encoding. The idea of CDQS is that we use four symbols "0", "1", "2" and "3" for encodings, and each symbol is stored with 2 bits, i.e. "00", "01", "10" and "11". The symbol "0" is used as the separator to separate different codes, and only "1", "2" and "3" are used in the CDQS codes. Note that for P-Containment scheme, we use "0" to separate the "start", "end" and "parent_start", and every three values form a group of "start, end, parent_start". We do not use the

"level" value because it will encounter the overflow problem. For the prefix scheme, we use "0" as the delimiter to separate different components of a label, and use "00" as the separator to separate different labels. Based on a similar idea of CDBS, CDQS also supports order-sensitive insertions. In addition, the separator "0" will never encounter the overflow problem, therefore CDQS can ***completely*** avoid re-labeling in XML leaf node updates. Note that we can not completely avoid re-labeling in internal node updates; this is the drawback of the existing labeling schemes, but not the drawback of our CDQS encoding.

Compared to CDQS, CDBS is more compact, and the variable length CDBS, i.e. V-CDBS only needs to modify the last 1 bit of the neighbor label to get the inserted label, but it can not completely avoid re-labeling. CDQS needs to modify the last 2 bits of the neighbor label to get the inserted label, but it can completely avoid re-labeling. Therefore, if the updates are intermittent or uniformly frequent updates, CDBS can work well; if the updates are skewed frequent updates, only CDQS efficiently works. CDBS and CDQS encodings are orthogonal to specific labeling schemes, therefore they can be applied ***broadly*** to different labeling shemes, e.g. containment, prefix and prime schemes, to maintain the document order when XML is updated.

(4) Combine P-Containment Scheme with CDBS or CDQS Encoding to Efficiently Process Both Queries and Updates

When the P-Containment scheme proposed in this paper is combined with CDBS or CDQS encoding, both the queries and updates can be processed efficiently.

Furthermore, the combination of P-Containment scheme and CDBS or CDQS encoding can help to efficiently process the internal node updates. CDBS-P-Containment or CDQS-P-Containment scheme only needs to modify the "parent_start" values of the child nodes of the inserted or deleted nodes, but need not change any labels of the other descendants of the inserted or deleted node which is much cheaper compared with the existing labeling schemes.

7.2 Future Works

There are no labeling schemes and encoding approaches which completely avoid re-labeling of nodes in internal node updates. Thus we need to consider how to solve this problem in the future.

It can be seen from this thesis that even if we do not handle the skewed insertion problem, our approaches still work the best to answer queries in the frequent update environment of XML data because the update time of our approaches are much smaller. Also we propose a method to process the skewed insertion problem, but this skewness processing method has some restrictions, e.g. it should estimate the number of nodes to be inserted at a fixed place, while the estimation will not be so easy. By balancing the query and update performance [68] or by re-labeling some nodes, we can solve this skewed insertion problem better. In the future, we want to research whether there are approaches that can completely avoid re-labeling and meanwhile solve the skewed insertion problem efficiently, but seems that it is not so easy to solve this problem because seems that these two aspects contradict each other.

Appendices

Appendix A: Meanings of Abbreviations

Table A1 illustrates the meanings of the abbreviations used in this thesis.

Table A1: Symbols to represent the existing labeling schemes

Abbreviations	Meaning
V	Represent Variable length encoding. If there is a V before an encoding name, it means that this encoding has variable length.
F	Represent Fixed length encoding. If there is an F before an encoding name, it means that this encoding has fixed length.
P-Containment	The P in P-Containment represents the “parent_start” value, and the “parent_start” value of a node is the “start” value of its parent.
CDBS	Compact Dynamic Binary String encoding
CDQS	Compact Dynamic Quaternary String encoding

Appendix B: Calculation of the SC Value for Prime Scheme

Chinese Remainder Theorem [7, 74] Let M = [m_1, m_2, ⋯, m_k] and N = [n_1, n_2, ⋯, n_k] be two lists of integers. If the Greatest Common Divisor GCD(m_1, m_2, ⋯, m_k) = 1, the Simultaneous Congruence SC(M, N) = x satisfies that x *mod* m_1 = n_1, x *mod* m_2 = n_2, ⋯, x *mod* m_k = n_k, and there exists exactly one solution x between 0 and C, where $C = \prod_{i=1}^{k} m_i$.

The Euler's quotient function $x = (\sum_{i=1}^{k} (C / m_i) \times n_i \times \phi(m_i)) \bmod C$ is used to calculate the x, where $\phi(m_i)$ is the Euler's totient function [7].

The following steps shows the calculation details:

Calculate C firstly, $C = \prod_{i=1}^{k} m_i$, then calculate $m'_i = C / m_i$ for each $i \in \{1,2,\cdots,k\}$. Multiply m' 1, 2, etc. times until m' mod m_i = 1, $i \in \{1,2,\cdots,k\}$. Finally $x = (\sum_{i=1}^{k} (m'_i \times n_i)) \bmod C$.

We use a concrete example to illustrate the calculations.

Example A1 *Suppose M = [2, 5, 7] and N = [1, 2, 3], then* $C = \prod_{i=1}^{k} m_i = 2 \times 5 \times 7 = 70$ *, and* $m'_1 = C / m_1 = 70 / 2 = 35$ *,* $m'_2 = C / m_2 = 70 / 5 = 14$ *, and* $m'_3 = C / m_3 = 70 / 7 = 10$*. Because* $m'_1 \times 1$ *mod* m_1 *= 35 mod 2 = 1, the final* m'_1

is equal to 35. Because $m'_2 \times 1$ *mod* m_2 *= 14 mod 5 ≠ 1,* $m'_2 \times 2$ *mod* m_2 *= 28 mod 5 ≠ 1, and* $m'_2 \times 3$ *mod* m_2 *= 42 mod 5 ≠ 1, we have to multiply* m'_2 *4 times such that* $m'_2 \times 4$ *mod* m_2 *= 56 mod 5 = 1, then the final* m'_2 *is equal to 56. Finally the 1, 2, 3 and 4 times of* m'_3 *mod* m_3 *≠ 1, hence we have to multiply* m'_3 *5 times such that* $m'_3 \times 5$ *mod* m_3 *= 50 mod 7 = 1, thus the final* m'_3 *is equal to 50. Therefore* $x = (\sum_{i=1}^{k} (m'_i \times n_i))$ *mod C* $= (35 \times 1 + 56 \times 2 + 50 \times 3)$ *mod 70 = 17, such that 17 mod M = N, i.e. 17 mod 2 =1, 17 mod 5 = 2, and 17 mod 7 = 3.*

Appendix C: Size Calculations for V-CDBS and CDQS

C1: Calculation of the Total Code Size for V-CDBS

Calculation of

$$1\times 1+2\times 2+2^2\times 3+2^3\times 4+\cdots+2^n\times(n+1)$$

$$=2^0\times 1+2^1\times 2+2^2\times 3+2^3\times 4+\cdots+2^n\times(n+1)$$

$$=(2^0+2^1+2^2+\cdots+2^n)+(2^1\times 1+2^2\times 2+\cdots+2^n\times n)$$

$$=(2^{n+1}-1)+2\times(2^0\times 1+2^1\times 2+\cdots+2^{n-1}\times n)$$

$$=(2^{n+1}-1)+2\times(2^0\times 1+2^1\times 2+\cdots+2^{n-1}\times n)+2\times 2^n\times(n+1)-2\times 2^n\times(n+1)$$

$$=(2^{n+1}-1)+2\times(2^0\times 1+2^1\times 2+\cdots+2^n\times(n+1))-2\times 2^n\times(n+1)$$

Let $x=2^0\times 1+2^1\times 2+2^2\times 3+\cdots+2^n\times(n+1)$, then the above formula becomes:

$$x$$

$$=(2^{n+1}-1)+2x-2\times 2^n\times(n+1)$$

Therefore $x=n\times 2^{n+1}+1$.

C2: Calculation of the Total Code Size for CDQS

Appendix C2 is similar to Appendix C1 which can be ignored from reading.

Calculation of

$$(2\times 3^0)\times(1\times 2)+(2\times 3^1)\times(2\times 2)+(2\times 3^2)\times(3\times 2)+$$

$$\cdots + (2\times 3^n)\times((n+1)\times 2)$$

$$= 4\times(3^0 + 3^1\times 2 + 3^2\times 3 + \cdots + 3^n\times(n+1))$$

We calculate $3^0 + 3^1\times 2 + 3^2\times 3 + \cdots + 3^n\times(n+1)$ firstly, then we multiply the result 4 times.

$$3^0 + 3^1\times 2 + 3^2\times 3 + \cdots + 3^n\times(n+1)$$

$$= (3^0 + 3^1 + 3^2 + \cdots + 3^n) + (3^1\times 1 + 3^2\times 2 + \cdots + 3^n\times n)$$

$$= (3^{n+1} - 1)/2 + 3\times(3^0\times 1 + 3^1\times 2 + \cdots + 3^{n-1}\times n)$$

$$\cdots + 3\times 3^n\times(n+1) - 3\times 3^n\times(n+1)$$

$$= (3^{n+1} - 1)/2 + 3\times(3^0\times 1 + 3^1\times 2 + \cdots + 3^n\times(n+1)) - 3\times 3^n\times(n+1)$$

Let $x = 3^0 + 3^1\times 2 + 3^2\times 3 + \cdots + 3^n\times(n+1)$, then the above formula becomes:

$$x$$

$$= (3^{n+1} - 1)/2 + 3x - 3\times 3^n\times(n+1)$$

Therefore $x = (n/2 + 1/4)\times 3^{n+1} + 1/4$, and we need to multiply x four times to get the final result. Then the final result is: $(2n+1)\times 3^{n+1} + 1$

Appendix D: Calculation of the Positions Based on V-CDBS

In this appendix, we show how to calculate the positions based on V-CDBS codes.

We use the following example to show how to calculate the positions.

Example A2 *The V-CDBS code "01001" in Table 4.1 is corresponding to the 6th number. We show how to calculate this 6 based on the V-CDBS code "01001" and the total number 18 (see Table 4.1). The first bit "0" indicates that "01001" is belong to the first half, i.e. between 0 and 10 (10=0+round((19-0)/2)). The second bit "1" indicates that "01001" is belong to the second half of 0 and 10, i.e. between 5 (5=0+round((10-0)/2)) and 10. The third bit "0" indicates that "01001" is belong to the first half of 5 and 10, i.e. between 5 and 8 (8=5+round((10-5)/2)). The fourth bit "0" indicates that "01001" is belong to the first half of 5 and 8, i.e. between 5 and 7 (7=5+round((8-5)/2)). The fifth bit is the last bit and the last bit is always "1". The number between 5 and 7 is only 6, therefore "01001" corresponds to number 6. In this way, the position of each V-CDBS code can be calculated based on the code itself and the total number.*

It is similar for the position calculation based on CDQS.

Appendix E: Publications During Ph.D. Period

1 **Changqing Li**, Tok Wang Ling, and Min Hu. Efficient updates in dynamic XML: From Binary String to Quaternary String. *Accepted by* ***VLDB Journal***, 2006.

2 **Changqing Li**, Tok Wang Ling, Min Hu. Efficient Processing of Updates in Dynamic XML Data. In *Proc. of the 22nd International Conference on Data Engineering (**ICDE**)*, Apr. 2006. **Best (Student) Paper Award *List (One of the best two student papers; one of the best six papers)***.

3 **Changqing Li**, Tok Wang Ling, Min Hu. Reuse or Never Reuse the Deleted Labels in XML Query Processing Based on Labeling Schemes. In *Proc. of the 11th International Conference on Database Systems for Advanced Applications (**DASFAA**)*, Apr. 2006.

4 **Changqing Li**, Tok Wang Ling. QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. In *Proc. of the 14th International Conference on Information and Knowledge Management (**CIKM**)*, Oct. 2005. **Student Travel Award**.

5 **Changqing Li**, Tok Wang Ling, Jiaheng Lu, Tian Yu. On Reducing Redundancy and Improving Efficiency of XML Labeling Schemes. In *Proc. of the 14th International Conference on Information and Knowledge Management (**CIKM**)*, Oct. 2005. (Poster paper).

6 Jiaheng Lu, Tok Wang Ling, Tian Yu, **Changqing Li**, Wei Ni. Efficient Processing of Ordered XML Twig Pattern. In *Proc. of the 16th Database and Expert Systems Applications (**DEXA**)*, Aug. 2005.

7 **Changqing Li**, Tok Wang Ling. An Improved Prefix Labeling Scheme: A Binary String Approach for Dynamic Ordered XML. In *Proc. of the 10th International Conference on Database Systems for Advanced Applications (**DASFAA**)*, Apr. 2005.

8 **Changqing Li**, Tok Wang Ling. From XML to Semantic Web. In *Proc. of the 10th International Conference on Database Systems for Advanced Applications (**DASFAA**)*, Apr. 2005. (Short paper).

9 **Changqing Li**, Tok Wang Ling. OWL-Based Semantic Conflicts Detection and Resolution for Data Interoperability. In *Proc. of the 23rd Int. Conf. on Conceptual Modeling **(ER) Workshop** LNCS3289*, Nov. 2004.

10 **Changqing Li**, Tok Wang Ling. A Basis for Semantic Web and e-Business: Efficient Organization of Ontology Languages and Ontologies. To appear *as a **Book Chapter** of book Semantic Web Technologies and eBusiness*. Publisher: IDEA GROUP INC. 701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA.

Below are the publications when I was in China for the master degree in Peking University

11 **Changqing Li**, Shiwei Tang, Hongyan Li. Using Associations to Mine the

Thick-Scale E-commerce Personalize Service Information. *Journal of Computer Science*, Jan. 2002.

12 **Changqing Li**, Shiwei Tang, Hongyan Li. The Design of a Whole E-commerce System. *Journal of Computer Science*, Jun. 2001.

13 **Changqing Li**, Wenbing Zhao, Shiwei Tang. A Personalized Service Protocol Based on HTTP. *11th Conference of Computer Networks and Data Communication in China*. Oct. 2000.

Bibliography

[1] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *Proc. of the 12th annual ACM-SIAM Symp. on Discrete Algorithms (SODA'01)*, pages 547-556, 2001.

[2] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proc. of the 16th ACM Symp. on Principles of Database Systems (PODS'97)*, pages 122-133, 1997.

[3] R. Agrawal, A. Borgida, and H.V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'89)*, pages 253-262, 1989.

[4] H. Ait-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Trans. on Progr. Languages and Systems*, 11(1):115-146, 1989.

[5] S. Al-Khalifa, H.V. Jagadish, J.M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. of the 18th Int. Conf. on Data Engineering (ICDE'02)*, pages 141-152, 2002.

[6] T. Amagasa, M. Yoshikawa, and S. Uemura. QRS: A Robust Numbering Scheme for XML Documents. In *Proc. of the 19th Int. Conf. on Data Engineering (ICDE'03)*, pages 705-707, 2003.

[7] J.A. Anderson and J.M. Bell. Number Theory with Application. *Prentice-Hall*, New Jersey, 1997.

[8] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simon. XML path language (XPath) 2.0. *W3C working draft 04*, Apr. 2005.

[9] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simon. XQuery 1.0: An XML Query Language. *W3C working draft 04*, Apr. 2005.

[10] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible markup language (XML) 1.1. *W3C recommendation*, Feb. 2004.

[11] D. Brickley and R.V. Guha. Resource Description Framework Schema (RDFS) Specification 1.0. *W3C Recommendation*, Feb. 2004.

[12] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'02)*, pages 310-321, 2002.

[13] B. Catania, W.Q. Wang, B.C. Ooi, and X. Wang. Lazy XML Updates: Laziness as a Virtue of Update and Structural Join Efficiency. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'05)*, 2005.

[14] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: A graphical language for querying and restructuring XML documents. In *Proc. of the 8th Int. World Wide Web Conf. (WWW'99)*, pages 93-109, 1999.

[15] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *Int. Workshop on the Web and Databases (WebDB'00)*, pages 53-62, 2000.

[16] E.C. Chang. Design and Analysis of Algorithms. Module CS3230 of Department of Computer Science, National University of Singapore.

[17] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On Boosting Holism in XML Twig Pattern Matching using Structural Indexing Techniques. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'05)*, 2005.

[18] Ting Chen, Tok Wang Ling, and Chee Yong Chan. Prefix Path Streaming: A New Clustering Method for Optimal Holistic XML Twig Pattern Matching. In *Proc. of the 15th Int. Conf. on Very Large Data Bases (DEXA'04)*, pages 801-810, 2004.

[19] Zhuo Chen, Tok Wang Ling, Mengchi Liu, and Gillian Dobbie. XTree for Declarative XML Querying. In *Proc. of the 9th Int. Conf. on Database Systems for Advanced Applications (DASFAA'04)*, pages 100-112, 2004.

[20] S.-Y. Chien, Z. Vagena, D. Zhang, V.J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proc. of the 28th Int. Conf. on Very Large Data Bases (VLDB'02)*, pages 263-274, 2002.

[21] V. Christophides, D. Plexousakis, M. Scholl, and S. Tourtounis. On labeling schemes for the semantic web. In *Proc. of the 12th Int. World Wide Web Conf. (WWW'03)*, pages 544-555, 2003.

[22] C. Chung, J. Min, and K. Shim. APEX: an adaptive path index for XML data. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'02)*, pages 121-132, 2002.

[23] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. In *Proc. of the 21th ACM Symp. on Principles of Database Systems (PODS'02)*, pages 271-281, 2002.

[24] B. Cooper, N. Sample, M.J. Franklin, G.R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proc. of the 27th Int. Conf. on Very Large Data Bases (VLDB'01)*, pages 341-350, 2001.

[25] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proc. of the 8th Int. World Wide Web Conf. (WWW'99)*, pages 77-91, 1999.

[26] P.F. Dietz. Maintaining order in a linked list. In *Proc. of the 14th Annual ACM Symp. on Theory of Computing (STOC'82)*, pages 122–127, 1982.

[27] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. of the 16th Annual ACM Symp. on Theory of Computing (STOC'87)*, pages 365-372, 1987.

[28] M. Duong and Y. Zhang. A New Labeling Scheme for Dynamically Updating XML Data. In *Proc. of the 16th Australasian Database Conference (ADC'05)*, pages 185-193, 2005.

[29] M. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *Proc. of the 14th Int. Conf. on Data Engineering (ICDE'98)*, pages 14-23, 1998.

[30] C. Gavoille and D. Peleg. Compact and localized distributed data structures. *Journal of Distributed Computing, Special Issue for the Twenty Years of Distributed Computing Research*, 2003

[31] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. of the 23th Int. Conf. on Very Large Data Bases (VLDB'97)*, pages 436-445, 1997.

[32] T. Grust. Accelerating XPath Location Steps. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'02)*, pages 109-120, 2002.

[33] F.V. Harmelen, J. Hendler, I. Horrocks, D.L. McGuinness, P.F. Patel-Schneider, and L.A. Stein. OWL Web Ontology Language Reference. *W3C Recommendation*, 2004.

[34] H. He, J. Xie, J. Yang, and H. Yu. Asymmetric Batch Incremental View Maintenance. In *Proc. of the 21th Int. Conf. on Data Engineering (ICDE'05)*, pages 106-117, 2005.

[35] H. He and J. Yang. Multiresolution Indexing of XML for Frequent Queries. In *Proc. of the 20th Int. Conf. on Data Engineering (ICDE'04)*, pages 683-694, 2004.

[36] http://www.saxproject.org/

[37] http://www.w3.org/DOM/

[38] http://www.w3.org/XML/Schema

[39] H. Jiang, H. Lu, W. Wang, and B.C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'03)*, pages 253-263, 2003.

[40] Enhua Jiao, Tok Wang Ling, and Chee Yong Chan. PathStack¬: A Holistic Path Join Algorithm for Path Query with not-predicates on XML Data. In *Proc. of the 10th Int. Conf. on Database Systems for Advanced Applications (DASFAA'05)*, pages 113-124, 2005.

[41] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *Proc. of the 13th annual ACM-SIAM Symp. on Discrete Algorithms (SODA'02)*, pages 954-963, 2002.

[42] R. Kaushik, P. Bohannon, J.F. Naughton, and H.F. Korth. Covering indexes for branching path queries. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'02)*, pages 133-144, 2002.

[43] R. Kaushik, P. Bohannon, J.F. Naughton, and P. Shenoy. Updates for Structure Indexes. In *Proc. of the 28th Int. Conf. on Very Large Data Bases (VLDB'02)*, pages 239-250, 2002.

[44] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *Proc. of the 18th Int. Conf. on Data Engineering (ICDE'02)*, pages 129-140, 2002.

[45] D.D. Kha, M. Yoshikawa, and S. Uemura. A Structural Numbering Data. In *Proc. of the 8th Int. Conf. on Extending Database Technology (EDBT'02) Workshop LNCS2490*, pages 91-108, 2002.

[46] D.D. Kha, M. Yoshikawa, and S. Uemura. An XML Indexing Structure with Relative Region Coordinate. In *Proc. of the 17th Int. Conf. on Data Engineering (ICDE'01)*, pages 313-320, 2001.

[47] O. Lassila and R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. *W3C Recommendation*, Feb. 2004.

[48] **Changqing Li**, Tok Wang Ling, and Min Hu. Efficient Processing of Updates in Dynamic XML Data. In *Proc. of the 22nd Int. Conf. on Data Engineering (ICDE'06)*, 2006. **Best Paper Award *List***.

[49] **Changqing Li**, Tok Wang Ling, and Min Hu. Reuse or Never Reuse the Deleted Labels in XML Query Processing Based on Labeling Schemes. In *Proc. of the 11th Int. Conf. on Database Systems for Advanced Applications (DASFAA'06)*, pages 659-673, 2006.

[50] **Changqing Li** and Tok Wang Ling. QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. In *Proc. of the 14th Int. Conf. on Information and Knowledge Management (CIKM'05)*, pages 501-508, 2005. **Student Travel Award**.

[51] **Changqing Li**, Tok Wang Ling, Jiaheng Lu, and Tian Yu. On Reducing Redundancy and Improving Efficiency of XML Labeling Schemes. In *Proc. of the 14th Int. Conf. on Information and Knowledge Management (CIKM'05)*, pages 225-226, 2005. (Poster paper).

[52] **Changqing Li** and Tok Wang Ling. An Improved Prefix Labeling Scheme: A Binary String Approach for Dynamic Ordered XML. In *Proc. of the 10th Int. Conf. on Database Systems for Advanced Applications (DASFAA'05)*, pages 125-137, 2005.

[53] **Changqing Li** and Tok Wang Ling. From XML to Semantic Web. In *Proc. of the 10th Int. Conf. on Database Systems for Advanced Applications (DASFAA'05)*, pages 582-587, 2005. (Short paper).

[54] **Changqing Li** and Tok Wang Ling. OWL-Based Semantic Conflicts Detection and Resolution for Data Interoperability. In *Proc. of the 23rd Int. Conf. on Conceptual Modeling (ER'04) Workshop LNCS3289*, pages 266-277, Nov. 2004.

[55] **Changqing Li**, Tok Wang Ling, and Min Hu. Efficient updates in dynamic XML: From Binary String to Quaternary String. *Accepted by VLDB Journal*, 2006.

[56] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of the 27th Int. Conf. on Very Large Data Bases (VLDB'01)*, pages 361-370, 2001.

[57] Jiaheng Lu, Tok Wang Ling, Chee Yong Chan, and Ting Chen. From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. In *Proc. of the 31st Int. Conf. on Very Large Data Bases (VLDB'05)*, pages 193-204, 2005.

[58] Jiaheng Lu, Tok Wang Ling, Tian Yu, **Changqing Li**, and Wei Ni. Efficient Processing of Ordered XML Twig Pattern. To appear in *Proc. of the 16th Int. Conf. on Database and Expert Systems Applications (DEXA'05)*, pages 300-309, 2005.

[59] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3): 54-66, 1997.

[60] J. McHugh and J. Widom. Query optimization for XML. In *Proc. of the 25th Int. Conf. on Very Large Data Bases (VLDB'99)*, pages 315-326, 1999.

[61] T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proc. of the 7th Int. Conf. on Database Theory (ICDT'99)*, pages 277-295, 1999.

[62] S. Nestorov, J.D. Ullman, J.L. Wiener, and S.S. Chawathe. Representative Objects: Concise Representations of Semistructured, Hierarchial Data. In *Proc. of the 13th Int. Conf. on Data Engineering (ICDE'97)*, pages 79-90, 1997.

[63] NIAGARA Experimental Data. Available at: http://www.cs.wisc.edu/niagara/data.html

[64] P.E. O'Neil, E.J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'04)*, pages 903-908, 2004.

[65] C. Qun, A. Lim, and K.W. Ong. D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'03)*, pages 134-144, 2003.

[66] P. Rao and B. Moon. PRIX: Indexing And Querying XML Using Prüfer Sequences. In *Proc. of the 20th Int. Conf. on Data Engineering (ICDE'04)*, pages 288-300, 2004.

[67] N. Santoro and R. Khatib. Labeling and implict routing in networks. *The Computer J.*, 28:5-8, 1985.

[68] A. Silberstein, H. He, K. Yi, and J. Yang. BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data. In *Proc. of the 21th Int. Conf. on Data Engineering (ICDE'05)*, pages 285-296, 2005.

[69] I. Tatarinov, Z.G. Ives, A.Y. Halevy, and D.S. Weld. Updating XML. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'01)*, 2001.

[70] I. Tatarinov, S. Viglas, K.S. Beyer, J. Shanmugasundaram, E.J. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'02)*, pages 204-215, 2002.

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

[72] H. Wang, S. Park, W. Fan, and P.S. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'03)*, pages 110-121, 2003.

[73] N. Wirth. Type extensions. *ACM Trans. on Progr. Languages and Systems*, 10(2):204-214, 1988

[74] X. Wu, M.L. Lee, and W. Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *Proc. of the 20th Int. Conf. on Data Engineering (ICDE'04)*, pages 66-78, 2004.

[75] G. Xing and B. Tseng. Extendible range-based numbering scheme for xml document. In *Proc. of the Int. Conf. on Information Technology: Coding and Computing (ITCC'04)*, pages 140-141, 2004.

[76] XMark — An XML Benchmark Project. Available at: http://monetdb.cwi.nl/xml/downloads.html

[77] B.B. Yao, M.T. Özsu, and N. Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. In *Proc. of the 20th Int. Conf. on Data Engineering (ICDE'04)*, pages 621-633, 2004.

[78] F. Yergeau. UTF8: A Transformation Format of ISO 10646. Request for Comments (RFC) 2279, January 1998.

[79] K. Yi, H. He, I. Stanoi, and J. Yang. Incremental Maintenance of XML Structural Indexes. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'04)*, pages 491-502, 2004.

[80] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Trans. Internet Techn.*, 1(1): 110-141, 2001.

[81] N. Zhang, S. Agrawal, and M.T. Özsu. BlossomTree: Evaluating XPaths in FLWOR Expressions. In *Proc. of the 21st Int. Conf. on Data Engineering (ICDE'05)*, pages 388-389, 2005.

[82] N. Zhang, V. Kacholia, and M.T. Özsu. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *Proc. of the 20th Int. Conf. on Data Engineering (ICDE'04)*, pages 54-65, 2004.

[83] C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'01)*, pages 425-436, 2001.