A Compact and Scalable Encoding for Updating XML based on Node Labeling Schemes

Martin Francis O'Connor

Bachelor of Science (Hons) in Computer Applications

Master of Science (Research)

A Dissertation submitted in fulfilment of the

requirements for the award of

Doctor of Philosophy (Ph.D.)

to

Dublin City University

Faculty of Engineering and Computing, School of Computing

Supervisor: Dr. Mark Roantree

September 2013

Declaration

Dedicated to Eva

Acknowledgements

I would first like to thank my supervisor Dr. Mark Roantree for his support, expert advice, encouragement and patience throughout the Ph.D. process. In particular, he allowed me the autonomy to find my own path in terms of research goals, objectives and contributions, while carefully guiding me and ensuring that I was on track to meet the requirements and standards of a Ph.D.

I would also like to thank Enterprise Ireland who provided financial support for the first three years of my Ph.D. research and to Dublin City University for the facilities and support structures they provided.

Gratitude is due to my colleagues in the Interoperable Systems Group, in particular - Gerard, Jun and Ken. All of us have now successfully completed the Ph.D. process, and the many conversations we had - from working through ideas to light-hearted discussions and distractions - helped to make the Ph.D. process an enjoyable and positive experience.

I am also grateful to Dr. Geoff Hamilton for his help in reviewing and verifying the mathematical content in Chapter 5 of this dissertation. I want to thank my sister Kay for listening to me on many occasions as I worked out my ideas. To my sister Sheila, I offer a heartfelt thanks for taking on the onerous task of proof reading my thesis in its entirety and suggesting helpful corrections.

I would especially like to thank my family for their encouragement, and in particular, my parents for their endless love and practical support throughout the entire Ph.D. process. Quite simply, without the support of my parents, this dissertation would not have been possible.

Last, but most important of all, I want to thank God for His goodness and the many gifts He has given me, including life, love, faith and the opportunity to complete this Ph.D.

Ad Majorem Dei Gloriam.

Contents

Abstract **xi**

1 Introduction **1**

1.1 The Evolution of XML 2

1.2 Querying and Updating XML Documents 3

1.3 Issues in Updating XML Documents 5

1.4 XML Labeling Schemes 6

1.4.1 XML Trees 6

1.4.2 Labeling Schemes 7

1.4.3 Encoding Schemes 8

1.5 Issues and Motivation 9

1.6 Research Aims and Objectives 11

1.7 Summary 12

2 Literature Review **14**

2.1 Dynamic Node Labeling Schemes 15

2.2 Containment Schemes 16

2.2.1 Analysis of Containment Schemes 18

2.3 Prefix Schemes 19

2.3.1 DeweyID 19

2.3.2 ORDPATH 19

2.3.3 DLN 21

2.3.4 LSDX 23

2.3.5 ImprovedBinary 24

2.3.6 Analysis of Prefix Schemes 26

2.4 Orthogonal Schemes 27

2.4.1 QED: The Quaternary Encoding Dynamic Labeling Scheme . 27

2.4.2 The Vector Labeling Scheme 30

2.4.3 DDE: The Dynamic DEwey Labeling Scheme 31

2.4.4 EXEL: Efficient XML Encoding Labeling 32

2.4.5 Analysis of Orthogonal Dynamic Labeling Schemes 33

2.5 Deleted Node Label Reuse 34

2.5.1 Extended QED Labeling Scheme 34

2.5.2 IBSL: Improved Binary String Labeling Scheme 36

2.5.3 EXEL Labeling Scheme 37

2.5.4 CDBS Labeling Scheme 38

2.5.5 Summary Analysis of Label Reuse 39

2.6 Label Storage Schemes 39

2.6.1 Length Fields 40

2.6.2 Control Tokens 40

2.6.3 Separators 42

2.6.4 Prefix-free Codes 42

2.6.5 Critique of Label Storage Schemes 44

2.7 Summary 45

3 Desirable Properties of Dynamic Labeling Schemes 47

3.1 Aims for a Good Dynamic Labeling Scheme 47

3.2 Evaluation Framework 50

3.3 Summary 54

4 Reuse of Deleted Node Labels 56

4.1 Problem Description 56

4.2 Concepts and Terminology 57

4.3 Attributes Enabling Deleted Node Label Reuse 59

4.4 Enhanced Binary String Labeling (EBSL) 61

4.4.1 e−Lexicographical Order 62

4.4.2 Self_label: A Positional Identifier 63
4.4.3 The `AssignInitialLabels` Algorithm 64
4.4.4 The `SimpleInsertion` Algorithm 65
4.4.5 EBSL Label Categories 66
4.4.6 Theorems and Proofs 67
4.4.7 Summary 69
4.5 EBSL: Reusing Deleted Node Labels 70
4.5.1 Case 1: Inserting a New Node Before the Leftmost Node . . . 71
4.5.2 Case 2: Inserting a New Node After the Rightmost Node . . 72
4.5.3 Case 3: Inserting New Node Between Two Consecutive Nodes 74
4.6 Analysis of the EBSL Labeling Scheme 76
4.6.1 EBSL Label Size Analysis 76
4.6.2 Critique of EBSL Labeling Scheme 78

5 Compact Adaptive Growth Method 80
5.1 The SCOOTER Dynamic Labeling Scheme 81
5.1.1 SCOOTER Lexicographical Order 82
5.2 Assigning Initial Labels 83
5.2.1 Rules for Assigning Labels 83
5.2.2 The `AssignInitialLabels_Sequential` Algorithm 84
5.2.3 The `Increment` Algorithm 88
5.2.4 The `AssignInitialLabels_NodeK` Algorithm 89
5.2.5 Summary Analysis of `AssignInitialLabels` Algorithms . . 91
5.3 Compact Adaptive Growth Method 92
5.4 Compact Adaptive Growth Evaluation 96
5.4.1 Analysis 102

6 SCOOTER Node Label Insertion Algorithms 103
6.1 Insertion After the Current Rightmost Node. 104
6.1.1 Analysis of `InsertNewNodeAfterRightmostNode` Algorithm . 106
6.2 Insertion Before the Current Leftmost Node. 107
6.2.1 Analysis of `InsertNewNodeBeforeLeftmostNode` Algorithm . 110

6.3 Insertion between Two Consecutive Sibling Nodes 110
6.3.1 The `Insertion_LongerThan` Algorithm 111
6.3.2 The `Insertion_EqualTo` Algorithm 116
6.3.3 The `Insertion_LessThanPrefix` Algorithm 118
6.3.4 The `generateAdaptivePostfix` Algorithm 122
6.3.5 The `Insertion_LessThanNotPrefix` Algorithm 124
6.4 Summary 127

7 A Scalable Binary-Encoded Dynamic Labeling Scheme 128
7.1 The SCOOBER Dynamic Labeling Scheme 129
7.1.1 SCOOBER Lexicographical Order 130
7.2 Assigning Initial Labels 130
7.2.1 Rules for Assigning Labels 130
7.2.2 The `AssignInitialLabels_Sequential` Algorithm 132
7.2.3 The `Increment` Algorithm 135
7.2.4 The `AssignInitialLabels_NodeK` Algorithm 135
7.2.5 Summary Analysis of `AssignInitialLabels` Algorithms . . 136
7.3 Compact Adaptive Growth Method (Binary) 137
7.4 Compact Adaptive Growth (Binary) Evaluation 140
7.5 SCOOBER Node Label Insertion Algorithms 140
7.6 Label Storage Scheme for Binary-Encoded Bit-String Labels 141
7.6.1 Fibonacci Label Storage Scheme 142
7.6.2 Encoding and Decoding the Length of the Label 143
7.6.3 Fibonacci Label Storage Scheme Size Analysis 145
7.7 Summary 146

8 Experiments 147
8.1 Label Storage Scheme Evaluation 148
8.2 AssignInitialLabels Evaluation 151
8.3 Node Insertion Evaluation 153
8.3.1 Insertion After Rightmost Node Evaluation 153
8.3.2 Frequently Skewed Insertions Evaluation 155

8.4 Node Insertion with Reuse Evaluation 157
8.5 Evaluation Framework 159
8.6 Summary 159

9 Conclusions 161
9.1 Thesis Summary 161
9.1.1 Impact on the Field 166
9.2 Future Work 167
9.2.1 Short Term Research Avenues 167
9.2.2 Long Term Research Avenues 168

Appendix A Related Peer Reviewed Publications 170

Appendix B SCOOTER Algorithms 172
B.1 The SCOOTER `Decrement` Algorithm 172

Appendix C Compact Adaptive Growth (Binary) Evaluation 174
C.1 Compact Adaptive Growth (Binary) Evaluation 174
C.1.1 Analysis 178

Appendix D SCOOBER Algorithms 179
D.1 The SCOOBER `InsertNewNodeAfterRightmostNode` Algorithm . . 179
D.2 The SCOOBER `InsertNewNodeBeforeLeftmostNode` Algorithm . . 179
D.3 The SCOOBER `Insertion_LongerThan` Algorithm 179
D.4 The SCOOBER `Insertion_EqualTo` Algorithm 179
D.5 The SCOOBER `Insertion_LessThanPrefix` Algorithm 179
D.6 The SCOOBER `generateAdaptivePostfix` Algorithm 179
D.7 The SCOOBER `Insertion_LessThanNotPrefix` Algorithm 179
D.8 The SCOOBER `Decrement` Algorithm 179

Bibliography 185

List of Figures

1.1 Three Representations of a Sample XML File 8
1.2 An XML Encoding of the Sample XML File 9

2.1 Preorder/Postorder Labeled XML Tree 17
2.2 Start/End Labeled XML Tree 18
2.3 DeweyID Labeled XML Tree 19
2.4 ORDPATH Labeled XML Tree 20
2.5 LSDX Labeled XML Tree 23
2.6 ImprovedBinary Labeled XML Tree 25
2.7 Reusing a Deleted Node Label in a QED Labeled Tree 35
2.8 Duplicate Label Assignment in a QED Labeled Tree 35
2.9 IBSL Labeled XML Tree 36
2.10 IBSL Case 3: Inserting a New Rightmost Node after a Node Deletion 37
2.11 Reusing a Deleted Node Label in the EXEL Labeled Tree 38
2.12 Reusing a Deleted Node Label in the CDBS Labeled Tree 39

3.1 Evaluation Framework 51

4.1 IBSL Labeled XML Tree 60
4.2 An EBSL Tree Labeled by the `AssignInitialLabels` Algorithm . . 64
4.3 An EBSL Labeled Tree with New Nodes Inserted (Dotted Circles) using the `SimpleInsertion` Algorithm (Algorithm 4.2) 67

8.1 Storage Costs of Encoding Integers for the Label Storage Schemes . 149
8.2 Integer Encoding and Decoding Times for Label Storage Schemes . . 150

8.3 Algorithms `AssignInitialLabels` Label Storage Costs 151
8.4 Time Taken (Milliseconds) by the `AssignInitialLabels` Algorithms 152
8.5 Algorithm `InsertNewNodeAfterRightmostNode` Label Storage Costs 154
8.6 Algorithm `InsertNewNodeAfterRightmostNode` Average Label Sizes 155
8.7 Bulk Node Insertions Label Storage Costs 156
8.8 Fixed Point Insertions Label Storage Costs 157
8.9 Insertion with Deleted Node Label Reuse Storage Costs 158
8.10 Evaluation Framework (With SCOOTER and SCOOBER) 159

List of Tables

2.1 DLN Control Tokens 22
2.2 QED Sample Labels 28
2.3 UTF-8 Multi-byte Encoding using Control Tokens 41
2.4 ORDPATH Variable-Length Prefix-Free Codes and Value Range . . 43

5.1 SCOOTER and QED Sample Labels 84
5.2 SCOOTER Labels When Inserting After Rightmost Node 94
5.3 SCOOTER Compact Adaptive Growth Rate Table 94

6.1 SCOOTER Labels When Inserting Before Leftmost Node 108

7.1 SCOOBER and CDBS Sample Labels 131
7.2 SCOOBER Labels When Inserting After Rightmost Node 138
7.3 SCOOBER Compact Adaptive Growth Rate Table 138
7.4 Fibonacci Coded Label Length Growth Rate 146

List of Algorithms

4.1 EBSL `AssignInitialLabels` 64

4.2 EBSL `SimpleInsertion` 65

4.3 EBSL `InsertNewNodeBeforeLeftmostNode` 71

4.4 EBSL `InsertNewNodeAfterRightmostNode` 73

4.5 EBSL `InsertNewNodeBetweenTwoConsecutiveNodes` 75

4.6 EBSL `InsertAfterNlabel` 77

5.1 SCOOTER `AssignInitialLabels_Sequential` 85

5.2 SCOOTER `Increment` 89

5.3 SCOOTER `AssignInitialLabel_NodeK` 90

5.4 SCOOTER `ComputePrefixPostfixLengths` 95

6.1 SCOOTER `InsertNewNodeAfterRightmostNode` 105

6.2 SCOOTER `InsertNewNodeBeforeLeftmostNode` 109

6.3 SCOOTER `Insertion_LongerThan` 112

6.4 SCOOTER `Insertion_EqualTo` 116

6.5 SCOOTER `Insertion_LessThanPrefix` 119

6.6 SCOOTER `generateAdaptivePostfix` 123

6.7 SCOOTER `Insertion_LessThanNotPrefix` 125

7.1 SCOOBER `AssignInitialLabels_Sequential` 132

7.2 SCOOBER `Increment` 135

7.3 SCOOBER `AssignInitialLabel_NodeK` 136

7.4 Fibonacci Label Storage Scheme `EncodeLabelLength` 145

7.5 Fibonacci Label Storage Scheme `DecodeLabelLength` 145

B.1 SCOOTER `Decrement` 173

D.1 SCOOBER `InsertNewNodeAfterRightmostNode` 180
D.2 SCOOBER `InsertNewNodeBeforeLeftmostNode` 181
D.3 SCOOBER `Insertion_LongerThan` 182
D.4 SCOOBER `Insertion_EqualTo` 182
D.5 SCOOBER `Insertion_LessThanPrefix` 183
D.6 SCOOBER `generateAdaptivePostfix` 183
D.7 SCOOBER `Insertion_LessThanNotPrefix` 184
D.8 SCOOBER `Decrement` 184

Abstract

The eXtensible Markup Language (XML) has been adopted as the new standard for data exchange on the World Wide Web. As the rate of adoption increases, there is an ever pressing need to store, query and update XML in its native format, thereby eliminating the overhead of parsing and transforming XML in and out of various data formats. However, the hierarchical, ordered and semi-structured properties of the tree structure underlying the XML data model presents many challenges to updating XML. In particular, many of the tree labeling schemes were designed to solve a particular problem or provide a particular feature, often at the expense of other important features. In this dissertation, we identify the core properties that are representative of the desirable characteristics of a *good* dynamic labeling scheme for XML. We focus on four features central to the outstanding problems in existing dynamic labeling schemes; namely a compact label encoding, scalability, deleted node label reuse and a label storage scheme for binary-encoded bit-string node labels. At present there is no dynamic labeling scheme that integrates support for all four features. We present a novel compact and scalable adaptive encoding method to facilitate a highly constrained growth rate of label size under arbitrary node insertion and deletion scenarios and our encoding method can scale efficiently. We deploy our encoding method in two novel dynamic labeling schemes for XML that can completely avoid node relabeling, process frequently skewed insertions gracefully and reuse deleted node labels.

Chapter 1

Introduction

The eXtensible Markup language (XML) [59] has become the defacto standard for information exchange over the Internet and finds widespread use in industry as the data format for interoperability between enterprise applications and web services. One of the key reasons underpinning the acceptance of XML is its expressive and extensible nature. XML is expressive in that it often allows data to be stored in a format that naturally fits the structure of the data - such as hierarchical information. XML is extensible in that the data is semantically enriched with tags to provide meaning and context, and the addition of new tags will not adversely affect existing tags. Consequently, XML has been incorporated into both mainstream and new database technologies to support the storage, querying and updating of XML repositories. However classic data management issues arise in the provision of XML repositories and in particular, challenges still remain in the provision of an efficient and effective update service. In this Chapter, we outline the evolution of XML; provide an overview of the challenges in the provision of an XML update service; introduce dynamic labeling schemes and highlight their current limitations and finally, provide a motivation and work plan for the research presented in this dissertation.

1.1 The Evolution of XML

The purpose of a Markup Language is to annotate a document with tags. These tags should be declarative in that they describe the structure or other attributes of a document but not how the document is to be processed. The Standard Generalized Markup Language (SGML) [20] was first adopted as an International ISO standard in 1986. SGML is a meta-language - that is a language or specification for defining markup languages. With the advent of the first Internet browsers in the early 1990s, a new markup language called HTML (HyperText Markup Language) was specified using SGML to facilitate the retrieval and presentation of documents over the World Wide Web.

The principle design goal driving the specification of HTML is the uniform and consistent presentation of content. Consequently, all tags must be predefined so as to be understood by all browsers. At the time, there was a growing awareness of the need for a new markup language that facilitates the exchange of documents over the Internet for processing and not just presentation. For any given exchange of documents, it would be advantageous if the participating parties could define their own custom tags suitable for the content and context of that exchange. However, SGML was considered too complex and unsuitable for general usage and HTML was primarily focused on the presentation of content. Hence, the eXtensible Markup Language (XML) was born.

XML first appeared as a World Wide Web Consortium (W3C) working draft in November 1996 and was finally adopted as a formal recommendation in February 1998. The W3C consortium is an international community of software and hardware vendors and interested parties that acts as a governing body for all specifications that concern the various XML initiatives. XML is a subset of SGML and was initially designed to enable generic SGML to be received and processed over the Internet in a way that is now possible with HTML. However, the simplicity and extensibility of XML have exceeded its initial design goals and XML has now become not only the standard for document exchange over the Internet but also for interoperability among heterogeneous information systems.

"XML provides an application-independent, language-independent and platform independent method to markup data" ([45] p2) so as to facilitate the structuring, transport, processing and storage of information. However, XML does nothing in and of itself, it is simply textual information enclosed in tags. By virtue of its plain text format, it is both human-readable and computer-readable and provides a software and hardware independent way to share data. It is a meta-language used to define other markup languages, e.g.: WSDL (Web Service Definition Language), RSS (Rich Site Summary), WML (Wireless Markup Language), SMIL (Synchronized Multimedia Integration Language).

1.2 Querying and Updating XML Documents

The XML Data Model (XDM) [62] was formally adopted as a W3C recommendation *XQuery 1.0 and XPath 2.0 Data Model* in January 2007, and updated in December 2010. The conceptual model underlying the XML data model is the tree, which represents semi-structured data. Semi-structured data has a structure that may not necessarily be known in advance and may be self-describing as is the case with XML. By self-describing, we mean the authors of an XML document can create their own tags and each tag can have its own meaning. Semi-structured data has irregular and variable formation; it may have data with missing or supplementary elements and some elements may have different types. All of these variations are permissible in XML documents. Thus, XML allows for an unlimited number of tree dialects, some with a formal structure described by Document Type Definitions (DTDs) documents [58] or XML Schemata [64] while others have an ad-hoc or schema-less structure (semi-structured or unstructured).

XPath. XPath was first adopted by the W3C consortium in 1999 and was extended significantly in 2007 and renamed to XPath 2.0 [60] to support the newly introduced and formally specified XML Data model. XPath models an XML document as a tree of nodes. There are serveral types of nodes, including elements nodes, text nodes and attribute nodes. *"XPath operates on the abstract, logical structure* *of an XML document rather than its surface syntax"* ([60]). The essential goal of XPath is to provide a way to facilitate the hierarchical addressing of nodes in an XML document tree. XPath selects a sequence of node via an axis. Given a context node, an XPath axis defines a set of nodes relative to the context node. The are 13 XPath axes: child, parent, descendant, descendant-or-self, ancestor, ancestor-or-self, following, following-sibling, preceding, preceding-sibling, attribute, self and namespace [60]. *"XPath is an expression language rather than a programming language. In its simplest form, an XPath expression takes an XML document as input and outputs a sequence of nodes that satisfy the expression"* ([45] p3).

XQuery. XQuery is a query language for XML designed *"to query a broad spectrum of XML information sources, including both databases and documents... whether physically stored in XML or viewed as XML via middleware"* ([61]). It is a formal W3C recommendation as of January 2007. *"XQuery is a functional language - instead of executing commands as procedural languages do, every query is an expression to be evaluated, and expressions can be combined quite flexibly with other expressions to create new expressions"* ([10] p4). XQuery is an extension of XPath 2.0 and like XPath, XQuery operates on the abstract logical structure of an XML document rather than its surface syntax. *"Any expression that is syntactically valid and executes successfully in both XPath 2.0 and XQuery 1.0 will return the same result in both languages"* ([61]).

XQuery Update Facility. The XQuery language is essentially a read-only language. The XQuery Update Facility 1.0 is an extension to the XQuery language and *"provides expressions that can be used to make persistent changes to instances of the XQuery 1.0 and XPath 2.0 Data Model"* ([63]). It is a formal W3C recommendation as of March 2011. The XQuery Update Facility allows the following operations; *"insertion of a node, deletion of a node and modification of a node by changing some of its properties while preserving its node identity"* ([63]). The update requests on a XML document are expressed as Pending Update Lists (PULs) and require a capacity for dynamic reasoning to correctly process updates under various scenarios (such as collaborative editing or document versioning) [8].

1.3 Issues in Updating XML Documents

At present, most modern databases provide support for storing and querying XML documents. They also support the updating of XML data at the document level, but provide limited and inefficient support for the more fine-grained (node-based) updates within XML documents. The ability to support XML updates is a difficult and challenging problem because the basic structure (data model) underlying an XML document is a tree. The traditional (relational) database has a structure similar to a spreadsheet (a table with rows and columns) and the provision of table-based updates are a well-understood problem. However, updating a tree-based structure is much more challenging. There are four key differences between the simple table-based model and the more complex XML (tree-based) model.

- *Hierarchical:* A table has a flat representation; a tree has a hierarchical representation.
- *Order:* The order of the rows in a database table do not matter (they have a unique id or primary key), the order of branches (nodes) and leaves in a tree is important.
- *Semi-structured:* A table has a fixed structure (rows multiplied by columns); a tree has a flexible structure (an arbitrary combination of branches and leaves).
- *Meta-data:* The information describing the meaning of the data in a table is stored separately from the table - consequently a schema is mandatory for a table in a relational database; In XML, the meaning of the data is stored in tags (markup) with the data itself inside the XML document. An XML document may also conform to an external schema such as a DTD (Document Type Definition) file or an XML schema document, but such schemata are optional and not an essential requirement for every XML document.

The differences that set the XML data model apart from the Relational data model present many unique challenges for the end-to-end management of XML documents, including storage, indexing, query planning, query processing, query optimization and result serialization. Furthermore, in a transaction processing environment, there is a need to support the reversion (or reversal) of node-based updates to permit the rollback of intermediate states if a transaction cannot be completed fully in order to maintain the consistency and integrity of an XML Database [9]. Consequently, the provision of XML updates must facilitate all of these services in addition to supporting arbitrary node insertions and deletions within an XML document, and guaranteeing unique node identity and node order at all times. Indeed, the provision of an efficient XML update service is still an open research problem. As the volume of XML data increases, and XML databases and XML repositories become more mainstream, the ability to support node-based XML updates efficiently is ever more pressing.

1.4 XML Labeling Schemes

As this dissertation is focused on dynamic labeling schemes to support XML updates, an overview of the core terminology will help to define the context and set the scope of our work. In the context of XML, the terms *labeling scheme* and *encoding scheme* are often used interchangeably. To begin, it is important to clarify what those terms mean and how they relate to each other.

1.4.1 XML Trees

Both labeling schemes and encoding schemes are defined over a tree representation of XML data and not the textual XML document. This is a direct consequence of the XML data model defining its operations in terms of a tree representation of an XML document. The leaves in the tree correspond to data values (text) and the internal nodes correspond to XML elements or attributes. *"The tree is an abstract datatype. There is no defined API (Application Programming Interface) and no defined data representation, only a conceptual model that defines the objects (nodes) in a tree, the properties and their relationships"* ([27] p33). The decision to define XPath operations in terms of a tree representation of an XML document was motivated by the fact that an XML document may be constructed from many data sources, such as a view over relational data or an XML fragment constructed by an application in memory. In such cases, it is not required or even desirable to put an XML document through an XML parser each time it is to be processed. In fact, it would put an unnecessary overhead on the query processing costs to convert these views into textual documents and parse them before XPath evaluation could proceed. Furthermore, given that trees are an abstract data type, in order to pass a tree as an argument to an XPath expression, one passes the root node of the tree.

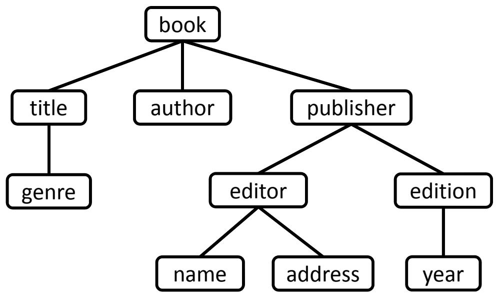
1.4.2 Labeling Schemes

An XPath processor must be capable of distinguishing between nodes to evaluate an expression and this motivates usage of a labeling scheme. The purpose of a labeling scheme is to assign unique labels to each node in the XML tree and these labels must facilitate node ordering. Figure 1.1(a) presents a sample XML document in textual format; Figure 1.1(b) illustrates the XML document tree representation of the sample XML file; and Figure 1.1(c) presents the labeled XML tree of the same document using a prefix labeling scheme. Labels may be numeric, alphabetic, alphanumeric or bit-strings. XPath 2.0 requires all its operators to return a result sequence (which is an ordered group of atomic values or nodes with duplicates permitted) in document order. Consequently, given that no two nodes may occupy the same position at the same time, node labels must be unique to facilitate the determination of node ordering.

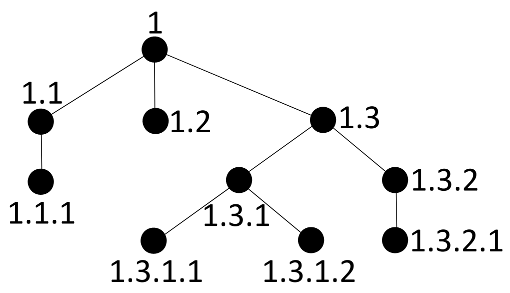
Although a labeling scheme's primary function is to provide unique ids that permit node ordering, often they are constructed to also capture some of the structural semantics of an XML tree itself. A prefix labeling scheme incorporates the labels of the parent and ancestors in the label of the node itself and thus, permit parent-child and ancestor-descendant XPath evaluations. For example, in Figure 1.1(c), node 1.3 is the parent of nodes 1.3.1 and 1.3.2. We will examine the properties of the various labeling schemes in the next Chapter but it is sufficient to say at this point that labeling schemes incorporate *some* of the structural semantics of an XML tree. The precise details of the structural semantics captured are determined by the properties

```
<book>
    <title genre="fantasy">The Hunger Games</title>
    <author>Suzanne Collins</author>
    <publisher>
        <editor>
            <name>Scholastic Inc</name>
            <address>USA</address>
        </editor>
        <edition year="2009">1.0</edition>
    </publisher>
</book>
```

(a) A Sample XML File in Textual Format



(b) An XML Document Tree Representation of the Sample XML File



(c) A DeweyID Prefix Labeled Tree Representation of the Sample XML File

Figure 1.1: Three Representations of a Sample XML File

of the particular labeling scheme employed.

1.4.3 Encoding Schemes

However, no labeling scheme captures the node type, the element or attribute names nor the content of the XML document and thus, lack the sufficient information required to permit full XPath evaluations. These requirements motivate the need for an XML encoding scheme. An XML encoding scheme is constructed upon a labeling scheme and augments it with the information necessary to perform full XPath expression evaluations. A sample encoding of the XML document presented in Figure 1.1(a) is illustrated in Figure 1.2. The XML encoding scheme should also permit the full reconstruction of the textual XML document that corresponds to the XML tree representation. The full reconstruction of the textual XML document cannot be achieved from the labeling scheme alone. In our example in Figure 1.2, the encoding scheme consists of the 4−ary relation E={Prefix Label, Node Type, Name, Value}. There is a one-to-one relationship between each tuple in the encoding scheme and each node in the XML tree.

Prefix Label	Node Type	Name	Value
1	Element	book	
1.1	Element	title	The Hunger Games
1.1.1	Attribute	genre	fantasy
1.2	Element	author	Suzanne Collins
1.3	Element	publisher	
1.3.1	Element	editor	
1.3.1.1	Element	name	Scholastic Inc
1.3.1.2	Element	address	USA
1.3.2	Element	edition	1.0
1.3.2.1	Attribute	year	2009

Figure 1.2: An XML Encoding of the Sample XML File

1.5 Issues and Motivation

The length of a node label is an important criterion in the quality of any dynamic labeling scheme [11] and the larger the label size, the more significant is the negative impact on query and update performance [28]. In the context of XML repositories, XML databases, and large scale XML document collections, larger label sizes result in significant storage costs on disk and in more expensive input and output (I/O) operations and network throughput for distributed systems. Most significant of all, large label sizes lead to higher computational processing costs when performing label comparison operations. Label comparison operations are a fundamental requirement for XPath evaluations in order to determine structural relationships between nodes, such as parent-child, ancestor-descendant and sibling relationships. Label comparison operations are also an essential component of an XML update service. A newly generated node label must permit the determination of node order relative to its sibling nodes. Consequently, a new node label can only be generated by performing an operation based on the existing node labels immediately adjacent to the position of insertion, and thus, requires a label comparison operation. **In summary, it is clear that a key consideration of all dynamic labeling schemes should be the generation of compact node labels to minimize the negative impact of large label sizes on the performance of query and update services.**

It is important to clarify what is meant by the term *compact* node labels. By compact, it is meant that the labeling scheme can ensure the label size will have a highly constrained growth rate both at initial document creation and after subsequent and repeated node insertions. Indeed, only a few dynamic labeling schemes to date [49] [34] [35] [38] [67] [68] [43] assign compact labels at document creation. However, all existing bit-string dynamic node labeling schemes have a best-case linear growth rate for subsequent node label insertions which quickly leads to large labels. The primary reason for the linear growth rate of label sizes under updates is due to the requirement to maintain document order among nodes.

Furthermore, almost all dynamic labeling schemes for XML are not truly dynamic in that they support insertion updates of nodes only. When a node is deleted, the node label is marked as deleted. Subsequently, if a new node is to be inserted at the same position in the XML tree as a previously deleted node, a new (larger) node label is generated. The deleted node label is not reused and is thus, wasted. In a highly dynamic environment with frequent node insertions and deletions, the inability to reuse deleted node labels leads to a rapid increase in label sizes.

Another key consideration for dynamic labeling schemes is the property of scalability. By scalable, we mean the labeling scheme can support an arbitrary number of node insertions and deletions while completely avoiding the need to relabel existing nodes. As the volume of data increases and the size of databases grows from Gigabytes to Terabytes and beyond, the computational costs of relabeling nodes and rebuilding the corresponding indices becomes prohibitive, not to mention the negative impact on query and update services while the indices are under reconstruction.

Each of the dynamic labeling schemes proposed to date have differing characteristics offering distinct advantages and limitations with respect to one another. Indeed, almost all existing research into dynamic labeling schemes for XML was developed in isolation. Each tree labeling scheme was designed to solve a particular problem or provide a particular feature, often without taking into account the impact their solution may have on other key features. Most of the evaluations provided by researchers have focused on comparative performance analysis. No comprehensive analysis of existing dynamic labeling schemes has been performed with a view to identifying the key characteristics of a robust dynamic labeling scheme, the number and specification of the core properties they should provide and the essential requirements they should satisfy.

1.6 Research Aims and Objectives

Given the issues outlined in the previous section, we can now present the research goals that constitute the workplan for this dissertation. The overall research goal is to provide a dynamic labeling scheme for XML that offers a high degree of functionality while minimizing the size of the node labels in the provision of this functionality. The hypothesis put forward in this dissertation is that it is possible to provide a dynamic labeling scheme for XML that is both *compact* and *scalable*. That is, a dynamic labeling scheme that ensures the node label size will have a highly constrained growth rate both at document creation and under arbitrary node insertion and deletion scenarios while completely avoiding the need to relabel nodes. Furthermore, the dynamic labeling scheme will be designed to support the reuse of deleted node labels while maintaining the scalable and compact properties. To demonstrate the effectiveness of our approach, we will provide both theoretical analysis and comparative experimental evaluations with existing state-of-the-art approaches. The main objectives of our research can be highlighted as follows:

- To define a set of requirements that are representative of the characteristics of a *good* dynamic labeling scheme for XML.
- To specify an evaluation framework by which all new and existing dynamic labeling schemes may be evaluated.
- To provide a set of algorithms for the generation of compact labels at document creation.
- To specify a method that facilitates the generation of compact, unique and ordered labels in a dynamic environment. These labels should be significantly smaller than labels generated by existing approaches when processing a large number of updates.

- To provide a suite of algorithms to generate and assign labels supporting any node label insertion and deletion scenario while completely avoiding the need to relabel existing nodes. These algorithms will also support the reuse of deleted node labels.
- To evaluate our approach using our evaluation framework, to provide an analysis of label size growth rates and to perform a comparative performance analysis with the existing state-of-the-art approaches.

1.7 Summary

In this Chapter, the XML data format was introduced and its underlying data model was outlined. The new challenges in the provision of an XML update service over and above the requirements of the tradition relational database table updates were then presented. The rapid growth in the adoption of XML as the standard for data exchange over the Internet and as a format of choice for interoperability among heterogeneous information systems has motivated the development of new XML database technologies and the extension of existing relational databases management systems to support XML documents. These new technologies and extensions will need to support an efficient and effective update service for XML.
Much of the initial research on the construction of labeling schemes for XML focused on efficient query processing and query optimization over static XML data. However, as the volume of XML data increases and the adoption of XML repositories in mainstream industry becomes more widespread, there is a requirement for labeling schemes that can support updates. A major obstacle in the provision of an XML update service is the *limited functionality* provided by existing dynamic labeling schemes and the *rapid growth* in node label size. In this Chapter, we discussed the importance of minimizing the node label size and the positive impact compact node labels have on both XML query and update services. We also highlighted the importance of scalability in the provision of an update service - the ability to completely avoid the relabeling of nodes, eliminates the need for *downtime* as indices do not need to be rebuilt.

We now provide an overview of the structure of this dissertation. In Chapter 2, we perform a detailed literature review of the existing work in this area; in Chapter 3, we identify a template of properties that are representative of the characteristics of a good dynamic labeling scheme for XML; in Chapter 4, we focus on the ability to reuse deleted node labels; in Chapter 5, we focus on the ability to generate compact and scalable labels, and introduce a novel quaternary-encoded bit-string dynamic labeling scheme; in Chapter 6, we present a suite of node label insertion algorithms to generate and assign compact, scalable and reusable labels under any arbitrary node insertion scenario; in Chapter 7, we introduce a new binary-encoded bit-string dynamic labeling scheme for XML and we also present a new label storage scheme for binary-encoded bit-string node labels; in Chapter 8, we present our evaluation and a detailed analysis of our experiments; and finally in Chapter 9 we provide conclusions and discuss potential future work.

Chapter 2

Literature Review

The central focus of our research is on dynamic node labeling schemes for XML updates and in this Chapter we provide a survey of dynamic node labeling schemes for XML. Chapter 2 consists of three parts. In the first part of Chapter 2 we provide a detailed literature review of the state-of-the-art of dynamic labeling schemes for XML and incorporates sections §2.1 to §2.4. In §2.1 we present an overview of the characteristics of dynamic labeling schemes. We identify three broad classifications of dynamic labeling schemes and review each of them in turn; in §2.2 we review the principle containment labeling schemes proposed to date and provide a summary highlighting their limitations in a dynamic environment; in §2.3 we provide a detailed review of five principle prefix labeling schemes proposed to date and conclude with a summary of our findings; in §2.4 we review four orthogonal dynamic labeling schemes and provide a summary analyzing their suitability as dynamic labeling schemes. The second part of Chapter 2 provides an overview of the current limitations in the provision of a dynamic labeling scheme to support the reuse of deleted node labels and is fully contained in §2.5. In the third and final part of Chapter 2, we review the existing label storage schemes to physically encode and store a label on disk in §2.6 and provide a summary detailing the advantages and disadvantages of each approach. Finally, in §2.7 we will conclude with a summary of the Chapter and highlight the key problems to be addressed in this dissertation.

2.1 Dynamic Node Labeling Schemes

Many dynamic labeling schemes for XML have been proposed in recent years and there are several surveys that provide an overview and analysis of the principle labeling schemes [51] [22] [52] [46]. Update operations on XML can be broadly classified as either *content updates* or *structural updates.* Content updates refer to changes in the underlying data values such as element content and the names of elements and attributes. Structural updates refer to the insertion or deletion of nodes in the XML tree. In this dissertation, we are only concerned with structural updates - updates to the structure of an XML tree, because updates to content have no effect on the structure of an XML tree. The elements in an XML document (and nodes in an XML tree) are intrinsically ordered and this order must be maintained in the presence of updates to comply with the semantic requirements of the W3C XPath and XQuery languages. We omit from this literature review the dynamic labeling schemes that do not support the maintenance of document order under updates [53] [11] [66]. We also omit from our literature review labeling schemes that do not allow the determination of node order from the label values alone such as Prime number labeling schemes [65] [33]. Prime number labeling schemes generate a separate auxiliary structure, namely the simultaneous congruence table, to maintain global order among nodes, and as acknowledged by the authors in [33], such schemes do not satisfy the criteria of a dynamic labeling scheme. Before we begin the literature review, we now briefly introduce the current approaches to capturing document order.

Document Order. This is defined for all nodes in an XML tree and corresponds to the order in which the first character of each element occurs in an XML document. There are three generic approaches to capturing document order in a labeling scheme [54]: Global order, Local order and Hybrid order (a hybrid of the local and global order). With global order, a node is assigned an identifier that represents the element's absolute position in the document. With local order, an identifier representing the position of the node relative to its sibling nodes is assigned. A global order approach tends to be more efficient for query processing but unsuitable for a dynamic labeling scheme because insertions modify the positional values of all nodes after the inserted node. A local order approach is more update friendly in that only the following siblings (and their descendants) of an inserted node must be modified. However, local order presents difficulties in the evaluation of several XPath axes. An XPath axis defines a set of nodes relative to a context node in an XML tree. The *following* axis selects all nodes in an XML tree that appear after the context node (excluding any descendant nodes). Local order does not permit the evaluation of the *following* XPath axis because no global order information is available. The hybrid approach attempts to strike a balance between the strengths and weaknesses of global and local order. Most of the dynamic labeling schemes proposed to date follow the hybrid approach.

Almost all dynamic labeling schemes can be broadly categorized under three headings: containment labeling schemes, prefix labeling schemes and orthogonal labeling schemes. Our detailed analysis of these schemes can now commence.

2.2 Containment Schemes

Containment labeling schemes (otherwise known as Interval based labeling schemes or Region encoded labeling schemes) exploit the properties of tree traversal to maintain document order and use containment relationships to determine various structural relationships between nodes. Thus, we begin by introducing tree traversal and then provide an overview of the containment labeling schemes proposed to date.

Tree traversal is the process of visiting each node in a tree data structure. Tree traversal permits the sequential processing of each node once and only once in, what is essentially, a non-sequential data structure. Such traversals are differentiated by the order in which the nodes are visited. In preorder traversal, each node u is visited and assigned its preorder traversal rank *pre(u)* immediately before its children are recursively traversed from left to right. In postorder traversal, a node u is assigned its postorder traversal rank *post(u)* immediately after all its children have been traversed from left to right. Figure 2.1 illustrates an XML tree that is labeled with (pre/post) identifiers. It is worth observing at this point that the act of parsing an XML document in document order, that is, processing each line from left to right and from top to bottom, corresponds to a preorder traversal of the XML document tree. Thus, preorder traversal maintains the document node order of an XML document.

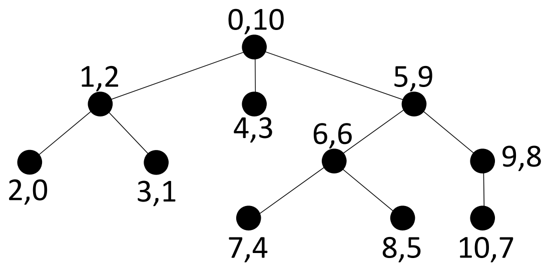


Figure 2.1: Preorder/Postorder Labeled XML Tree

The concept of containment or interval based labeling schemes for tree structured data was first proposed in [14]. The author made use of tree traversals to determine the ancestor-descendant relationships between any given pair of nodes. The author proposed that node u is an ancestor of node ν in a tree τ if and only if u occurs before ν in the preorder traversal of τ and after ν in the postorder traversal. Most containment schemes adopt a global ordering approach to document order.

Several variations of the containment labeling schemes have been proposed [1, 2, 70, 71] that record the start position and end position of each element in the XML document and optionally their level. The start and end positions may be generated by performing a depth-first traversal of the tree and sequentially assigning a number at each visit. Each non-leaf node will be traversed twice, once before visiting all its descendants and once after. Leaf nodes will always contain content values and not structural information and are thus, only visited once. Figure 2.2 depicts an XML tree that is labeled with (start/end) identifiers. The level of an element is its nesting depth in the document. For any pair of nodes, u and ν, u is an ancestor of ν if and only if u.start $<$ ν.start and ν.end $<$ u.end. Essentially, this states that the interval of u contains the interval of ν (e.g.: node (7,15) is an ancestor of node (8,11) in Figure 2.2). By incorporating the level information in the label identifiers, this labeling scheme permits the evaluation of the parent-child axis. Node u is a parent of node ν if and only if u is an ancestor of ν and u.level $= \nu$.level $- 1$. Node u is a sibling of node ν if they share the same parent and are at the same level. In [55], a hybrid ordering approach is adopted whereby sectors are used instead of intervals and mathematical formulae are presented to determine ancestor-descendant and document-order relationships between label pairs.

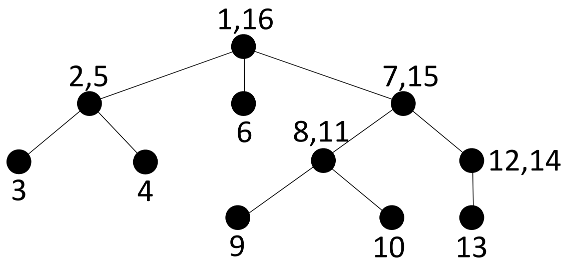


Figure 2.2: Start/End Labeled XML Tree

2.2.1 Analysis of Containment Schemes

The limitation with all of the above containment labeling schemes is that all nodes that appear (in document order) after a newly inserted node must have their labels recomputed. For example in Figure 2.2, a new node inserted between node (6) and node (7,15) will require the labels of node (7,15) and all its descendants to be recomputed and relabeled. Several extensions were proposed [21,29,40] which permit gaps in the labeling schemes to facilitate future insertions gracefully. However, these solutions serve to increase the label size through the sparse allocation of labels and only postpone the relabeling process until the interval gaps have been consumed by the update process. In [3], they propose the use of real (floating point) numbers for label identifiers instead of integers to facilitate an arbitrary number of insertions between two labels. However, computers represent floating point numbers with a fixed number of bits and thus, in practice the solution is similar to an integer representation of labels with sparse allocation and consequently suffers from the same limitation. In other words, none of these solutions are scalable.

2.3 Prefix Schemes

In a prefix labeling scheme, the label of a node in the XML tree consists of the parent's label concatenated with a delimiter and a positional identifier of the node itself. The positional identifier indicates the position of the node relative to its sibling nodes and incorporates the *Local Order* approach for document order. For a pair of nodes u and ν, u is an ancestor of ν if and only if label(u) is a prefix of label(ν). The node's positional identifier combined with its ancestors' labels provide a path vector that uniquely identifies the absolute position of the node in the document - capturing *Global order*. Thus, a prefix labeling scheme follows the *Hybrid order* approach to document order.

2.3.1 DeweyID

DeweyID [54] is an integer-based prefix labeling scheme adapted from the Dewey Decimal Classification system [13] for the organization of library collections. Figure 2.3 illustrates a DeweyID labeled XML tree. DeweyID is a naive prefix labeling scheme whereby the positional identifier of the n^{th} child of a node is assigned the integer n and this is concatenated to the parent's label and a delimiter.

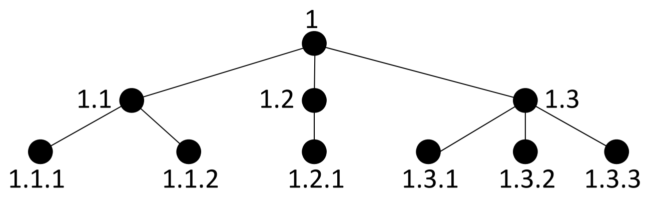


Figure 2.3: DeweyID Labeled XML Tree

Limitations: The insertion of new nodes requires the relabeling of any *following-sibling* nodes (and their descendants) which can have significant costs.

2.3.2 ORDPATH

The ORDPATH labeling scheme [49] is conceptually similar to the DeweyID labeling scheme and permits the insertion of new nodes in arbitrary positions in the XML tree. The XML tree is initially traversed in document order and nodes are labeled with positive, odd integers only (beginning with 1). Even-numbered and negative integer component values are reserved for later node insertions into an existing tree. The ORDPATH labels are not stored as dotted-decimal strings but rather in a compressed binary representation to enable efficient XPath evaluations (to be discussed in §2.6.4). Figure 2.4 illustrates an ORDPATH labeled XML tree where the grey nodes indicate newly inserted nodes. A new node inserted to the right of all existing child nodes is labeled by adding two to the positional identifier of the right-most child node (e.g.: node 1.3.3 in Figure 2.4). In a similar fashion, a new node inserted to the left of all existing child nodes is labeled by adding −2 to the positional identifier of the left-most child node (e.g.: node 1.1.−1 in Figure 2.4). Lastly, a new node is inserted between two consecutive sibling nodes using a *careting-in* technique whereby the positional identifier of the new node is assigned the even-number that sits between the two odd positional identifiers of its neighbor siblings, and then concatenating a new component consisting of an odd number (e.g.: node 1.5.2.1 in Figure 2.4). Subtree insertions may be serialized as a sequence of nodes and inserted individually. The level or depth of each node in the tree may be determined by counting the number of odd component values in the label. ORDPATH labels permit the evaluation of ancestor-descendant, parent-child, sibling-order and document order relationships.

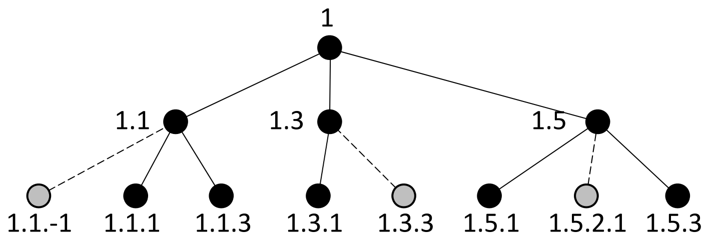


Figure 2.4: ORDPATH Labeled XML Tree

Limitations: The labels assigned by the ORDPATH labeling scheme are not compact. When an XML document is initially labeled, half of the total numbers available are wasted by virtue of employing only odd integers in the labels. Furthermore, node insertion operations result in the generation of node labels such that the positional identifier of a node is always an odd integer; even integers are never used and thus, are wasted. Secondly, consecutive sibling nodes may have labels of differing lengths (e.g.: sibling node labels 1.5.1, 1.5.2.1 and 1.5.3 in Figure 2.4). This results in increased storage costs in the case of frequent node insertions as well as expensive label comparison operations when determining structural relationships (parent-child, ancestor-descendant, etc) between sibling nodes with differing lengths. Thirdly, under frequent node insertions at a fixed point, the length of the labels grow rapidly due to the *careting-in* technique employed by the node insertion operation. Lastly, the ORDPATH labeling scheme cannot completely avoid the relabeling of existing nodes due to the overflow problem (to be discussed in §2.3.6).

2.3.3 DLN

The DLN labeling scheme [7] is an extension of the DeweyID labeling scheme [54] such that a fixed-length of bits is assigned to the labels of the children nodes of any given parent node. The size of the fixed-length assigned to label the children of a parent node may differ from parent node to parent node to take advantage of statistical information known in advance about the XML document and consequently can help to reduce the label size. However, the fixed-length will eventually overflow in the presence of dynamic insertions and to overcome this limitation, DLN introduces the concept of *subvalues.* A subvalue allows for a new positional identifier to be generated between two existing consecutive positional identifiers.
For example, given two consecutive sibling node (DLN) labels 1.2 and 1.3; a new node inserted between them will be assigned the label 1.2/1 . The dot in the label "1.2" indicates a new hierarchical level (that is nodes 1.2 and 1.3 are children of the root node labeled 1), and the forward slash represents a new subvalue. Recall that a fixed length of bits is assigned to label all children of a parent node. Thus, assuming the fixed length of bits is three, then only 7 ($2^3 - 1$) labels are possible in any given subvalue range (the first value 0 is reserved for future insertions to the left). New subvalues are inserted when all possible labels in the existing subvalue range have been consumed by the update process. For example, given a label 1.2/1 where subvalues have a fixed length of 3 bits, if we insert 8 new nodes immediately to the right of this label, the last five labels (5th to 8th) will be 1.2/6, 1.2/7, 1.2/7/1,

Subvalues	Bit Codes (length = 4)	Label range (start - end)
0	0XXX	1 - 7
1	10XX 1 XXXX	8 - 71
2	110X 1 XXXX 1 XXXX	72 - 583
3	1110 1 XXXX 1 XXXX 1 XXXX	584 - 4679
4	1111 1 0XXX 1 XXXX 1 XXXX 1 XXXX	4680 - 37447

Table 2.1: DLN Control Tokens

1.2/7/2 . The dot and the forward slash are used as a visual aid for human legibility. In actuality, when storing DLN labels on disk, a "0" bit is used as a control token to indicate a new hierarchical level between positional identifiers and a "1" bit is used as a control token to indicate the subsequent bit sequence refers to a subvalue. Continuing our example of a 3-bit fixed length subvalue, the label 1.2/6 is stored on disk as *001 0 010 1 110* (the spaces are present for legibility only).
However, a major limitation in this approach acknowledged by the authors is the usage of a fixed-length code for subvalue ranges. If the fixed length is too large, a significant wastage of bits follows. If the fixed length is too small, label sizes grow rapidly with the addition of many suvalues. Furthermore, all internal nodes would have to store metadata detailing the size of the fixed-length code used to label their children. Therefore the authors of DLN proposed a dynamic assignment of fixed-length codes that do not require any advanced knowledge of the documents to be labeled.
Referring to Table 2.1, the first 7 ($2^3 - 1$) child nodes will have positional identifiers that are 4 bits long (subvalue 0). The next 64 child nodes (2^6) will have positional identifiers that are 8 bits long (subvalue 1). Note, when the first bit of a control token is 0, it indicates a new hierarchical level. When the first bit of a control token is 1, it indicates a subvalue. The number of subvalues in a label is computed by counting the number of consecutive control token "1" bits until a control token "0" bit is encountered. Consequently, the next 512 (2^9) child nodes will have positional identifiers that are 12 bits long. This process can be applied repeatedly to encode a label of any length.

Limitations: Although the dynamic and stepwise assignment of DLN fixed-length labels is an improvement over a single predefined fixed-length value, nevertheless an initial fixed-length step must be selected. Due to the heterogeneity of XML and the many types and variations of document-centric and data-centric XML data, there is no predefined fixed-length step that provides a suitable encoding for all labels to ensure compact label sizes. Also, under frequent node insertions at a fixed position the label sizes grow rapidly. Lastly, the DLN dynamic encoding itself contains redundant bits. Specifically, all control tokens "1" that appear after the control token "0" are superfluous and unnecessary [22]. Recall that the number of subvalues is computed by counting the number of consecutive control token "1" bits until a control token "0" bit is encountered. This information is sufficient to determine the length of the subvalue and all subsequent "1" bit control tokens that appear in the label are unnecessary.

2.3.4 LSDX

In [15], the authors present the LSDX labeling scheme which employs both integers and letters in the construction of a node's label. The root node of the tree is label *0a*, where the integer component *0* represents the level or depth of the node and the alphabetic component *a* represents the positional identifier. Figure 2.5 illustrates an LSDX labeled XML tree where the grey nodes indicate newly inserted nodes in an existing tree.

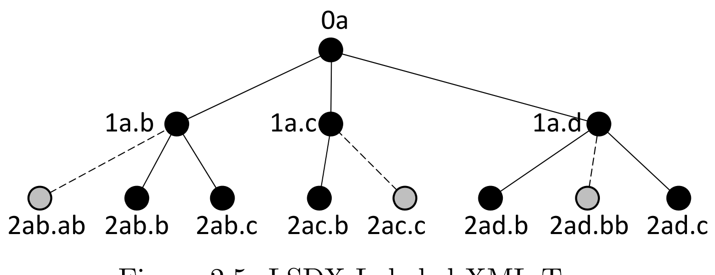


Figure 2.5: LSDX Labeled XML Tree

During the initial XML tree construction, the first child of every node uses the letter *b* instead of *a* to permit future insertions before the first child. If the previously assigned positional identifier is *z*, then the next identifier will be *zb*. A new node inserted to the left of all existing child nodes is labeled by taking the existing leftmost child label and prefixing an *a* to its positional identifier (e.g.: node 2ab.ab in Figure 2.5). A new node inserted to the right of all existing child nodes is labeled by taking the existing rightmost child label and lexicographically incrementing the last letter of its positional identifier (e.g.: node 2ac.c in Figure 2.5). A new node is inserted between two existing nodes by lexicographically incrementing the positional identifier of the new node such that it is greater than its left neighbor and less than its right neighbor (e.g.: node 2ad.bb in Figure 2.5). Essentially, the LSDX labeling scheme supports updates such that the labels of sibling nodes will always have their alphabetic components lexicographically ordered. LSDX permits the evaluation of ancestor-descendant, parent-child and sibling-based relationships. Furthermore, labels are not persistent and may be reassigned upon deletion.

The authors of LSDX acknowledge that the label size grows very quickly for nodes with hundreds of siblings (and this is relatively small) and they propose an improved version of their labeling scheme in [16] called Compressed Dynamic Labeling Scheme or Com-D for short. The basic concept is to compress reoccurring letters within a label by prefixing the repetitive letter(s) with an integer indicating the number of repetitions. For example, the positional identifier *aaaaabcbcbcddddde* would be rewritten as *5a3(bc)4de*. In [30] a labeling scheme very similar to LSDX is proposed and differs only in the method to determine the positional identifier of a node.

Limitations: LSDX and the two labeling schemes derived from it do not always produce unique node labels for several corner-case update scenarios and therefore they are unsuitable for use as dynamic labeling schemes for XML. Examples and illustrations of the labeling collisions that may occur using the LSDX labeling scheme are outlined in [51].

2.3.5 ImprovedBinary

In [34], the authors propose a prefix labeling scheme called *ImprovedBinary* that uses bit strings in conjunction with a recursive algorithm to assign unique labels to each node in the XML tree. Figure 2.6 illustrates an ImprovedBinary labeled XML tree where the grey nodes indicate newly inserted nodes in an existing tree. When the XML tree is initially constructed, the root node is assigned the empty string. Initially the leftmost child of the root node is assigned the positional identifier *01* and the rightmost child of the root node is assigned the positional identifier *011*. From this point onward, the labeling algorithm is a recursive function that takes three inputs: the number of sibling children between the leftmost and rightmost child nodes inclusive; the label of the leftmost child node; and the label of the rightmost child node. An `AssignMiddleSelfLabel` function is invoked to compute a binary string (positional identifier) for the middle node residing between the leftmost and rightmost child nodes (e.g.: node 0101 in Figure 2.6). The middle node is determined using the simple calculation ((1 + n) / 2) where n is the number of child nodes passed to the labeling algorithm. The `AssignMiddleSelfLabel` function takes into account both labels of the leftmost and rightmost nodes as well as their lengths to compute a binary string identifier that is minimal in length while ordered lexicographically between the leftmost and rightmost node labels. This is always possible due to a useful property of the algorithm that ensures the computed binary string always ends with a *1* bit. Finally, the labeling algorithm uses the new left and right node labels to recursively call itself until every sibling node has been assigned a label.

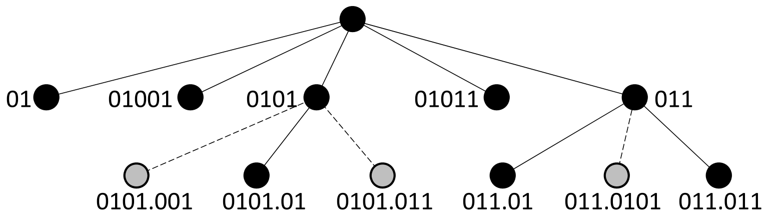


Figure 2.6: ImprovedBinary Labeled XML Tree

There are three types of node insertions possible. To insert a new node before the first (leftmost) sibling node, the positional identifier of the inserted node is assigned the identifier of the first sibling node with the last *1* changed to *01* (e.g.: node 0101.001 in Figure 2.6). To insert a new node after the last (rightmost) sibling node, the positional identifier of the inserted node is assigned the identifier of the last sibling node with an extra *1* concatenated (e.g.: node 0101.011 in Figure 2.6). To insert a node between any two nodes, the `AssignMiddleSelfLabel` function is

used to compute the new positional identifier of the node label (e.g.: node 011.0101 in Figure 2.6).
The ImprovedBinary labeling scheme ensures that the positional identifiers and node prefixes are lexicographically ordered and consequently, node labels are lexicographically ordered when performing component by component comparisons.

Limitations: The label sizes in the ImprovedBinary labeling scheme can grow quite rapidly. In particular, repeated insertions before the first sibling node and after the last sibling node has a label size growth rate of 1-bit for each node insertion. Also, the ImprovedBinary labeling scheme cannot completely avoid the relabeling of existing nodes due to the overflow problem, which is discussed in the next section.

2.3.6 Analysis of Prefix Schemes

All prefix labeling schemes permit the determination of the following structural relationships among nodes: Parent/Child (PC), Ancestor/Descendant (AD), Sibling Order (SO) and Document node Order (DO). However, all of the above prefix labeling schemes (except the ImprovedBinary labeling scheme) require nodes to be relabeled in the presence of updates. More specifically, all sibling nodes that appear (in document order) after a newly inserted node must have their labels recomputed. Hence, prefix labeling is somewhat more dynamic in that node relabeling after updates are limited to following-sibling nodes (and their descendants) and not all following nodes.
However, all of the labeling schemes surveyed thus far (including the ImprovedBinary labeling scheme) cannot avoid the overflow problem. We now describe the overflow problem and in the next section, we provide an overview of the dynamic labeling schemes that overcome this problem.

The Overflow Problem. The Overflow Problem concerns the bit encoding used to store the label on disk or any physical digital medium and affects both fixed-length and variable-length encodings. It should be clear that all fixed length encodings are subject to overflow once all the assigned bits have been consumed by the update process and consequently require the relabeling of all existing nodes. It is not so obvious that variable-length encodings are also subject to the overflow problem. Variable length labels require the size of the label to be stored in addition to the label itself. Thus, if many nodes are inserted into the XML tree, then at some point, the original fixed length of bits assigned to store the size of the label will be too small and overflow, requiring all existing nodes to be relabeled. This problem has been named the overflow problem in [35].

2.4 Orthogonal Schemes

All of the labeling schemes outlined in this section are orthogonal to the different approaches used to determine structural relationships between nodes: in other words, they may be applied to and used as a containment labeling scheme or as prefix labeling scheme.

2.4.1 QED: The Quaternary Encoding Dynamic Labeling Scheme

The authors of the ImprovedBinary labeling scheme proposed a novel quaternary dynamic labeling scheme called QED [35] that can *completely* avoid the relabeling of nodes in the presence of updates. The QED labeling scheme is conceptually very similar to the approach taken by the ImprovedBinary labeling scheme. However, instead of using a binary string, a quaternary code is employed consisting of four numbers *0*, *1*, *2*, *3* and each number is stored using two bits, i.e.: *00*, *01*, *10*, *11*. The number *0* is reserved for use as a separator (delimiter) and only *1*, *2*, and *3* are used in the QED code itself.
The labeling algorithm is a recursive function and operates in a similar manner to the ImprovedBinary scheme outlined earlier. The difference arises from the fact that the ImprovedBinary scheme is based on the $(\frac{1}{2})^{th}$ node position between the leftmost and rightmost child nodes, whereas the QED scheme is based on $(\frac{1}{3})^{th}$ and $(\frac{2}{3})^{th}$ node positions between the leftmost and rightmost child nodes. The `AssignMiddleSelfLabel` function of the ImprovedBinary scheme is replaced with the single `GetOneThirdAndTwoThirdCode` function in the QED scheme. Thus, rather

Decimal	QED Code
1	112
2	12
3	13
4	132
5	2
6	212
7	22
8	222
9	23
10	232
11	3
12	312
13	32
14	33
15	332
Total Size	70 bits

Table 2.2: QED Sample Labels

than computing a middle node self_label as in the ImprovedBinary scheme, the QED scheme computes two QED self_labels (positional identifiers), one each for the $(\frac{1}{3})^{th}$ and $(\frac{2}{3})^{th}$ nodes that reside between the input left and right nodes. The QED labeling scheme maintains document order among nodes using lexicographic order. We use the notation labelA $\prec$ labelB to denote labelA is lexicographically less than labelB. Hence, the two new QED self_labels always have the following lexicographic order properties: left node $\prec$ $(\frac{1}{3})^{th}$ node $\prec$ $(\frac{2}{3})^{th}$ node $\prec$ right node. The QED labeling algorithm recursively calls itself until all nodes in the XML tree have been labeled.

For example, given 15 sibling nodes to label (as illustrated in Table 2.2), the QED labeling algorithm starts by assuming there is one more number before the ordinal number 1 representing the first node, namely number 0, and one more number after the ordinal number 15 representing the last node, namely number 16. Then the ordinal number of the $(\frac{1}{3})^{th}$ sibling node is identified using equation 2.1a and is calculated as follows: 5 = round((0 + (16 − 0)) /3).

$$\frac{1}{3}^{th}\text{node} = \text{round}\ ((\text{leftOrdinal} + (\text{rightOrdinal} - \text{leftOrdinal}))\ /\ 3) \tag{2.1a}$$

$$\frac{2}{3}^{th}\text{node} = \text{round}\ ((\text{leftOrdinal} + (\text{rightOrdinal} - \text{leftOrdinal}))\ *\ 2/3) \tag{2.1b}$$

The ordinal number of the $(\frac{2}{3})^{th}$ sibling node is identified using equation 2.1b and is calculated as follows: 11 = round((0 + (16 − 0)) x 2/3). Finally, the $(\frac{1}{3})^{th}$ sibling node is encoded with "2" and the $(\frac{2}{3})^{th}$ sibling node is encoded with "3" (please refer to Table 2.2). The labeling algorithm is applied recursively taking into account both the leftmost and rightmost node labels as well as their lengths to compute two QED self_labels (one each for the $(\frac{1}{3})^{th}$ and $(\frac{2}{3})^{th}$ nodes) that are minimal in length and lexicographically ordered.

The key mechanism employed to overcome the overflow problem is the use of the separator *0* (2 bits) to identify where one QED self_label ends and another QED self_label starts instead of explicitly storing the size of each self_label. The QED self_labels may vary in size but the size of the separator remains constant. Each number in the QED label will always be represented by two bits and due to the properties of the labeling scheme, the number will never have the 2-bit value *00*, which has been reserved as the separator. For example, when deployed as a prefix labeling scheme, the label 3.12 will be encoded as 11000110 where the first 2 bits "11" denotes the quaternary code 3, the next 2 bits "00" denotes the separator, the next 2 bits "01" denotes the quaternary code 1 and the last two bits "10" denotes the quaternary code 2.

The QED codes are lexicographically and not numerically ordered. Furthermore, the properties of the QED labeling scheme ensure that an infinite number of QED labels may be inserted between any two consecutive node labels without the need to relabel existing nodes and document order will be maintained. The QED labeling scheme is orthogonal to the different classifications of labeling schemes - that is, it may be applied as a prefix labeling scheme or as a containment labeling scheme, or indeed to any application that needs to maintain order in a dynamic sequence.

Limitations: One significant limitation occurs when nodes are repeatedly inserted at a fixed position, the size of the QED labels increase rapidly. The authors of QED attempted to address this problem in [36] and proposed the Compact Dynamic Binary String (CDBS) labeling scheme which is an adaptation of the ImprovedBinary

labeling scheme with more efficient update costs. However, these improvements were made possible through the use of fixed length bit encoding of the labels and thus, are subject to the overflow problem mentioned earlier and will require significant node relabeling after a certain number of node insertions. A detailed presentation of QED and CDBS is presented in [38] (where QED is renamed to Compact Dynamic Quaternary String (CDQS)).
Another limitation occurs when processing moderately large XML documents. The XML document requires a first pass to create a table of QED codes and only after the encoding table has been generated, can the labels actually be assigned to the nodes in the XML document. This is a direct consequence of the recursive property of the label assignment algorithm. Furthermore, when QED is employed as a prefix labeling scheme, each component in the *prefix* label will possibly require its own encoding table (the more irregular the structure of the XML document, the more encoding tables required). Consequently, the memory requirements and computations processing costs to labeling XML documents are significant using QED [68].

2.4.2 The Vector Labeling Scheme

A novel compact dynamic labeling scheme for XML was first proposed in [67] based on vector encoding. The Vector labeling algorithm is conceptually similar to the ImprovedBinary and QED labeling algorithms in that it is a recursive algorithm. Initially the leftmost and rightmost nodes are assigned fixed values - vectors (1,0) and (0,1) respectively. Thereafter, a recursive function is called that assigns to the middle node, a vector that equals the sums of two vectors that corresponds to the start and end positions in each iteration. Document order is maintained among nodes based on the numerical order of the gradients of the vector labels. Although the gradient of a vector is defined in terms of division, this labeling scheme exploits the property permitting the comparison of the gradient of two vectors via multiplication; that is given two vectors A=(x1,y1) and B=(x2,y2), then the gradients (denoted by G) of vectors A and B have the following properties: G(A) > G(B) iff y1x2 > x1y2.
The vector labeling scheme is orthogonal to the different classifications of labeling schemes and the authors provide empirical evidence to show that the update

processing costs are less expensive than QED.

Limitations: The authors state the vector labeling scheme completely avoids the relabeling of existing nodes in the presence of updates by using UTF-8 encoding to process delimiters. UTF-8 [69] is a variable-length character encoding for unicode and will be discussed in more detail in section §2.6.2. However, given that the largest integer that may be encoded with a single UTF-8 6byte instance is $2^{31} - 1$, it is unclear how the vector labeling scheme uses UTF-8 to process delimiters for larger integer values and thus, avoid the overflow problem.

2.4.3 DDE: The Dynamic DEwey Labeling Scheme

The authors of the Vector dynamic labeling scheme integrated their approach with the DeweyID labeling scheme [54] to create a new dynamic labeling scheme called DDE (Dynamic DEwey) [68]. The DDE labeling algorithm [1] initially labels an XML document in the same way as DeweyID and thus, for static XML documents, the two labeling schemes appear indistinguishable. However when inserting a new node between two existing nodes, each component of a DDE label $a_1.a_2.a_3...a_n$ is conceptually viewed as a component of a vector. Therefore, to insert a new node between two existing nodes (with the same number of components), the new node label is simply the vector addition of the two existing node labels. For example, given two nodes A and B, with labels A=2.4.3 and B=1.2.2, a new node C inserted between nodes A and B will have the label C=3.6.5.

The DDE dynamic labeling scheme can determine all four key structural relationships (ancestor-descendant, parent-child, sibling-relationship and document order) from the labels alone. In [68], the authors also proposed an enhanced variant of DDE called CDDE (Compact DDE) designed to optimize the performance of DDE when processing frequent node insertions. The DDE and CDDE labels are physically stored on disk using the compressed ORDPATH format presented in [49].

[1]Strictly speaking, DDE is not an Orthogonal scheme, we present it here because of its relationship with the Vector labeling scheme.

Limitations: In [41], the authors provide a detailed explanation and an illustrated example of how quickly the label sizes grow under frequent insertions between two consecutive sibling nodes. They prove that in the worst case when performing frequent insertions between two consecutive sibling nodes, the value of every component in the new DDE label follows a growth pattern similar to the Fibonacci sequence. To put it in context, after just 31 node insertions every component value of a node label would be a number larger than 1 million (24 bits), and after just 45 node insertions, every component value would be an number larger than 1 billion (32 bits). Given that the minimum number of components in a new DDE label is at least two (only the root node has 1 component), the minimum growth in label length after just 15 node insertions from our example (from 31 to 45) is at least 16 bits. This increase in label size is comparable to a linear growth rate of 1-bit per node insertion. Hence, DDE labels under dynamic insertions are not compact.
Furthermore, after a modest one hundred frequent node insertions between two consecutive sibling nodes, every component value of a node label will be a number larger than 354 billion billion, requiring 69 bits. A 64-bit integer or floating-point number cannot accurately represent a number this large. Consequently, the DDE dynamic labeling scheme is subject to the overflow problem presented in section §2.3.6 after just 100 frequent node insertions between two consecutive sibling nodes.

2.4.4 EXEL: Efficient XML Encoding Labeling

A bit-string dynamic labeling scheme called EXEL (Efficient XML Encoding and Labeling) was first proposed in [42] and extended in [43]. Although the authors present their labeling scheme as a containment labeling scheme, it is orthogonal to the classification and may also be deployed as a prefix labeling scheme. EXEL can support the evaluation of the 4 key structural relationships (ancestor-descendant, parent-child, sibling-relationship and document order). Given a sequence of sibling nodes, the labeling algorithm initializes the first label to 1. For each subsequent label, given the i^{th} label (bit-string) $b(i)$, if $b(i)$ contains a 0 bit, then $b(i{+}1) = b(i) + 10$. Otherwise, $b(i{+}1) = b(i)0^k1$ when k is the length of $b(i)$. For example, the first four labels generated by the EXEL labeling algorithm are 1, 101, 111 and

1110001.
The algorithm to insert a new node between two consecutive sibling nodes is identical to the approach taken by the CDBS labeling scheme [38] but was developed independently. They also propose an update optimization in [43] to containment labeling schemes that minimizes the relabeling required to the descendants of a child node after a node insertion between the parent and that child. The optimization replaces the (start, end, level) label with a (start, end, parent_start)label and is identical to the P-Containment scheme presented in [39] but was developed independently.

Limitations: The EXEL labeling algorithm is not compact. To illustrate, a 7-bit EXEL bit-string label can only represent 8 possible nodes, whereas a 6-bit QED code can represent 18 possible nodes; a 15-bit EXEL bit-string label can only represent 128 possible nodes, a 14-bit QED code can represent 1458 possible nodes. The authors of EXEL recognized this limitation and proposed an enhanced version of EXEL that uses predefined-length bit strings which reduce the size of the labels. However, the labels assigned by CDBS [38] are more compact than the labels generated by Enhanced EXEL because CDBS generates labels that are as small as the binary number encoding of consecutive decimal numbers and hence are the most compact. Furthermore, all existing fixed-length and variable-length bit-string label storage schemes are subject to the overflow problem. Therefore, EXEL is not scalable and will require the relabeling of existing nodes after an arbitrarily large number of node insertions. Lastly, EXEL has a best case linear growth rate (one-bit per node insertion) when processing frequent node insertions between two consecutive sibling nodes, consequently label sizes grow rapidly.

2.4.5 Analysis of Orthogonal Dynamic Labeling Schemes

All orthogonal dynamic labeling schemes facilitate the processing of all 13 XPath axes. Each orthogonal labeling scheme has the distinct property that allow them to be deployed as either a prefix labeling scheme or as a containment labeling scheme. This property gives the database designer the choice to select the appropriate labeling scheme for XML according to the type of XML data to be stored, the type of queries to be processed and the frequency and type of updates to be performed. However, all orthogonal dynamic labeling schemes proposed to date suffer from a linear growth rate in label size under frequent node insertions. Hence, the label sizes are not compact in a dynamic environment. Furthermore, each of the labeling schemes suffer from additional limitations deriving from their individual intrinsic properties. The QED labeling scheme has large memory requirements and high computational costs due to the requirement to generate the complete encoding tables before labels can be assigned. The DDE labeling scheme is subject to the overflow problem in the presence of frequent insertions at a fixed position. EXEL labels are not compact at document creation or under frequent insertions, whereas Enhanced EXEL labels are more compact than the EXEL labels at document creation but are subject to the overflow problem.

2.5 Deleted Node Label Reuse

In the first part of this Chapter, we introduced and described the principle dynamic labeling schemes proposed to date. In the second part of this Chapter, we now review the dynamic labeling schemes claiming to support the reuse of deleted node labels under arbitrary node insertion scenarios. To the best of our knowledge, there are only four published dynamic labeling schemes for XML that claim to guarantee the reuse of deleted node labels while maintaining document order [37] [32] [38] [44]. However, none of these approaches offer a complete solution to guarantee the reuse of a deleted node label. When a node is deleted, the node label is marked as deleted. If a new node is to be inserted at the same position in the XML tree as the previously deleted node, a new node label is generated. The deleted node label is not reused and is thus, wasted. In this section, we illustrate using examples the shortcomings of all four approaches and we conclude that none of the existing solutions are adequate.

2.5.1 Extended QED Labeling Scheme

The Extended QED labeling scheme [37] is an extension to the QED labeling scheme (presented in §2.4) to enable support for the reuse of deleted node labels. The authors provide the Algorithm `AssignInsertedCodeWithReuse` to enable the reuse of deleted node labels and the algorithm claims to have the property of always selecting the smallest deleted node label available.

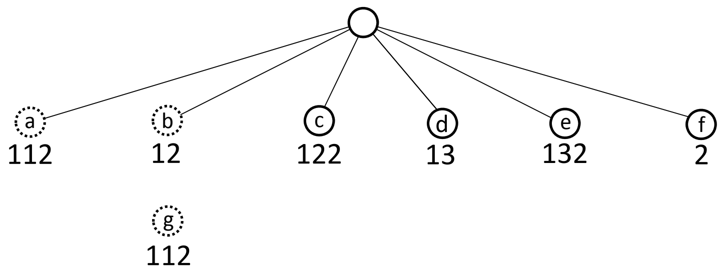


Figure 2.7: Reusing a Deleted Node Label in a QED Labeled Tree

Consider an XML tree which has been initially labeled with 16 nodes. The first 6 nodes labeled *a* to *f* are illustrated in Figure 2.7. Now delete the first two nodes *a* and *b*. Insert a new node *g* before the current leftmost node *c*. According to Algorithm `AssignInsertedCodeWithReuse`, the new node *g* will be assigned the label 112. The original node label 12 (which is also smallest deleted label) of node *b* is not reused.

Perhaps a more significant problem with Algorithm `AssignInsertedCodeWith-Reuse` is that the same label is assigned to two different nodes. Consider an XML tree which has been initially labeled with 16 nodes. The first 4 nodes (*a*, *b*, *c* and *d*) are illustrated in Figure 2.8. Insert a new node *g* to the left of the current leftmost node *a*. Then insert a new node *h* between node g and node *a*. The new node *h* will be assigned the label 1112. This label has already been assigned to node *g*. The assignment of the same node label to two different nodes violates the property of unique node identity as required by the XML data model. Hence, the Algorithm `AssignInsertedCodeWithReuse` makes the Extended QED labeling scheme unsuitable for use as a dynamic labeling scheme for XML.

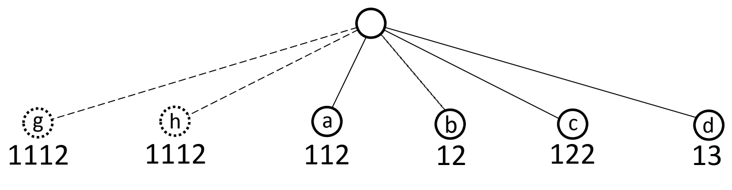


Figure 2.8: Duplicate Label Assignment in a QED Labeled Tree

2.5.2 IBSL: Improved Binary String Labeling Scheme

In [32], the authors propose a dynamic labeling scheme called IBSL (Improved Binary String Labeling). IBSL, an extension of their earlier work [31], is a binary string prefix labeling scheme and introduces new insertion algorithms to permit the reuse of deleted node labels in their original position. Figure 2.9 illustrates an IBSL labeled XML Tree, where the dotted circles indicate newly inserted nodes in an existing tree.

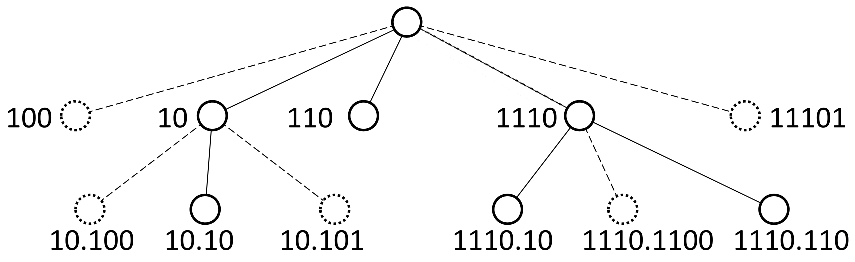


Figure 2.9: IBSL Labeled XML Tree

The authors present an algorithm (Algorithm 3) to insert a new node label with the smallest length between two consecutive node labels. The algorithm is designed to process three generic cases: inserting a new leftmost node (case 1); inserting a new node between two consecutive sibling nodes (case 2); and finally inserting a new rightmost node (case 3). The second case of inserting a new node between two consecutive nodes is broken down further into 3 subcases. Thus, five distinct case scenarios are presented in all. However, it can be shown that the algorithm fails to reuse deleted labels in four of these scenarios. We highlight two of these now.

Case 1: Inserting a new node to the left of the current leftmost node Consider an XML tree which has been initially labeled with just two nodes *a* and *b*. Insert the following new nodes in the order they are listed: node *c* and node *d*, as illustrated in Figure 2.10. Delete node *c* (the current leftmost node is now node *d*). Finally, insert a new node *e* to the left of the current leftmost node *d*. Node label *e* is assigned the label 10010 and not assigned the deleted label of node *c* (100). Thus, Algorithm 3 case 1 did not reuse the deleted node label. Furthermore, the properties of Algorithm 3 ensure the node label 100 will never be reused under any subsequent node insertion scenario.

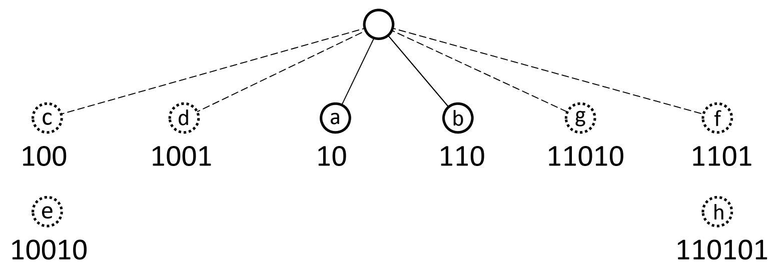


Figure 2.10: IBSL Case 3: Inserting a New Rightmost Node after a Node Deletion

Case 3: Inserting a new node to the right of the current rightmost node Consider an XML tree which has been initially labeled with just two nodes a and b. Insert the following new nodes in the order they are listed: node f and node g, as illustrated in Figure 2.10. Delete node f (the current rightmost node is now node g). Finally, insert a new node h to the right of the current rightmost node g. Node label h = 110101. There is a typographical error in case 3 [32]. However, there is a clear symmetry between case 1 and case 3 and we believe the authors meant to write that N_{new} should be lexicographically greater than N_{left}. In either case, we are certain N_{new} does not reuse the deleted node label that belonged to node f (1101).

2.5.3 EXEL Labeling Scheme

In [44], the authors propose an extension to the EXEL labeling scheme to support the reuse of deleted node labels. They introduce a new algorithm called `ModifiedMakeNewBitString` to enable the reuse of deleted node labels when inserting new nodes, and claim the algorithm has the property of always reusing a deleted node label if it exists.

Consider an XML tree which has been initially labeled with 18 child nodes. In Figure 2.11, we illustrate the fourth, fifth and sixth child nodes which are labeled d, e and f respectively. We perform the following four node operations:

1. Insert a new node g between nodes d and e.

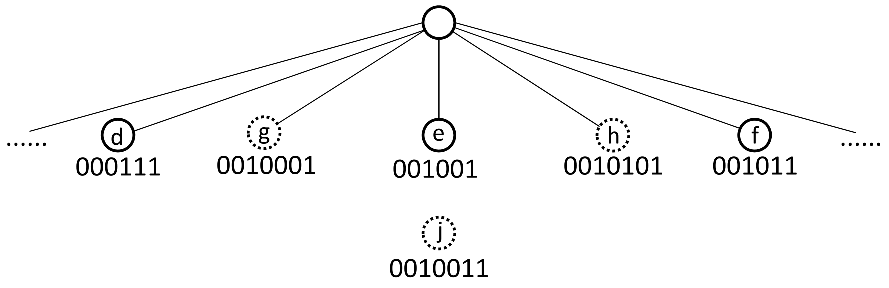


Figure 2.11: Reusing a Deleted Node Label in the EXEL Labeled Tree

2. Insert a new node *h* between nodes *e* and *f*.

3. Then delete node *e*.

4. Finally, insert a new node *j* between the two consecutive nodes *g* and *h*.

The new node *j* will be assigned the label 0010011. The shorter deleted node label of node *e* (001001) is not reused, and is wasted. Furthermore, the properties of Algorithm `ModifiedMakeNewBitString` ensure the node label 001001 will never be reused under any subsequent node insertion scenario.

2.5.4 CDBS Labeling Scheme

In [38], the authors propose an extension to the dynamic labeling scheme CDBS that claims to fully support the reuse of deleted node labels. CDBS is a binary-encoded bit-string dynamic labeling scheme. They introduce a new algorithm called `AssignMiddleBinaryStringWithSmallestSize` to enable the reuse of deleted node labels when inserting new nodes, and claim the algorithm has the property of always selecting the smallest node label when inserting a new node between two consecutive sibling nodes.

Consider an XML tree which has been initially labeled with 18 child nodes. In Figure 2.12, we illustrate the first three child nodes which are labeled *a*, *b* and *c* respectively.

If we insert a new node *d* between nodes *b* and *c*, node *d* is assigned the label 00010001. However, a valid label with the smallest size and lexicographically ordered between nodes *b* and *c* is label 00011 which is not used. Although a subsequent node insertion between node *d* and node *c* will use the label 00011, the

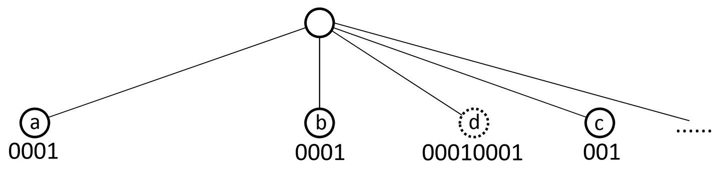


Figure 2.12: Reusing a Deleted Node Label in the CDBS Labeled Tree

`AssignMiddleBinaryStringWithSmallestSize` does not perform according to the stated functionality of always selecting the node label with the smallest size.

2.5.5 Summary Analysis of Label Reuse

In this section, we reviewed four dynamic labeling schemes that claim to support the reuse of deleted node labels under arbitrary node insertion scenarios. Using illustrated examples, we demonstrated that none of these approaches offer a complete solution to guarantee the reuse of a deleted node label. In a dynamic environment with frequent node insertions and deletions, the inability to reuse deleted node labels results in label wastage and a more rapid increase in label size. *Thus, the provision of a dynamic node labeling scheme with the ability to support the reuse of deleted node labels remains an open research question.*

2.6 Label Storage Schemes

In the first part of this Chapter, we introduced and described the principle dynamic labeling schemes proposed to date. In particular, we focused on their labeling algorithms, discussed their ability to support node insertions and highlighted their limitations. In the second part of this Chapter, we reviewed the dynamic labeling schemes claiming to support the reuse of deleted node labels under arbitrary node insertion scenarios. However, a key consideration for all labeling schemes is how they choose to physically encode and store their labels on disk. All digital data is ultimately stored as binary, but the logical representation of the label on disk directly influences the size of the label on disk and the computational cost to encode/decode from the logical to the physical representation. In the final part of this Chapter, we provide an overview of label storage schemes. All existing approaches to the storage of dynamic (variable-length) labels fall under four classifications: length fields, control tokens, separators and prefix-free codes. We employ the same four classifications as those presented in [22].

2.6.1 Length Fields

The concept underlying length fields is to store the length of the label immediately before the label itself. The naive approach is to assign a fixed-length bit code to indicate the length of the label. In a dynamic environment, after a certain number of node insertions, the label size will grow beyond the capacity indicated by the fixed-length bit code and consequently a larger fixed-length bit code will have to be assigned and all existing labels will have to be relabeled according to the new larger fixed-length bit code. One could initially assign a very large fixed-length bit code to minimize the occurrence of the relabeling process, but that would lead to significant wastage in storage for all relatively small labels. In [22], they present several different variations of variable-length bit codes to indicate the size of the label but the authors acknowledge that all of the variable-length approaches lead to either relabeling of existing nodes or involve significant wastage of storage space (before eventually requiring the relabeling of nodes).

2.6.2 Control Tokens

The concept underlying control tokens is similar to length fields, except rather than storing the length of the label immediately before the label itself, tokens stored immediately before the label are used to indicate or *control* how the subsequent bit sequence is to be interpreted. We now provide a brief overview of UTF-8 [69] which is a multi-byte variable encoding that uses control tokens to indicate the size of a label.

UTF-8 is employed by the DeweyID and Vector labeling schemes. Originally, UTF-8 was designed to represent every character in the UNICODE character set, and to be backwardly compatible with the ASCII character set.

Referring to Table 2.3, any number between 0 and 127 ($2^7 - 1$) inclusive, may be represented using 1 byte. The first bit sequence in the label is the control token(s).

If the first bit is the control token "0", it indicates the label length is 1 byte. If the first bit is the control token "1", then the number of bytes used to represent the label is computed by counting the number of consecutive control token "1" bits until the control token "0" bit is encountered. The first two bits of the second and subsequent bytes always consist of the bit sequence "10" as illustrated in Table 2.3.

Value	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6
0 - $(2^7 - 1)$	0xxxxxxx					
2^7 - $(2^{11} - 1)$	110xxxxx	10xxxxxx				
2^{11} - $(2^{16} - 1)$	1110xxxx	10xxxxxx	10xxxxxx			
2^{16} - $(2^{21} - 1)$	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
2^{21} - $(2^{26} - 1)$	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
2^{26} - $(2^{31} - 1)$	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

Table 2.3: UTF-8 Multi-byte Encoding using Control Tokens

Example 2.1. *To encode the DeweyID label 1.152 in UTF-8, we first determine how many bytes each component requires, then convert each component to binary and finally encode using the appropriate number of bytes. 1 is less than 127 ($2^7 - 1$), hence the UTF-8 encoding of 1 is 0 0000001. 152 is between 128 (2^7) and 2047 ($2^{11} - 1$), and 152 in binary is 10011000, hence the UTF-8 encoding of 152 requires two bytes and is 110 00010 10 011000 (the spaces are present for legibility only). Finally, the full UTF8 encoding of the label 1.152 is 0 0000001 110 00010 10 011000.*

The primary limitation of control tokens relate to the requirement to predefine a fixed-length step governing the growth of the labels under dynamic insertions. The fixed-length step cannot dynamically adjust to the characteristics of the XML document or the type of updates to be performed. Also, UTF-8 encodings suffer from bit wastage in their encoding. Recall that the first two bits of the second and subsequent bytes in UTF-8 consists of the bit sequence "10". This was included in the UTF-8 standard to facilitate error correction and recovery algorithms during the transmission of data. However, from the point of view of label storage schemes for XML, this represents a 25 percent wastage of bit capacity in every byte after the first byte.

2.6.3 Separators

Whereas control tokens are used to interpret and give meaning to the sequence of bits that immediately follow the token, a separator reserves a predefined bit sequence to have a particular meaning. Consequently, regardless of where the predefined bit sequence occurs, it must be interpreted as a separator. The QED [35] scheme is the only dynamic labeling scheme to date that employs the separator label storage scheme to encode variable-length labels.

In QED, a quaternary code is defined as consisting of four numbers *0*, *1*, *2*, *3* and each number is stored with two bits, i.e.: *00*, *01*, *10*, *11*. The number *0* (and bit sequence *00*) is reserved as a separator and only *1*, *2*, and *3* are used in the QED code itself. Although the separator label storage scheme is employed by the QED labeling scheme to encode quaternary-encoded bit-string labels, any positive integer can be encoded in the base 3 and represented as a quaternary code and thus, physically stored using the separator label storage scheme. For example, the DeweyID label 2.10.8 can be represented in the base 3 as 2.101.22 and can be encoded using quaternary codes and stored on disk as 11 00 100110 00 1111 (the spaces are present for legibility only).

The primary advantage of the separator label storage schemes over control token label storage schemes is that no matter how big the individual components of a label grow, the separator size remains constant. In the case of quaternary code, the separator size will always be 2-bits no matter how large the label grows. A disadvantage suffered by separator storage schemes compared to control token schemes is that control tokens permit a fast byte-by-byte or bit-by-bit comparison operation [22] and consequently facilitate fast query performance when labels have comparable lengths.

2.6.4 Prefix-free Codes

Prefix-free codes [17] are fixed-length or variable-length codes that are members of a set which have the distinct property that no member (code) in that set is a prefix to any other member (code) in that set. For example, the set m={1,2,3,4} is a prefix set, however the set n={1,2,3,22} is not a prefix set because the member “2” is a prefix of the member “22”.

The ORDPATH [49] dynamic labeling scheme presented in §2.3 uses prefix-free codes as its label storage scheme. In their paper [49], the authors present two prefix-free encoding tables; we present their first encoding table in Table 2.4 and omit the second table as it is conceptually very similar.

Prefix-free Code	Number of bits	Value range
0000001	48	$[-2.8\text{x}10^{14}, -4.3\text{x}10^{9}]$
0000010	32	$[-4.3\text{x}10^{9}, -69977]$
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	$[69976, 4.3\text{x}10^{9}]$
11110	48	$[4.3\text{x}10^{9}, 2.8\text{x}10^{14}]$

Table 2.4: ORDPATH Variable-Length Prefix-Free Codes and Value Range

Example 2.2. *To encode the ORDPATH label 1.152, we must encode each of the components in the label individually. 1 lies in the value range [0, 7] and hence will be represented using three bits (001) and have the prefix code 01. Thus, the full representation of 1 is 01 001. 152 lies in the value range [88, 343] and hence will be represented using 8 bits and have the prefix code 1100. Note, 152 in binary is the 8 digit number 10011000 but ORDPATH uses the binary representation of 64 to represent this number. 64 is obtained by subtracting the start of the value range from the number to be encoded, that is* $152 - 88 = 64$. *The binary representation of 64 is 01000000 (using 8 bits). Hence the full representation of 152 is 1100 01000000. Finally, the full representation of the ORDPATH label 1.152 is 01 001 1100 01000000 (the spaces are present for legibility only).*

The ORDPATH prefix-free label storage schemes often require less bits to represent a label that the UTF-8 control token scheme - recall the label 1.152 requires 24-bits to be represented in UTF-8 but only 17 bits using ORDPATH prefix-free codes. However, the ORDPATH prefix free codes have higher computational costs in order to decode a label.

2.6.5 Critique of Label Storage Schemes

In this section, we outlined the four approaches underlying the implementation of all existing label storage schemes for XML to date: length fields, control tokens, separators, and prefix-free codes. No single approach stands out, each has their own advantages and limitations. Fixed length fields are ideal for static data and variable length fields are ideal for data that is rarely updated. Control token label storage schemes *may* facilitate fast byte-by-byte label comparison operations if the properties of the labeling scheme is designed to take advantage of such operations. However the control token storage schemes proposed to date are not compact. The separator label storage schemes offer compact label encoding however, the entire label must be decoded bit-by-bit in order to identify each individual component in the label, which is computationally expensive for large labels. Lastly, prefix-free codes *may* also permit fast byte-by-byte comparisons but require a pre-computed prefix-free code table to encode and decode labels and a more complex encode/decode function that leads to higher label comparison computational costs.
To date, all bit-string dynamic labeling schemes employ either a length field label storage scheme or a separator label storage scheme. Of these two approaches, only the separator label storage scheme can overcome the overflow problem presented in §2.3.6. However, the separator label storage scheme cannot encode binary-encoded bit-string labels. Hence, there does not exist a scalable label storage scheme for binary-encoded bit-string labels that can completely avoid the relabeling of nodes under any arbitrary node insertion and deletion scenario. All existing length field label storage schemes are subject to the overflow problem. Furthermore, there does not exist to date a control token or prefix-free label storage scheme for bit-string dynamic labeling schemes.

2.7 Summary

In this Chapter, we presented a critical analysis of related work in the area of dynamic labeling schemes for XML. We began with a discussion of the various approaches to capturing document order and identified the hybrid approach as striking a balance between the strengths and weaknesses of global and local order.

Containment labeling schemes were presented and their underlying properties outlined and their limitations discussed. The key limitation undermining containment labeling schemes, from the perspective of XML updates, is the requirement to relabel all *following* nodes and their descendants after a new node has been inserted into the XML document. Several extensions were proposed but they merely defer the problem but do not overcome it.

We then focused on prefix labeling schemes which are more update friendly than containment labeling schemes and we described in detail five dynamic labeling schemes that are representative of the key prefix labeling schemes proposed to date. All of the prefix labeling schemes are subject to the overflow problem and require existing labels to be relabeled after a certain number of frequent node insertions.

We subsequently presented several orthogonal dynamic labeling schemes that may be deployed as either a prefix or containment labeling scheme. The QED labeling scheme overcomes the overflow problem through the novel use of quaternary encoding and a separator label storage scheme. However, label sizes grow rapidly in the presence of dynamic insertions and QED has high computational costs and large memory requirements when initially assigning labels to an XML document. The Vector and DDE label sizes grow rapidly under frequent insertions and both are subject to the overflow problem. The EXEL labeling scheme generates labels that are not compact and the Enhanced EXEL labeling scheme is subject to the overflow problem.

We then reviewed and illustrated using examples, the limitations of all dynamic labeling schemes supporting the reuse of deleted node labels, and demonstrated that all existing approaches offer incomplete solutions.

Lastly, we reviewed all four categories of label storage schemes and highlighted their benefits and limitations. We demonstrated that all existing length field label storage schemes are subject to the overflow problem, and identified to date that there does not exist a control token or prefix-free code label storage scheme for bit-string dynamic labeling schemes.

To summarize, there are four major problems outstanding in the provision of a dynamic bit-string node labeling scheme for XML: compact labels, scalability, label reuse and label storage.

1. **Label Reuse**. The dynamic labeling scheme should support the reuse of deleted node labels in a dynamic environment. The provision of deleted node label reuse ensures that new (longer) labels are not unnecessarily created when shorter deleted node labels are available for reuse.

2. **Compact Labels**. The dynamic labeling scheme must ensure the label size will have a highly constrained growth rate both at initial document creation and after subsequent or repeated node insertions.

3. **Scalability**. The dynamic labeling scheme must never require the relabeling of existing nodes after any arbitrary combination of node insertions or deletions.

4. **Label Storage**. The dynamic labeling scheme employs a label storage scheme that supports the scalability property.

In Chapter 3, we identify and present a template of properties that are representative of a *good* dynamic labeling scheme for XML. This is followed by an evaluation framework by which all new and existing dynamic labeling schemes may be evaluated.

Chapter 3

Desirable Properties of Dynamic Labeling Schemes

In the previous Chapter, we provided a comprehensive literature review and critical analysis of the state-of-the-art in dynamic node labeling schemes for XML. Each dynamic labeling scheme proposed to date offers a unique set of characteristics presenting distinct advantages and limitations with respect to one another, and thus, demonstrates a wide range of open issues in this area. Our first goal was to focus on research into those properties which are most important to a labeling scheme. To achieve this goal, we identify a set of properties that are representative of the characteristics of a *good* dynamic labeling scheme for XML. In §3.1, we introduce the template of properties and in §3.2 we present an evaluation framework and provide in-depth analysis of our findings.

3.1 Aims for a Good Dynamic Labeling Scheme

Each dynamic labeling scheme to date was constructed to advance a particular feature or to overcome a particular limitation and the distinguishing characteristic was designed, often without consideration for the impact their solution had on other key properties. To the best of our knowledge, no comprehensive analysis of existing dynamic labeling schemes has been performed with a view to identifying the key characteristics of a holistic and *good* dynamic labeling scheme for XML, the number and specification of the core properties it should exhibit and the essential requirements it should satisfy. The output of such an analysis will be a template of properties that should form a key component in the design and specification of all new dynamic labeling schemes. Furthermore, the template of properties should constitute the principle component of an evaluation framework of metrics by which all new and existing dynamic labeling schemes can be evaluated [46]. In this section, the template of properties are introduced and described briefly.

- **Document Order.** *This property indicates the approach adopted by the labeling scheme to maintain document order. The three approaches to document order are local order, global order and hybrid order.* The maintenance of document order is an intrinsic property and an essential requirement of the XML data model. Document order was introduced in §2.1 and the advantages and limitations of the various approaches were outlined.

- **Encoding Representation.** *This property indicates the label storage scheme employed by the dynamic labeling scheme: length fields (LF), control tokens (CT), Separators (S) and Prefix-free codes (PF).* The label storage schemes were reviewed and described in §2.6.

- **XPath Evaluations.** *The value of a node label permits the evaluation of the ancestor-descendant, parent-child and sibling-based relationships.* Enabling the evaluation of the above relationships from the node labels alone contributes significantly to an efficient query and update service. Structural relationships between nodes may be determined by processing the labels directly in memory (from indices of labels loaded into memory) and thus avoiding a scan of the XML document on the filesystem or a table lookup in a database, which are expensive operations.

- **Division Computation.** *The labeling scheme is* ***not*** *required to perform division computations when initially assigning labels to the XML tree or during an update operation.* Division computations may lead to floating-point errors when processing very large numbers.

- **Recursive labeling Algorithm.** *The labeling scheme does **not** employ a recursive algorithm to compute and assign labels during the initial construction of the XML tree.* The use of a recursive algorithm is more computationally expensive because it requires multiple passes of the XML tree. A recursive algorithm also has significantly higher memory requirements because each parent node must first generate the table of encoded labels for all child nodes before the labels can be assigned to them.

- **Orthogonal Labeling Scheme.** *The labeling scheme may be applied as both a containment labeling scheme and a prefix labeling scheme.* This is an important property, because the database designer and data modeler should have the freedom to choose the appropriate approach to determine structural relationships (parent-child, ancestor-descendant, sibling-order, and so on). The appropriate approach (prefix or containment) is selected according to the type of data to be stored and the type of queries and updates to be performed. Also, new advances in query processing and query optimization are being made all the time. A dynamic labeling scheme should ideally be *structurally agnostic* (that is to say, orthogonal [68] [35] [67]) so that it may be used by a wide variety of encoding schemes and consequently facilitate new and existing query processing and query optimization strategies while maintaining its dynamic properties.

- **Compact Initial.** *The labeling scheme generates compact labels when a document tree is initially assigned labels for the first time.* Under all existing dynamic labeling schemes, future node insertions result in labels with a size that is, at a minimum, equal to or longer than the labels assigned at document initialization. Consequently, it is important the labels assigned at document initialization be as small as possible.

- **Compact Update.** *The labeling scheme maintains a reasonably constrained label size growth rate under various node update scenarios, particularly under frequent node insertions.* There are three types of frequent node insertions possible:

  - **Frequent Random Insertions.** Frequent node insertions at random positions in the XML document.
  - **Frequent Uniform Insertions.** Frequent node insertions at uniform positions in the XML document.
  - **Frequent Skewed Insertions.** There are two types of frequent skewed insertions: Bulk Insertions or Fixed-point Insertions. Bulk Insertions are frequent node insertions between two consecutive sibling nodes such that each new node is inserted immediately after the previously inserted node. Fixed-point Insertions are frequent skewed insertions between two consecutive sibling nodes *leftNode* and *rightNode* such that each new node is inserted at a fixed point immediately after *leftNode*.

- **Overflow Problem.** *The labeling scheme is* ***not*** *subject to the overflow problem presented in §2.3.6 and consequently, completely avoids the relabeling of nodes under any arbitrary node insertion or deletion scenario.* The cost of relabeling nodes and rebuilding indices are prohibitively expensive and result in the unavailability of an efficient query and update service while the indices are under construction.

- **Reuse of Deleted Node Labels.** The labeling scheme supports the ability to reuse a deleted node label. The ability to reuse a deleted node label limits the growth rate of the label size in a dynamic environment with a moderate to heavy update load, and thus, contributes to maintaining compact label sizes.

The ten properties presented here are not intended to represent an exhaustive list of all requirements for a dynamic node labeling scheme. Instead the template of properties are indicative of the key properties that are representative of the characteristics of a holistic and *good* dynamic node labeling scheme for XML.

3.2 Evaluation Framework

In this section, we present our evaluation framework and analyze our findings. The evaluation framework is presented in Figure 3.1 and evaluates all of the prefix la-

Labelling Schemes	Document Order	Encoding Rep.	Xpath Eval.	Division Comp.	Recursion Alg.	Orthogonal	Compact Initial	Compact Update	Overflow Prob.	Label Reuse
DeweyID	**Hybrid**	**CT**	**F**	**F**	**F**	**N**	**N**	**N**	**N**	**N**
ORDPATH	**Hybrid**	**PF**	**F**	**N**	**F**	**N**	**N**	**N**	**N**	**N**
DLN	**Hybrid**	**CT**	**F**	**F**	**F**	**N**	**N**	**N**	**F**	**N**
LSDX	**Hybrid**	**LF**	**F**	**F**	**F**	**N**	**N**	**N**	**N**	**N**
ImprovedBinary	**Hybrid**	**LF**	**F**	**N**	**N**	**F**	**P**	**N**	**N**	**N**
QED	**Hybrid**	**S**	**F**	**N**	**N**	**F**	**F**	**N**	**F**	**N**
CDBS	**Hybrid**	**LF**	**F**	**N**	**N**	**F**	**F**	**N**	**N**	**N**
Vector	**Hybrid**	**CT**	**F**	**F**	**N**	**F**	**P**	**N**	**N**	**N**
DDE	**Hybrid**	**PF**	**F**	**N**	**N**	**N**	**P**	**N**	**N**	**N**
EXEL	**Hybrid**	**LF**	**F**	**F**	**F**	**F**	**N**	**N**	**N**	**N**
Enhanced EXEL	**Hybrid**	**LF**	**F**	**F**	**F**	**F**	**P**	**N**	**N**	**N**

Figure 3.1: Evaluation Framework

beling schemes and orthogonal labeling schemes presented in Chapter 2. We have omitted from our evaluation framework containment labeling schemes because they are unsuitable for use as a dynamic labeling scheme due to the significant node relabeling costs after each update. The properties in our evaluation framework are described as Full (F) compliance; Partial (P) Compliance and Non (N) compliance. It is clear from our evaluation framework that no two labeling schemes share the same properties. This is a positive finding as there is no *one size fits all* solution to the XML update problem. There is a natural tension between the requirements of a query processor and those of an update service. An important design principle that should govern the specification of a dynamic labeling scheme for XML is: it is acceptable (although not desirable) that the provision of new functionality *may adversely affect the performance of existing functionality*, but the provision of new functionality should *never reduce or eliminate existing functionality.* In that context, all of the dynamic labeling schemes in our evaluation framework provide full support for XPath evaluations of ancestor-descendant, parent-child and sibling based relationships.

In this dissertation, we focus on minimizing the label size when an XML document is initially labeled and under various node insertion and deletion scenarios. The motivation and importance of small and compact labels was discussed in §1.5. Consequently, we concentrate on the four key characteristics that directly concern label size: Label Reuse, Compact labels, Scalability and Label Storage. We now provide a detailed description of the challenges to be overcome in the provision of these characteristics.

1. **Label Reuse.**
   The dynamic labeling scheme can support the reuse of deleted node labels to ensure compact label sizes in a dynamic environment with frequent node insertions and deletions. When a node is deleted from an XML document and a new node is subsequently inserted in the same position as the deleted node, the new node label generated will have a label that is at least 1 bit longer that the label of the deleted node. ***No existing dynamic labeling scheme offers a complete solution for the reuse of deleted node labels.*** Consequently, the deleted node label is not reclaimed nor reused: it is wasted. As a result, label sizes grow faster than is necessary and larger label sizes have a negative impact on query and update performance.

2. **Compact Labels**.
   The dynamic labeling scheme can ensure the label size will have a highly constrained growth rate both at initial document creation and after subsequent and repeated node insertions. The compact labels characteristic incorporates two properties from our evaluation framework, namely Compact Initial and Compact Update.
   Compact Initial.
   Only two labeling schemes in our evaluation framework, namely QED and CDBS, generate compact labels when an XML document is labeled for the first time. However, CDBS employs a length field label storage scheme that is subject to the overflow problem after a certain number of updates and consequently requires all existing labels to be relabeled. QED is not subject to the overflow problem but does not generate compact labels under frequent node insertions. Also, both QED and CDBS are unable to reuse deleted node labels

and hence, unnecessarily generate larger labels even though smaller deleted labels may be available for reuse. Furthermore, both QED and CDBS use recursive algorithms and division operations which result in higher computation costs and significantly larger memory requirements compared to the labeling schemes that do not employ these approaches.

Compact Update.

No existing dynamic labeling scheme maintains a highly constrained label size growth rate in the presence of dynamic updates, and in particular, under frequently skewed insertions. All bit-string dynamic labeling schemes (ImprovedBinary, QED, CDBS, EXEL, Enhanced EXEL) have a best-case *linear growth rate* of node label size under frequently skewed insertions. Therefore, the label size grows quickly. After 1000 node insertions, the 1000^{th} node label will be at least 1000 bits long. After 1 million node insertions, the 1 millionth node label will be at least 1 million bits long. All other dynamic labeling schemes (they use numeric or alphanumeric symbols in their labels) are subject to the overflow problem (with the exception of DLN) and require all nodes to be relabeled after a certain number of updates. The DLN labeling scheme is not subject to the overflow problem but does not generate compact labels at document initialization. Also, DLN label sizes grow rapidly in the presence of frequently skewed insertions.

3. **Scalability**.

   The dynamic labeling scheme will never require the relabeling of existing nodes after any arbitrary combination of node insertions and deletions. There are only two reasons that cause a dynamic labeling scheme to relabel nodes. The first reason is that the node insertion algorithms of the dynamic labeling scheme do not permit arbitrary node insertions without relabeling. For example, when a new node is inserted into an XML tree, the DeweyID labeling scheme (presented in §2.3.1) will require the relabeling of any *following-sibling* nodes (and their descendants). The second reason that causes a dynamic labeling scheme to relabel nodes is because it employs a label storage scheme subject to the overflow problem.

4. **Label Storage.**

   The dynamic labeling scheme employs a label storage scheme that is not subject to the overflow problem. The label storage characteristic incorporates the encoding representation property of our evaluation framework. A bit-string dynamic labeling scheme may only employ a length field or separator label storage scheme because all existing control token and prefix-free code label storage schemes encode numeric or alphanumeric values. The only bit-string dynamic labeling scheme to date that overcomes the overflow problem is the quaternary-encoded bit-string QED labeling scheme which employs the separator label storage scheme. However, the separator label storage scheme cannot cannot encoded binary-encoded bit-string labels. All existing length field label storage schemes are subject to the overflow problem. Consequently, there is no binary-encoded bit-string dynamic labeling scheme proposed to date that overcomes the overflow problem because there is no scalable label storage scheme for binary-encoded bit-string labels.

Summary of Results

The QED and DLN labeling schemes are the only dynamic labeling schemes to date to support characteristic 3 (Scalability) and characteristic 4 (Label Storage). **However, there is no dynamic labeling scheme proposed to date that supports characteristic 1 (Label Reuse) or characteristic 2 (Compact Labels).** *There is also no binary-encoded bit-string dynamic labeling scheme proposed to date that supports any one of characteristics 1, 2, 3 or 4. The aim of this dissertation is to present a dynamic labeling scheme for XML that supports all four characteristics.*

3.3 Summary

In the previous Chapter, we performed a comprehensive analysis of existing dynamic labeling schemes and in this Chapter we extrapolated from those labeling schemes, the core properties that are characteristic of a holistic and *good* dynamic labeling scheme for XML. With the aid of our Evaluation Framework, we now set the scope of this dissertation to focus on four key characteristics that directly concern label size: 1) Label Reuse; 2) Compact Labels; 3) Scalability; and 4) Label Storage. These four problems will be addressed in Chapters 4 through 7 in this dissertation.

The provision of label reuse functionality is a difficult challenge and in the previous Chapter we outlined the various attempts to overcome this problem. In the next Chapter, we describe in detail the non-trivial nature of enabling support for the reuse of deleted node labels and we present our solution.

Chapter 4

Reuse of Deleted Node Labels

In the previous Chapter, we identified the desirable properties that are characteristic of a *good* dynamic labeling scheme for XML. Using these properties, we constructed an evaluation framework by which all new and existing dynamic labeling schemes can be evaluated. In particular, we highlighted four key characteristics that directly concern label size. In this Chapter, we focus on one characteristic - label reuse - the ability to reuse deleted node labels. This Chapter begins with a description of the problem we are trying to solve in §4.1, and in §4.2, we introduce some concepts and terminology which we will use throughout the remainder of this dissertation. In §4.3, we identify two core attributes that facilitate the reuse of deleted node labels and in §4.4, we introduce our EBSL dynamic labeling scheme which incorporates the attributes necessary to facilitate deleted node label reuse. In §4.5 we present our EBSL node insertion algorithms that support the reuse of deleted node labels in a dynamic environment. Finally in §4.6 we present an evaluation of the EBSL dynamic labeling scheme and highlight the benefits and limitations of EBSL.

4.1 Problem Description

All dynamic node labeling schemes for XML published to date are not truly dynamic in that they support updates in the form of node insertions only. When a node is deleted, the node label is *marked* as deleted. Subsequently, if we want to insert a new node at the same position in the XML tree as the previously deleted node, a new node label is generated. The deleted node label is not reused and is thus, wasted. In an online transaction processing environment with frequent node insertions and deletions, the inability to reuse deleted node labels leads to a rapid increase in label sizes. Large node label sizes result in slower label comparison operations and consequently lead to slower query evaluations and poor update performance.

To incorporate deleted node label reuse as a property of a dynamic labeling scheme is a non-trivial task. Between any two consecutive nodes, there may have been an arbitrary number of node deletions. **The ability to detect, identify and reclaim a deleted node label must be provided from the information encoded in the label alone.** The labeling scheme must not rely on external indices to keep track of nodes as they are deleted and inserted. Furthermore, it is not sufficient to reclaim deleted node labels simply because they exist. A deleted node label may be longer in size than a newly generated label. In such scenarios, it may be preferable to generate and insert the smaller node labels first and to only reuse the larger labels when no smaller labels are available. Consequently, it is necessary to identify the deleted label and determine the size of the deleted label in order to determine if it is suitable for reuse in the current update operation. All of this functionality must be provided from the information encoded in the labels alone while maintaining document order and guaranteeing that all nodes labels are unique whether they are newly generated or recently reused.

4.2 Concepts and Terminology

Before proceeding with this Chapter, it is important to clarify some concepts and terminology that we will use throughout the remainder of this dissertation.

Document Lifecycle

There are conceptually three stages in the lifecycle of an XML document. The first stage is when a document is initially created. The second stage is when a document is updated by adding (inserting) new elements or attributes. The third stage is when the document is updated by deleting existing elements or attributes (and subsequently optionally inserting new elements or attributes). These three stages correspond to the specification of three algorithms that reflect the initial construction of the XML tree, the initial insertion of new nodes in the XML tree and finally the ability to dynamically detect, identify and reclaim deleted node labels when performing new node insertions. The three algorithms may be listed as follows:

1. The `AssignInitialLabels` Algorithm.

   This algorithm is the labeling algorithm that takes as input an XML tree (a sequence of nodes) and outputs a labeled XML tree in which each node has been assigned a unique and ordered label.

2. The `SimpleInsertion` Algorithm.

   This algorithm takes as input the labeled XML tree outputted by the Algorithm `AssignInitialLabels` and inserts a new node at the requested position in the XML tree. The Algorithm `SimpleInsertion` assumes no nodes have been deleted. However, new nodes may have already been inserted by the `SimpleInsertion` Algorithm previously.

3. The `InsertionWithReuse` Algorithm.

   This algorithm will identify, reclaim and reuse a deleted node label if one is available at the position of insertion. Otherwise, a new node label will be generated using the `SimpleInsertion` Algorithm.

Node Insertions. Every node insertion is considered to be an insertion between two consecutive sibling nodes where N_{left} denotes the node label on the left and N_{right} denotes the node label on the right. In a bit-string dynamic labeling scheme, document order is maintained using lexicographical order. N_{left} is always lexicographically less than N_{right}. We denote *lexicographically less than* with the symbol $\prec$ and we denote *lexicographically greater than* with the symbol $\succ$. Thus, $N_{left} \prec N_{right}$ means N_{left} is lexicographically less than N_{right} and $N_{right} \succ N_{left}$ means N_{right} is lexicographically greater than N_{left}. When inserting a new node to the right of the current rightmost node, N_{left} is said to be not empty (N_{left} has the label of the current rightmost node) and N_{right} is empty. The new node to be inserted is always denoted as N_{new}. Finally, in our algorithms enabling node insertions and deletions, we use the symbol $\oplus$ to denote the concatenation of two binary strings.

4.3 Attributes Enabling Deleted Node Label Reuse

It is our view that all existing approaches to enable the reuse of deleted node labels offer incomplete solutions because they do not generate node labels in a *deterministic* manner and the dynamic labeling schemes are not *consistent*. In this section, we introduce and define the new attributes *deterministic* and *consistent* with respect to node labeling schemes and then demonstrate how the absence of these attributes forms the underlying reason why all existing approaches to solving the deleted node label reuse problem have failed.

Definition 4.1. ***(Deterministic Dynamic Labeling Scheme)***
The value of a node label generated by a deterministic dynamic labeling scheme is a function of the value of the node label(s) immediately adjacent to it.

When an XML tree is labeled initially, the first child node is labeled according to some predefined function. A deterministic `AssignInitialLabels` Algorithm generates labels for the second and subsequent child nodes based on the value of the node label immediately preceding it. In a similar manner, a deterministic `SimpleInsertion` Algorithm and a deterministic `InsertionWithReuse` Algorithm when performing node insertions generates the new node label based on the values of the node labels immediately adjacent to the position of insertion. That is, the node label N_{new} is generated based on the values of node labels N_{left} and N_{right} such that $N_{left} \prec N_{new} \prec N_{right}$.
Assume a dynamic labeling scheme is not *deterministic*. After an arbitrary number of node deletions between two consecutive siblings nodes N_{left} and N_{right}, for any subsequent node insertions between N_{left} and N_{right} the labeling scheme is unable to identify (and consequently unable to reuse) the sequence of deleted node labels lexicographically ordered between N_{left} and N_{right}. The Extended QED labeling scheme and the CDBS labeling scheme employ `AssignInitialLabels` Algorithms that are not deterministic because they are recursive algorithms. These recursive

algorithms use a mathematical `round` function as part of their label assignment process. The `round` function can generate the same output for two different inputs and consequently is not *deterministic.*

Definition 4.2. ***(Consistent Dynamic Labeling Scheme)***
A dynamic labeling scheme is consistent iff:

1. *Its `AssignInitialLabels` Algorithm, `SimpleInsertion` Algorithm and `InsertionWithReuse` Algorithm are all deterministic algorithms, and*

2. *All three algorithms generate the same label when inserting a new rightmost child sibling node, and*

3. *The `SimpleInsertion` and `InsertionWithReuse` Algorithms generate the same label when inserting a new leftmost child sibling node.*

The initial assignment of labels to the second and subsequent sibling nodes **is a functionally identical operation** to an insertion of a new node after the current rightmost node. For example, if node 110 in Figure 4.1 were the rightmost sibling node, then the node label 1110 would be generated by a deterministic `SimpleInsertion` Algorithm and by a deterministic `AssignInitialLabels` Algorithm.

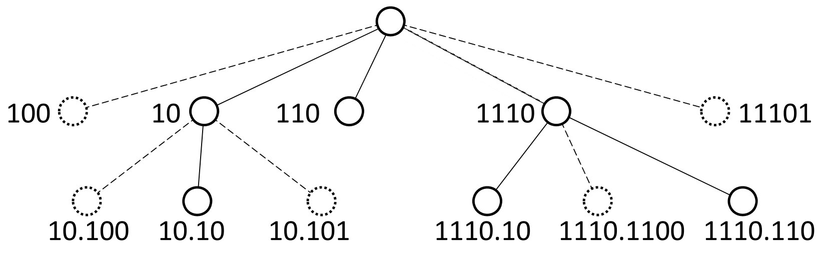


Figure 4.1: IBSL Labeled XML Tree

In order for a dynamic labeling scheme to fully guarantee the reuse of a deleted node label, it must be a consistent dynamic labeling scheme. The functionality of a deterministic `AssignInitialLabels` Algorithm when initially labeling the second and subsequent nodes must be identical to functionality performed by the deterministic `SimpleInsertion` and `InsertionWithReuse` Algorithms when inserting a new node to the right of the current rightmost node. However, all four dynamic labeling schemes that claim to support the reuse of deleted node labels (IBSL [32], Extended EXEL [44], Extended QED [37], and V-CDBS [38]) are not consistent dynamic labeling schemes because the node labels assigned by all four labeling schemes when initially labeling the second and subsequent nodes at document creation are not the same as the node labels assigned by their `SimpleInsertion` and `InsertionWithReuse` Algorithms when inserting a new node to the right of the current rightmost node. Consequently, when these nodes are deleted, the `InsertionWithReuse` Algorithms has no way of identifying the deleted node label to be reused.

4.4 Enhanced Binary String Labeling (EBSL)

In the previous section, we presented the platform for a dynamic labeling scheme to support the reuse of deleted node labels. In this section, we present a dynamic labeling scheme for XML called the Enhanced Binary String Labeling scheme (EBSL) [47] to address the problem of deleted node label reuse. EBSL does not require the relabeling of existing nodes nor the recalculation of any values when inserting new nodes in an XML tree. Furthermore, EBSL fully supports the reuse of deleted node labels when inserting new nodes into positions that previously contained deleted nodes. EBSL guarantees that every deleted node label can be reused. That is to say, there are no (simple or complex) insertion/deletion scenarios that will result in a deleted node label remaining unused when it would be appropriate to reuse that label. EBSL may be deployed as a prefix labeling scheme and thus, supports ancestor-descendant, parent-child and sibling-ordered XPath evaluations.

Roadmap for this section. In this section, let us now proceed with the core functionality necessary to delivery a dynamic labeling scheme for XML. We present the EBSL label assignment and insertion algorithms which will use the two attributes when generating new node labels *when no deleted node labels are available.* This is necessary in order to specify clearly how new node labels are generated when no deleted node labels are available for reuse. In the next section §4.5, we will present our node insertion algorithms that fully support the reuse of deleted node labels. This section is structured as follows: In §4.4.1, we introduce our extended definition of lexicographical order, a prerequisite condition to facilitate the *deterministic* attribute (presented in §4.3). In §4.4.2, we highlight and explain why we only consider the positional identifier of a node label in our algorithms. In §4.4.3, we introduce the EBSL `AssignInitialLabels` Algorithm and in §4.4.4, we present the EBSL `SimpleInsertion` Algorithm. In §4.4.5 we introduce and define three categories of labels generated by the EBSL labeling scheme. The distinct characteristics of the three categories of labels ensure EBSL is a *consistent* dynamic labeling scheme. The *consistent* attribute (presented in §4.3) is the core component enabling support for the reuse of deleted node labels. Finally, in §4.4.6, we present several theorems and proofs to substantiate our claims.

4.4.1 e−Lexicographical Order

EBSL compares node labels using lexicographical order and not numerical order. Our extended definition of lexicographical order differs from existing approaches [34] [22] [43] [4] [68] [31] [38] [32]. EBSL employs an extended definition of lexicographical order because given an arbitrary node N_{arb}, the conventional definition of lexicographical order does not always permit the identification of the node labels to the immediately left and right of N_{arb}. The extended definition of lexicographical order will always facilitate the identification of the node labels immediately adjacent to N_{arb}. The extended definition of lexicographical order also permits the generation of new node labels in a *deterministic* manner. We now present the extended definition of lexicographical order.

Definition 4.3. *(e−Lexicographical order) Given two consecutive binary strings S_{left} and S_{right} (S_{left} represents the left binary string, S_{right} represents the right binary string), S_{left} is said to be lexicographical equal to S_{right} iff they are identical. S_{left} is said to be lexicographically less than S_{right} ($S_{left} \prec_e S_{right}$) iff:*

1. *the comparison of S_{left} and S_{right} is bit by bit from left to right. If the current*

bit of S_{left} is 0 and the current bit of S_{right} is 1, then $S_{left} \prec_e S_{right}$ and stop the comparison, or

2. *length(S_{left}) $<$ length(S_{right}), S_{left} is a prefix of S_{right}, the first extra bit of S_{right} $=$ 1 (i.e.: substring(S_{right}, len(S_{left})+1, len(S_{left})+1) $=$ 1), then $S_{left} \prec_e S_{right}$ and stop the comparison, or*

3. *length(S_{left}) $>$ length(S_{right}), S_{right} is a prefix of S_{left}, the first extra bit of S_{left} $=$ 0 (i.e.: substring(S_{left}, len(S_{right})+1, len(S_{right})+1) $=$ 0), then $S_{left} \prec_e S_{right}$ and stop the comparison.*

Under the conventional definition of lexicographical order, a prefix string is lexicographically less than the larger string beginning with that prefix (e.g.: 110 $\prec$ 11001). In our definition of lexicographical order, (condition 3) the larger string containing the prefix is lexicographical less than the prefix string if and only if the subsequent bit immediately after the prefix in the larger string is a 0 bit (e.g.: 11001 $\prec_e$ 110). Conversely, (condition 2) the larger string containing the prefix is lexicographical greater than the prefix string if and only if the subsequent bit immediately after the prefix in the larger string is a 1 bit (e.g.: 110 $\prec_e$ 11010). *Throughout the remainder of this Chapter, all references to lexicographical order indicates e−lexicographical order.*

4.4.2 Self_label: A Positional Identifier

Before we present our label assignment algorithms, we must highlight the importance of the *positional identifier* component of a prefix label. In a prefix labeling scheme, the label of a node in the XML tree consists of the parent's label concatenated with a delimiter and a positional identifier of the node itself. **The positional identifier, hereafter we refer to as the self_label, is a label encoding the position of the node relative to its sibling nodes.** Given a parent node P, the insertion of new child nodes of P or the deletion of existing child nodes of P will never affect the label of P itself. *Consequently, when we consider identifying, reclaiming and reusing a deleted node label, we consider only identifying, reclaiming and reusing the self_label portion of the node label.* The parent's label concatenated with a delimiter defines the unique path vector from the root node of the tree to the parent node and is already fixed by virtue of the hierarchical location of the node in the tree.

4.4.3 The `AssignInitialLabels` Algorithm

Algorithm 4.1: EBSL `AssignInitialLabels`

```
   /* The purpose of this algorithm is to assign unique labels to all child nodes of
      parent node P.                                                                 */
   input  : A parent node P.
   output: A unique EBSL label for each child node of parent node P.
 1 begin
 2     if (P is the root node) then
 3         prefix_label ⟵ ∅;
 4     else
 5         prefix_label ⟵ label(P);
 6     end
 7     self_label[0] ⟵ 0;
 8     for (i=1; i ≤ P.numberOfChildren; i++) do
           /* P.numberOfChildren denotes the number of child nodes of parent node P.    */
 9         self_label[i] ⟵ 1 ⊕ self_label[i − 1] ;          // ⊕ denotes concatenation
           /* self_label[i] denotes the self_label of the i^th child node.              */
10         label[i] ⟵ prefix_label ⊕ delimiter ⊕ self_label[i];
11     end
12 end
```

Algorithm 4.1 is the EBSL `AssignInitialLabels` Algorithm and assumes a prefix labeling scheme. The algorithm takes as input a parent node, and assigns a unique label to every child node of the parent. The first child is always assigned the self_label 10. Thereafter, all subsequent children are deterministically labeled such that the self_label of child i is computed as the concatenation of a 1 bit and the self_label of child $i - 1$. Algorithm 4.1 may be applied recursively to the XML tree to assign labels to every node in the tree. The EBSL `AssignInitialLabels` Algorithm is functionally identical to the IBSL `AssignInitialLabels` Algorithm [32].

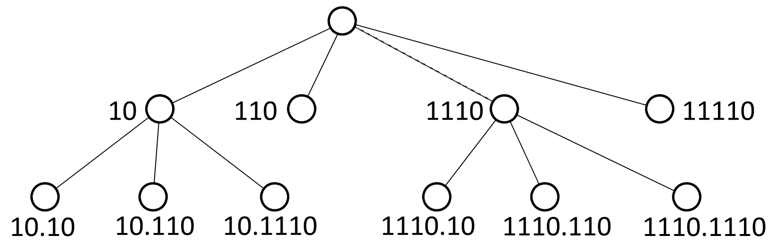


Figure 4.2: An EBSL Tree Labeled by the `AssignInitialLabels` Algorithm

An example of an EBSL labeled tree is illustrated in Figure 4.2.

4.4.4 The `SimpleInsertion` Algorithm

The EBSL **`SimpleInsertion`** Algorithm takes as input two node labels, N_{left} and N_{right}, and generates a new node label N_{new} such that $N_{left} \prec_e N_{new} \prec_e N_{right}$. The **`SimpleInsertion`** Algorithm assumes no nodes have been deleted in the XML tree. This assumption is important so as to permit the clear specification of how to generate a new node label between two existing consecutive node labels when no deleted labels are available to be reused.

Algorithm 4.2: EBSL `SimpleInsertion`

```
input  : left self_label $N_{left}$; right self_label $N_{right}$.
output: New self_label $N_{new}$ such that $N_{left} \prec_e N_{new} \prec_e N_{right}$.
1 begin
2     Case 1: $N_{left}$ is empty but $N_{right}$ is not empty
      /* Insert a new node before the current leftmost node.            */
3     $N_{new} \longleftarrow N_{right} \oplus 0$ ;                 // $\oplus$ means concatenation.
4     Case 2: $N_{left}$ is not empty but $N_{right}$ is empty
      /* Insert a new node after the current rightmost node.            */
5     $N_{new} \longleftarrow 1 \oplus N_{left}$;
6     Case 3: $N_{left}$ is not empty and $N_{right}$ is not empty
      /* Insert a new node between two existing nodes.                  */
7     if (len($N_{left}$) $\leq$ len($N_{right}$)) then $N_{new} \longleftarrow N_{right} \oplus 0$;
8     else if (len($N_{left}$) $>$ len($N_{right}$)) then $N_{new} \longleftarrow N_{left} \oplus 1$;
9 end
```

It should also be noted that although cases 1 and 3 of our **`SimpleInsertion`** Algorithm is the same as the IBSL **`SimpleInsertion`** Algorithm, case 2 is different. Concerning case 2, the IBSL **`SimpleInsertion`** Algorithm assigns $N_{new} = N_{left} \oplus 1$. The EBSL **`SimpleInsertion`** Algorithm case 2 assigns $N_{new} = 1 \oplus N_{left}$. This change fundamentally distinguishes the EBSL labeling scheme from the IBSL labeling scheme in a dynamic scenario, because new node insertions to the right of the current rightmost node will now end in a 0 bit, and not a 1 bit. *This will directly influence lexicographical order evaluations and consequently the label values of new node inserted after the rightmost node.* For example, given the rightmost sibling node N_{left} = 11110, the IBSL **`SimpleInsertion`** Algorithm, case 2 generates N_{new} = 111101, whereas the EBSL **`SimpleInsertion`** Algorithm, case 2 generates N_{new} = 111110,

4.4.5 EBSL Label Categories

Every label assigned by the EBSL labeling scheme falls into one of three label categories: $\Re label$, $\ell label$ or $\aleph label$. The distinct characteristics of the three categories of labels ensure EBSL is a *consistent* dynamic labeling scheme. The *consistent* attribute is the core component facilitating support for the reuse of deleted node labels.We now define each of the label categories and then explain their importance.

Definition 4.4. *($\Re label$): An $\Re label$ is a binary string with a minimum length of 2 bits such that there is only one 0 bit in the string and the 0 bit is the last (rightmost) bit in the string.*

Examples of an $\Re label$ are 10, 110, 1110, 11110, 111110 and so on. All $\Re label$ node labels are lexicographically greater than or equal to the first child self_label 10. The `AssignInitialLabels` Algorithm always inserts a new node to the right of the current rightmost node, and consequently always assigns an $\Re label$. The `SimpleInsertion` Algorithm case 2 also inserts a new node to the right of the current rightmost node, and consequently always assigns an $\Re label$.

Definition 4.5. *($\ell label$) An $\ell label$ is a binary string with a minimum length of 3 bits such that there is only a single 1 bit in the string and the 1 bit occurs at the beginning of the string.*

Examples of an $\ell label$ are 100, 1000, 10000, 100000, 1000000 and so on. All $\ell label$ node labels are lexicographically less than the first child node 10. The `SimpleInsertion` Algorithm case 1 always inserts a new node to the left of the current leftmost node, and consequently always assigns an $\ell label$.

Definition 4.6. *($\aleph label$) An $\aleph label$ is a binary string with a minimum length of 4 bits such that it contains at least two 1 bits and two 0 bits. (An $\aleph label$ is any valid EBSL label that does not have the properties of an $\Re label$ or an $\ell label$).*

Examples of an $\aleph label$ are 1001, 10011, 10010, 1100, 11001, 11000. The `SimpleInsertion` Algorithm case 3 always inserts a new node between two consecutive sibling nodes, and consequently always assigns an $\aleph label$.

We provide an illustration of an EBSL labeled tree in Figure 4.3 with nodes *a* through *f* assigned by the `AssignInitialLabels` Algorithm and nodes *g* through *m* inserted in alphabetical order using the `SimpleInsertion` Algorithm. Nodes *a*, *b*, *c* and *i* are examples of $\Re label$. Node *g* is an example of an $\ell label$. Node *h* is an example of an $\aleph label$.

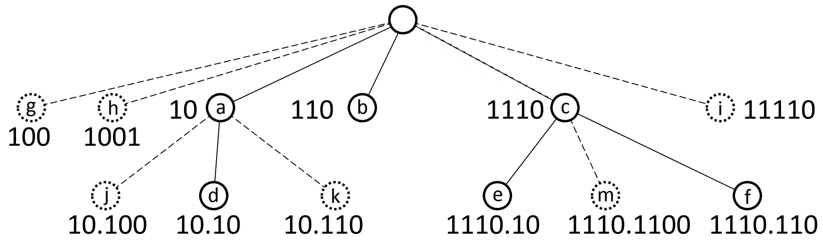


Figure 4.3: An EBSL Labeled Tree with New Nodes Inserted (Dotted Circles) using the `SimpleInsertion` Algorithm (Algorithm 4.2)

In summary, the `SimpleInsertion` Algorithm case 1 inserts a new node N_{new} to the left of the current leftmost node such that the new node label will always have the properties of an $\ell label$. The `SimpleInsertion` Algorithm case 2 inserts a new node N_{new} to the right of the current rightmost node such that the new node label will always have the properties of an $\Re label$. The `SimpleInsertion` Algorithm case 3 inserts a new node N_{new} between two consecutive sibling nodes N_{left} and N_{right} such that the new node label will always have the properties of an $\aleph label$. The three insertion cases in the `SimpleInsertion` Algorithm are closed (i.e.: both N_{left} and N_{right} cannot be empty). The fact that the three insertions cases of the EBSL `SimpleInsertion` Algorithm each generate labels with three distinguishing characteristics is **the foundation that facilitates a complete solution for the reuse of deleted node labels** (to be presented in §4.5).

4.4.6 Theorems and Proofs

In the remainder of this section, we provide theorems and proofs to substantiate our claims that each insertion case creates labels with distinguishing characteristics.

Theorem 4.1. *If the current rightmost sibling node label is an* $\Re label$*, then the label of a new node inserted to the right of the current rightmost sibling node will* *also be an $\Re label$.*

Proof: The `SimpleInsertion` Algorithm Case 2 generates a new rightmost node label by prepending a 1 bit to the current rightmost node label. If the current rightmost node label is an $\Re label$, prepending a 1 bit to an $\Re label$ maintains the properties of an $\Re label$. □

Theorem 4.2. *If the current leftmost sibling node label is an $\ell label$, then the label of a new node inserted to the left of the current leftmost sibling node will also be an $\ell label$.*

Proof: The `SimpleInsertion` Algorithm Case 1 generates a new leftmost node label by appending a 0 bit to the current leftmost node label. If the current leftmost node label is an $\ell label$, appending a 0 bit to an $\ell label$ maintains the properties of an $\ell label$. □

Theorem 4.3. *Given two consecutive sibling node labels N_{left} and N_{right}, such that $N_{left} \prec_e N_{right}$, then the label of a new node N_{new} inserted between N_{left} and N_{right} will always be an $\aleph label$.*

Proof: (Proof by Contradiction)

- Case A: Assume the label of N_{new} is an $\Re label$.
  Given that N_{new} is an $\Re label$, we know that N_{new} contains only one 0 bit and the 0 bit occurs at the end of the label. But line 7 of the `SimpleInsertion` Algorithm Case 3 in Algorithm 4.2 is the only expression of Case 3 that creates a new label ending with a 0 bit. Therefore, the label of N_{right} used to create N_{new} must have consisted of a sequence of 1 bits in order to create the $\Re label$ N_{new}. However, a label consisting of a sequence of 1 bits is not a valid EBSL label, and therefore, cannot exist. Therefore, N_{new} can not be an $\Re label$.

- Case B: Assume the label of N_{new} is an $\ell label$.
  It is given that two node labels N_{left} and N_{right} were used to generate the label of N_{new}. We know from line 8 of the `SimpleInsertion` Algorithm Case 3 in Algorithm 4.2 if the length of label N_{left} is greater than the length of

label N_{right}, then N_{new} would end in a 1 bit and hence, N_{new} could not be an $\ell label$. However, it given that N_{new} is an $\ell label$ therefore, the length of label N_{left} must be less than the length of label N_{right}.

Also, given that N_{new} is an $\ell label$ and was generated by appending a 0 bit to N_{right}, then N_{right} must be either an $\ell label$ or label 10. But all labels lexicographically less than N_{right} (an $\ell label$ or label 10) will have a length greater than N_{right} (because according to our definition of lexicographical order, to be lexicographically less than N_{right}, a 0 bit must be appended to N_{right}).

Therefore, we have the contradiction that the length of label N_{left} must be less than the length of label N_{right}, and the length of label N_{left} must be greater than the length of label N_{right}. Therefore, our assumption that the label N_{new} is an $\ell label$ is false.

Given an $\aleph label$ is defined as any valid EBSL label that does not have the properties of an $\ell label$ or an $\Re label$ (from Definition 4.6), and given that N_{new} is neither an $\ell label$ nor an $\Re label$ (from Case A and Case B above), therefore, N_{new} must be an $\aleph label$. □

4.4.7 Summary

The `AssignInitialLabels` Algorithm **always** generates an $\Re label$. The `SimpleInse-rtion` Algorithm assumes no nodes have been deleted. From Theorem 4.2, we know the `SimpleInsertion` Algorithm Case 1 will **always** generate a new node label with the properties of an $\ell label$. From Theorem 4.1, we know `SimpleInsertion` Algorithm Case 2 will **always** generate a new node label with the properties of an $\Re label$. Finally, from Theorem 4.3, we know `SimpleInsertion` Algorithm Case 3 will **always** generate a new node label with the properties of an $\aleph label$. Thus, each distinct case of the `SimpleInsertion` Algorithm generates a node label consistently and repeatedly with distinguishing characteristics. We shall exploit this unique property of the `SimpleInsertion` Algorithm in order to support the reuse of deleted node labels in the next section.

4.5 EBSL: Reusing Deleted Node Labels

In this section, we present the node insertion algorithms that support the reuse of deleted node labels. Before we present these algorithms, it is useful to summarize what we know thus far so as to identify the conditions in which these algorithms must operate.

Lexicographical Categories. An arbitrary node label in an EBSL labeled tree will always fall into one of three lexicographical categories (Recall that the node label 10 is the first label assigned by the `AssignInitialLabels` Algorithm):

1. The node label will be lexicographically greater than ($\succ_e$) the node label 10.

2. The node label will be lexicographically less than ($\prec_e$) the node label 10.

3. The node label will be lexicographically equal to the node label 10.

Label Categories. An arbitrary node label in an EBSL labeled tree will always fall into one of three label categories: $\Re label$, $\ell label$ or $\aleph label$.

Insertion Cases. When we consider a node insertion algorithm, there are always three high level cases to be processed: insertion of a new node before the current leftmost node; after the current rightmost node; and between two consecutive sibling nodes.

Roadmap for this section. The goal of this section is to present the node insertion algorithms that support the reuse of deleted node labels such that for each insertion case, the newly inserted node label is *consistent* with the node labels assigned by the `SimpleInsertion` Algorithm under the same insertion case. This section is structured as follows: We first present the algorithm (permitting deleted node label reuse) to process node insertions before the current leftmost node (Case 1), followed by the algorithm to process node insertions after the current rightmost node (Case 2). Lastly, we present the algorithm to process node insertions between two consecutive sibling nodes (Case 3).

4.5.1 Case 1: Inserting a New Node Before the Leftmost Node

Algorithm 4.3 is the EBSL `InsertNewNodeBeforeLeftmostNode` Algorithm that supports the reuse of deleted node labels when inserting a new node before the current leftmost sibling node. Given one input node label N_{right}, Algorithm 4.3 generates a node label N_{new} such that N_{new} is the first $\ell label$ or $\Re label$ lexicographically less than N_{right}. Algorithm 4.3 will never generate an $\aleph label$. More specifically, the input self_label N_{right} may be an $\Re label$, an $\ell label$ or an $\aleph label$. If $N_{right} \prec_e$=10, then N_{new} will always be an $\ell label$. If $N_{right} \succ_e 10$, then N_{new} will always be an $\Re label$. Hence, Algorithm 4.3 is both *deterministic* and *consistent* with case 1 and case 2 of the `SimpleInsertion` Algorithm in Algorithm 4.2.

Algorithm 4.3: EBSL `InsertNewNodeBeforeLeftmostNode`

```
/* This algorithm inserts a new node before the current leftmost node $N_{right}$.      */
/* $N_{right}$ can be an $\Re label$, $\ell label$ or $\aleph label$.                   */
/* If $N_{right} \prec_e$= 10, then $N_{new}$ will always be an $\ell label$.           */
/* If $N_{right} \succ_e 10$, then $N_{new}$ will always be an $\Re label$.             */
input  : A self_label $N_{right}$ that is the current leftmost sibling node.
output: A self_label $N_{new}$ such that $N_{new} \prec_e N_{right}$.
begin
    if ($N_{right}$ is an $\ell label$) or ($N_{right}$ == 10) then
        /* $N_{new}$ is set to the $\ell label$ adjacent to $N_{right}$ such that $N_{new} \prec_e N_{right}$.   */
        $N_{new} \longleftarrow N_{right} \oplus 0$;

    /* The Following IF statement is processed when $N_{right}$ is an $\Re label \succ 10$ or when
       $N_{right}$ is an $\aleph label \succ_e 10$.                                         */
    else if (prefix of $N_{right}$ == 11) then
        positionZero $\longleftarrow$ position of first 0 bit in $N_{right}$;
        /* $N_{new}$ is set to the $\Re label$ adjacent to $N_{right}$ such that $N_{new} \prec_e N_{right}$.   */
        $N_{new} \longleftarrow$ substring($N_{right}$, 1, positionZero $-$ 2) $\oplus$ 0;

    /* The Following IF statement is processed when $N_{right}$ is an $\aleph label \prec_e 10$.   */
    else if (prefix of $N_{right}$ == 100) then
        positionOne $\longleftarrow$ position of second 1 bit in $N_{right}$;
        /* $N_{new}$ is set to the $\ell label$ adjacent to $N_{right}$ such that $N_{new} \prec_e N_{right}$.   */
        $N_{new} \longleftarrow$ substring($N_{right}$, 1, positionOne $-$ 1);
    end
    return $N_{new}$;
end
```

If $N_{right} \prec_e$= 10, there may or may not be deleted node labels available for reuse, but the node label N_{new} will always be assigned the shortest $\ell label$ lexicographically less than N_{right}, (which is also the shortest possible label lexicographically less than N_{right}). It should be observed that the shortest $\ell label$ lexicographically less than N_{right} will always be the first $\ell label$ to the immediate left of N_{right}.

If $N_{right} \succ_e 10$, then we are certain there exists at least one deleted node label to the left of N_{right} because the first self_label assigned by the `AssignInitialLabels` Algorithm is always 10 and the current rightmost node $N_{right} \succ_e 10$, therefore label 10 has been deleted. Thus, when $N_{right} \succ_e 10$, N_{new} will always reuse a deleted node label. The reused label assigned to N_{new} is selected to be the $\Re label$ to the immediate left of N_{right}, that is the longest $\Re label$ lexicographically less than N_{right}. It should be noted that the longest $\Re label$ lexicographically less than N_{right} will always be a shorter label than N_{right} (at least 1 bit shorter). For example, if N_{right} is 1110 (an $\Re label$), then the longest $\Re label$ lexicographically less than N_{right} is 110; if N_{right} is 111001 (an $\aleph label$), then the longest $\Re label$ lexicographically less than N_{right} is 110. The longest $\Re label$ lexicographically less than N_{right} will always be the first $\Re label$ to the immediate left of N_{right}. It may be observed that the algorithm did not select the smallest deleted $\Re label$ lexicographically less than N_{right} that is available for reuse. When given a sequence of k nodes to insert in an XML tree, the k nodes must be inserted in document order. The labeling scheme cannot arbitrarily decide the order in which to insert the nodes. If the algorithm selected N_{new} to be the shortest deleted $\Re label$ available, then the deleted node labels between N_{new} and N_{right} will remain unused when inserting a contiguous sequence of nodes between two consecutive nodes labels. Thus, by always selecting the $\Re label$ to the immediate left of N_{right}, Algorithm 4.3 is both *deterministic* and *consistent* with case 1 and case 2 of the `SimpleInsertion` Algorithm and consequently, ensures every deleted node label available for use at the position of insertion will be reused.
In summary, Algorithm 4.3 generates a node label N_{new} such that N_{new} is always the first $\ell label$ or $\Re label$ to the immediate left of N_{right}.

4.5.2 Case 2: Inserting a New Node After the Rightmost Node

Algorithm 4.4 is the EBSL `InsertNewNodeAfterRightmostNode` Algorithm that supports the reuse of deleted node labels when inserting a new node after the current rightmost sibling node. In essence, given one input node label N_{left}, Algorithm 4.4 generates a node label N_{new} such that N_{new} is the first $\ell label$ or $\Re label$ lexicographically greater than N_{left}. Algorithm 4.4 will never generate an $\aleph label$. Although,

Algorithm 4.4: EBSL `InsertNewNodeAfterRightmostNode`

```
/* This algorithm inserts a new node after the current rightmost node $N_{left}$.    */
/* $N_{left}$ can be an $\Re label$, $\ell label$ or $\aleph label$.                  */
/* If $N_{left} \prec_e$ 100, then $N_{new}$ will always be an $\ell label$.          */
/* If $N_{left} \succ_e$= 100, then $N_{new}$ will always be an $\Re label$.          */
input  : A self_label $N_{left}$ that is the current rightmost sibling node.
output: A self_label $N_{new}$ such that $N_{left} \prec_e N_{new}$.
begin
    if ($N_{left}$ is an $\Re label$) then
        $N_{new} \longleftarrow 1 \oplus N_{left}$ ;          // Apply SimpleInsertion algorithm, Case 2

    else if ($N_{left}$ is an $\ell label$) then
        $N_{new} \longleftarrow$ substring($N_{left}$, 1, len($N_{left}$) − 1) ;   // SimpleInsertion algorithm, Case 1

    /* The Following IF statement is processed when $N_{left}$ is an $\aleph label \succ_e$ 10.   */
    else if (prefix of $N_{left}$ == 11) then
        positionZero $\longleftarrow$ position of first 0 bit in $N_{left}$;
        /* $N_{new}$ is set to the first $\Re label \succ_e N_{left}$.                          */
        $N_{new} \longleftarrow$ substring($N_{left}$, 1, positionZero);

    /* The Following IF statement is processed when $N_{left}$ is an $\aleph label \prec_e$ 10.   */
    else if (prefix of $N_{left}$ == 100) then
        positionOne $\longleftarrow$ position of second 1 bit in $N_{left}$;
        /* $N_{new}$ is set to either label 10 or to the first $\ell label \succ_e N_{left}$.    */
        $N_{new} \longleftarrow$ substring($N_{left}$, 1, positionOne − 2);
    end
    return $N_{new}$;
end
```

the input self_label N_{left} may be an $\Re label$, an $\ell label$ or an $\aleph label$. If $N_{left} \prec_e$ 100, then N_{new} will always be an $\ell label$. If $N_{left} \succ_e$= 100, then N_{new} will always be an $\Re label$. Thus, Algorithm 4.4 is both *deterministic* and *consistent* with case 1 and case 2 of the `SimpleInsertion` Algorithm in Algorithm 4.2.

If $N_{left} \prec_e$ 10, then we are certain there exists at least one deleted node label to the right of N_{left} because the first self_label assigned by the `AssignInitialLabels` Algorithm is always 10 and the current rightmost node $N_{left} \prec_e$ 10, therefore label 10 has been deleted. If $N_{left} \succ$=10, there may or may not be deleted node labels available, but the node label N_{new} will always be assigned the shortest $\Re label$ lexicographically greater than N_{left}. It should be noted that the shortest $\Re label$ lexicographically greater than N_{left} is also the shortest possible label lexicographically greater than N_{left}. Consequently, there may exist a deleted $\aleph label$ node label immediately adjacent to and lexicographically greater than N_{left}, but it will be inappropriate to reuse it because there will always be shorter $\Re label$ lexicographically greater than N_{left} available for use (or reuse).

In summary, Algorithm 4.4 generates a node label N_{new} such that N_{new} is always the first $\ell label$ or $\Re label$ to the immediate right of N_{left}.

4.5.3 Case 3: Inserting New Node Between Two Consecutive Nodes

Algorithm 4.5 is the EBSL `InsertNewNodeBetweenTwoConsecutiveNodes` Algorithm to insert a new node between two consecutive sibling nodes that supports the reuse of deleted node labels.

There are only six possible scenarios to consider and Algorithm 4.5 processes all six scenarios in the following order:

1. Regardless of the labels of N_{left} and N_{right}, if there is an $\ell label$ or $\Re label$ available for reuse between N_{left} and N_{right}, then Algorithm 4.5 reuses the first $\ell label$ or $\Re label$ lexicographically greater than N_{left}. We could have chosen to select the smallest $\ell label$ or $\Re label$ between N_{left} and N_{right} but that would ensure the node labels between N_{left} and the shortest $\ell label$ or $\Re label$ will remain unused when inserting a contiguous sequence of nodes between two consecutive node labels. Consequently, we always select the first $\ell label$ or $\Re label$ lexicographically greater than N_{left} to guarantee every deleted node label can be reused. In the remaining five scenarios, we know there does not exist an $\ell label$ or $\Re label$ available for reuse between N_{left} and N_{right}.

2. If N_{left} and N_{right} are both an $\Re label$, then Algorithm 4.5 will always reuse the smallest possible label between N_{left} and N_{right}, or generate the smallest possible label between N_{left} and N_{right} if no deleted node label exists.

3. If N_{left} is an $\Re label$ and N_{right} is an $\aleph label$, then Algorithm 4.5 will always reuse the smallest label between N_{left} and N_{right}.

4. If N_{left} is an $\ell label$ and N_{right} is an $\ell label$ or label 10, then Algorithm 4.5 will always reuse the smallest possible label between N_{left} and N_{right}, or generate the smallest possible label between N_{left} and N_{right} if no deleted node label exists.

Algorithm 4.5: EBSL `InsertNewNodeBetweenTwoConsecutiveNodes`

```
/* Insert a new node between two consecutive sibling nodes $N_{left}$ and $N_{right}$.   */
/* $N_{left}$ and $N_{right}$ can be any one of {$\Re label$, $\ell label$, $\aleph label$}.   */
input  : A self_label $N_{left}$, the left sibling node.
         A self_label $N_{right}$, the right sibling node.
output: A self_label $N_{new}$ such that $N_{left} \prec_e N_{new} \prec_e N_{right}$.
begin
    $N_{next}$ = InsertNewNodeAfterRightmostNode($N_{left}$);
    if ($N_{next} \prec_e N_{right}$) then
        /* $N_{next}$ is the $\ell label$ or $\Re label$ adjacent to $N_{left}$ such that $N_{left} \prec_e N_{next}$.   */
        $N_{new} \longleftarrow N_{next}$;

    else if ($N_{left}$ is an $\Re label$) and ($N_{next} == N_{right}$) then
        /* $N_{right}$ is the $\Re label$ adjacent to $N_{left}$ such that $N_{left} \prec_e N_{right}$.   */
        $N_{new} \longleftarrow$ SimpleInsertion($N_{left}$, $N_{right}$);

    else if ($N_{left}$ is an $\Re label$) and ($N_{next}$ != $N_{right}$) then
        /* Therefore, $N_{right}$ is an $\aleph label$ and at least 2 bit longer than $N_{left}$.   */
        $N_{temp} \longleftarrow$ substring($N_{right}$, len($N_{left}$) + 1, len($N_{right}$));
        if AllBitsAreZero ($N_{temp}$) then
            $N_{new} \longleftarrow$ SimpleInsertion($N_{left}$, $N_{right}$);
        else
            positionOne $\longleftarrow$ position of first 1 bit in $N_{temp}$;
            zeroSequence $\longleftarrow$ substring($N_{temp}$, 1, positionOne $-$ 1);
            $N_{prefix} \longleftarrow N_{left}$ with last 0 bit removed;
            $N_{new} \longleftarrow N_{prefix} \oplus 1 \oplus$ zeroSequence;
        end

    else if ($N_{left}$ is an $\ell label$) and ($N_{next} == N_{right}$) then
        /* $N_{right}$ is $\ell label$ (or label 10) adjacent to $N_{left}$ such that $N_{left} \prec_e N_{right}$.
           */
        $N_{new} \longleftarrow$ SimpleInsertion($N_{left}$, $N_{right}$);

    else if ($N_{left}$ is an $\ell label$) and ($N_{next}$ != $N_{right}$) then
        /* Therefore, $N_{left}$ is a prefix of $N_{right}$; $N_{right}$ is an $\aleph label$; $N_{right}$ is at
           least 1 bit longer than $N_{left}$.   */
        $N_{temp} \longleftarrow$ substring($N_{right}$, len($N_{left}$) + 2, len($N_{right}$));
        countZeros $\longleftarrow$ number of consecutive 0 bits at beginning of $N_{temp}$;
        if (countZeros == len($N_{temp}$)) then
            $N_{new} \longleftarrow$ SimpleInsertion($N_{left}$, $N_{right}$);
        else
            $N_{new} \longleftarrow$ substring($N_{right}$, len($N_{left}$) + 1, countZeros);
        end

    else if ($N_{left}$ is an $\aleph label$) then
        $N_{new} \longleftarrow$ InsertAfterNlabel($N_{left}$, $N_{right}$);
    end
    return $N_{new}$;
end
```

5. If N_{left} is an $\ell label$ and N_{right} is an $\aleph label$, then Algorithm 4.5 will always reuse the smallest label between N_{left} and N_{right}.

6. Lastly, if N_{left} is an $\aleph label$, then the `InsertAfterNlabel` Algorithm in Algorithm 4.6 is invoked which will always reuse the smallest label between N_{left}

and N_{right} or generate a new label that is the smallest label between N_{left} and N_{right}.

Scenario 1 will always select the first $\ell label$ or $\Re label$ lexicographically greater than N_{left}, that is the first $\ell label$ or $\Re label$ to the immediate right of N_{left}. Consequently, Algorithm 4.5 is both *deterministic* and *consistent* with the `SimpleInsertion` Algorithm and `AssignInitialLabels` Algorithm when generating labels of type $\ell label$ or $\Re label$. Scenarios 2 to 6 (inclusive) will always reuse or generate a new $\aleph label$ and this label will always be the smallest possible valid label between N_{left} and N_{right}. Consequently, Algorithm 4.5 is both *deterministic* and *consistent* with the `SimpleInsertion` Algorithm in the generation of labels of type $\aleph label$. Therefore, the EBSL labeling scheme always guarantees that every deleted node label can be reused.

4.6 Analysis of the EBSL Labeling Scheme

In this section, we present an analysis of the EBSL labeling scheme. In the first part, we analyze the EBSL label sizes and in the second part, we identify the benefits and limitations of the EBSL dynamic node labeling scheme.

4.6.1 EBSL Label Size Analysis

We now briefly analyze the growth rate of the EBSL label sizes when a document is labeled initially and the growth rate of labels under frequent node insertions.

Document Initialization Given an arbitrary node P that has M child nodes, what is the total label size of all M child nodes assigned by the `AssignInitialLabels` Algorithm? The first child node is always assigned the label 10. Each subsequent child node is assigned a label that is one bit longer than the preceding node label. Hence, the growth rate of the label size is linear, that is one bit per node insertion. We now determine the total label size of all M child sibling nodes.

If M = 3, then labels are: 10, 110, 1110.

Algorithm 4.6: EBSL `InsertAfterNlabel`

```
/* This algorithm inserts a new node between two consecutive sibling nodes $N_{left}$ and
   $N_{right}$ where $N_{left}$ is an $\aleph label$.                                        */
input  : A self_label $N_{left}$, the left sibling node such that $N_{left}$ is an $\aleph label$.
         A self_label $N_{right}$, the right sibling node such that $N_{right}$ is one of $\{\Re label, \ell label, \aleph label\}$.
output: A self_label $N_{new}$ such that $N_{left} \prec_e N_{new} \prec_e N_{right}$.
begin
    if (length($N_{left}$) > length($N_{right}$)) then
        if ($N_{right}$ is a prefix of $N_{left}$) then
            if (length($N_{left}$) == (length($N_{right}$) + 1 )) then
                /* There is no deleted node label between $N_{left}$ and $N_{right}$.       */
                $N_{new} \longleftarrow$ SimpleInsertion($N_{left}$, $N_{right}$);
            else
                /* We know $N_{left}$ is at least 2 bits longer than $N_{right}$.           */
                $N_{temp} \longleftarrow$ substring($N_{left}$, length($N_{right}$) + 2, len($N_{left}$));
                countOnes $\longleftarrow$ number of consecutive 1 bits at beginning of $N_{temp}$;
                if (countOnes == length($N_{temp}$)) then
                    /* If no 0 bit in $N_{temp}$, then no deleted node label available.   */
                    $N_{new} \longleftarrow$ SimpleInsertion($N_{left}$, $N_{right}$);
                else
                    /* Reuse shortest deleted node label between $N_{left}$ and $N_{right}$.   */
                    $N_{new} \longleftarrow$ substring($N_{left}$, 1, len($N_{right}$) + 1 + countOnes);
                end
            end
        else if ($N_{right}$ is not a prefix of $N_{left}$) then
            P $\longleftarrow$ first position of difference between $N_{left}$ and $N_{right}$;
            /* Reuse shortest deleted node label between $N_{left}$ and $N_{right}$.         */
            $N_{new} \longleftarrow$ substring($N_{right}$, 1, P − 1);
        end

    else if (length($N_{left}$) == length($N_{right}$)) then
        P $\longleftarrow$ first position of difference between $N_{left}$ and $N_{right}$;
        $N_{new} \longleftarrow$ substring($N_{left}$, 1, P − 1);

    else if (length($N_{left}$) < length($N_{right}$)) then
        if ($N_{left}$ is a prefix of $N_{right}$) then
            if (length($N_{right}$) == (length($N_{left}$) + 1)) then
                $N_{new} \longleftarrow$ SimpleInsertion($N_{left}$, $N_{right}$);
            else
                $N_{temp} \longleftarrow$ substring($N_{right}$, length($N_{left}$) + 2, len($N_{right}$));
                countZeros $\longleftarrow$ number of consecutive 0 bits at beginning of $N_{temp}$;
                if (countZeros == len($N_{temp}$)) then
                    $N_{new} \longleftarrow$ SimpleInsertion($N_{left}$, $N_{right}$);
                else
                    $N_{new} \longleftarrow$ substring($N_{right}$, 1, len($N_{left}$) + 1 + countZeros);
                end
            end
        else if ($N_{left}$ is not a prefix of $N_{right}$) then
            P $\longleftarrow$ first position of difference between $N_{left}$ and $N_{right}$;
            $N_{new} \longleftarrow$ substring($N_{right}$, 1, P − 1);
        end
    end
    return $N_{new}$;
end
```

Total Label Size =	2 + 3 + 4 = 9 bits
If M = 4, then labels are:	10, 110, 1110, 11110.
Total Label Size =	2 + 3 + 4 + 5 = 14 bits

if M = N, then labels are 10, 110, 1110, 11110, ...

Total Label Size = 2 + 3 + 4 + ... + N + (N+1)

We know from number theory that the sum of 1 to N = $\frac{N*(N+1)}{2}$

Therefore Total Label Size = $\underbrace{\frac{N*(N+1)}{2}}_{\textit{sum from 1 to N}} \quad \underbrace{+ (N+1)}_{\textit{add (N+1)}} \quad \underbrace{-1}_{\textit{remove 1}}$

$$= \frac{N*(N+1)}{2} + \frac{N}{1}$$

$$= \frac{(N*(N+1))+2N}{2}$$

$$= \frac{N^2+N+2N}{2}$$

$$= \frac{N^2+3N}{2} \text{ bits.} \qquad \square$$

Thus, given M child nodes of a parent node P, the space complexity of the total label size of M child nodes is quadratic: O(N^2).

Frequent Node Insertions. Under frequent node insertions, the label size growth rate is also linear and has the total label size as described under document initialization. Thus, for N frequently skewed node insertions, the space complexity of the total label size of N labels is quadratic: O(N^2).

4.6.2 Critique of EBSL Labeling Scheme

The EBSL dynamic labeling scheme offers the first complete solution to the open research question of how to fully support the reuse of deleted node labels in a dynamic XML environment. EBSL does not require the relabeling of existing nodes nor the recalculation of any values when inserting new nodes in an XML tree. EBSL also guarantees that every deleted node label can be reused. There is no node insertion scenario that will result in a deleted node label remaining unused when it would be appropriate to reuse that label at the position of insertion.

However, while the EBSL labeling scheme solves the problem of functionality with regard to the deleted node label reuse problem, it does so at a cost: the label size growth rate both at document initialization and under frequent node insertions is linear - one bit per node insertion. This growth rate leads rapidly to large label sizes when processing moderately large XML documents and under moderately-intensive node insertion scenarios. Recall that large label sizes lead to higher computational processing costs when performing label comparison operations - a core operation at the heart of every XML query and update service.

If the sizes of the labels initially assigned to an XML tree are large, then the benefits of being able to reuse those labels after arbitrary node deletions is minimal. If the sizes of the labels initially assigned to an XML tree are initially compact and small, then the benefits of reusing smaller labels are comparatively greater than the benefits of reusing larger labels. Consequently, the ability to reuse deleted node labels is a beneficial and positive feature to constrain the growth rate of label sizes under various node insertion scenarios.

Thus, while we have resolved the problem of label reuse, we have progressed toward the goal of label compactness. In the next Chapter, we will present the primary contribution of our dissertation - a compact and scalable adaptive encoding method for dynamic node labeling schemes. The overriding design goal motivating the specification of the compact adaptive encoding method is to enable a dynamic labeling scheme to guarantee a highly constrained label size growth rate both at document initialization and under any arbitrary node insertion scenario.

Chapter 5

Compact Adaptive Growth Method

In the previous Chapter, we proposed a labeling scheme that fully supports the reuse of deleted node labels under arbitrary node insertions. In this Chapter, we focus on an ever greater problem that affects all dynamic labeling schemes proposed to date: the ability to assign compact labels both at document initialization and under arbitrary node insertions, whether they are single insertions or frequent node insertions.

This Chapter is structured as follows: In §5.1, we introduce the SCOOTER dynamic labeling scheme and outline its unique characteristics. The dynamic labeling scheme serves as a useful deployment in which we can present our Compact Adaptive Growth Method. In §5.2, we present two `AssignInitialLabels` Algorithms, one permitting a sequential assignment of labels, the other permitting a random access assignment of labels. In §5.3, we present our Compact Adaptive Growth Method - a method that facilitates the generation of highly compact labels in a dynamic environment. Finally, in §5.4, we provide a comprehensive theoretical evaluation of the Compact Adaptive Growth Method.

5.1 The SCOOTER Dynamic Labeling Scheme

In this section, we present the SCOOTER dynamic node labeling scheme [48] and highlight its unique characteristics.

The name SCOOTER, encapsulates the core properties - Scalable, Compact, Ordered, Orthogonal, Trinary Encoded Reusable dynamic labeling scheme. The labeling scheme is scalable insofar as it can support an arbitrary number of node insertions and deletions while completely avoiding the need to relabel nodes. SCOOTER provides compact label sizes by constraining the label size growth rate both at document initialization and under various node insertions scenarios. Order is maintained between nodes at all times by way of lexicographical comparison. The labeling scheme is orthogonal to the encoding technique employed to determine structural relationships between node labels. Specifically, the labeling scheme can be deployed as a prefix labeling scheme or as a containment labeling scheme. The labeling scheme adopts the quaternary encoding presented in the QED labeling scheme [35]. A quaternary code consists of four numbers "0", "1", "2", "3", and each number is stored with two bits, i.e.: "00", "01", "10", "11". Like the QED labeling scheme, the SCOOTER labeling scheme employs the separator label storage scheme. Consequently, a SCOOTER code is a quaternary code such that the number "0" is reserved as a separator and only the numbers "1", "2", "3" are used in the code itself. Thus, the codes are encoded in the ternary base. Lastly, the labeling scheme supports the reuse of shorter deleted node labels when available.

Throughout this dissertation, we present SCOOTER and the accompanying algorithms to encode structural relationships between nodes as a prefix labeling scheme because several studies [22] [23] have shown the prefix-based approach is most suitable for a dynamic XML environment. Specifically, in [22] they confirm that a prefix labeling scheme is of *paramount importance for the lock protocol* and *for the entire performance of concurrency control in XML trees.* Recall that in a prefix labeling scheme, the label of a node in an XML tree consists of the parent's label concatenated with a delimiter (separator in the case of SCOOTER) and a positional identifier of the node itself. The positional identifier indicates the position of the node relative to its siblings. The SCOOTER code represents the positional identifier of a node, also known as the self_label.

5.1.1 SCOOTER Lexicographical Order

SCOOTER compares node labels using lexicographical order and not numerical order. The definition of lexicographical order used by SCOOTER is in line with existing approaches [34] [22] [43] [4] [68] [31] [38] [32]. Let us define lexicographical order now as all other research deals with binary strings or alphabetical strings and not quaternary strings.

Definition 5.1. *(Lexicographical order) Given two SCOOTER codes S_{left} and S_{right} (S_{left} represents the left code, S_{right} represents the right code), S_{left} is said to be lexicographical equal to S_{right} iff they are identical. S_{left} is said to be lexicographically less than S_{right} ($S_{left} \prec S_{right}$) iff:*

1. *the comparison of S_{left} and S_{right} is digit by digit from left to right. If the current digit of S_{left} is less than the current digit of S_{right} then $S_{left} \prec S_{right}$ and stop the comparison, or*

2. *S_{left} is a prefix of S_{right}.*

Example 5.1. *Given two SCOOTER codes 112 and 113, $112 \prec 113$ because the comparison is from left to right and the 3^{rd} digit of 112 is 2 and the 3^{rd} digit of 113 is 3. Given two codes 232 and 23212, $232 \prec 23212$ because 232 is a prefix of 23212. Given two codes 32 and 23, $32 \succ 23$ because the comparison is from left to right and the 1^{st} digit of 32 is 3 and the 1^{st} digit of 23 is 2.*

A SCOOTER code must end in a "2" or a "3" in order to maintain lexicographical order in the presence of dynamic insertions. An example illustrating why this is necessary is presented in Example 5.2.

Example 5.2. *Let us assume a SCOOTER code can terminate with a "1" digit. Given two SCOOTER codes 1 and 11, $1 \prec 11$ because 1 is a prefix of 11 (from Definition 5.1). There does not exist a SCOOTER code S_{new} such that $1 \prec S_{new} \prec$*

11. (Recall that "0" is reserved as a separator and only the digits "1", "2" and "3" may appear in a SCOOTER code).

5.2 Assigning Initial Labels

In this section, we present two different algorithms for assigning labels at document initialization - a sequential assignment algorithm and a random access assignment algorithm. Both algorithms generate the same labels. However, each algorithm offers distinct benefits with respect to each other which will be highlighted below. Both of these algorithms implement an optimization to ensure the smallest possible labels are assigned, so before we present the algorithms, it is useful to first present the rules governing the assignment of SCOOTER labels.

5.2.1 Rules for Assigning Labels

There are a small number of rules used to determine the assignment of labels in order to ensure a compact label size and to maintain lexicographical order between labels. The first two SCOOTER (assignment) rules concern the first label while the last two rules concern the remaining labels.

SR 5.1. *The first label consists of a sequence of "1" digits of length k and terminates with a "2" digit, where the length of the label is k + 1.*

SR 5.2. *The first label will always be the maximum allowable length.*

SR 5.3. *No label can terminate with a "1" digit.*

SR 5.4. *The second and remaining label can be of any allowable length.*

In Table 5.1, columns 2, 3 and 4 illustrate the labels assigned by the SCOOTER `AssignInitialLabels` Algorithms when the maximum label length is 1, 2 and 3 digits respectively. Columns 5 and 6 illustrate the labels generated by the `AssignInitialLabels` Algorithms of the SCOOTER and QED labeling schemes respectively when initially labeling 20 child nodes (they are presented to facilitate a comparison).

Column 1	Column 2	Column 3	Column 4	Column 5	Column 6
Decimal	SCOOTER all labels with max length of 1 digit	SCOOTER all labels with max length of 2 digits	SCOOTER all labels with max length of 3 digits	SCOOTER assign 20 labels	QED assign 20 labels
1	2	12	112	12	112
2	3	13	113	13	12
3		2	12	2	122
4		22	122	212	123
5		23	123	213	13
6		3	13	22	132
7		32	132	222	2
8		33	133	223	212
9			2	23	22
10			212	232	222
11			213	233	223
12			22	3	23
13			222	312	232
14			223	313	3
15			23	32	312
16			232	322	32
17			233	323	322
18			3	33	323
19			312	332	33
20			313	333	332
21			32		
22			322		
23			323		
24			33		
25			323		
26			333		
Total Size				100 bits	100 bits

Table 5.1: SCOOTER and QED Sample Labels

As previously stated, the first label must terminate with a "2" digit but as it must be of maximum allowable length, it is preceded by a sequence of "1" digits. The sequence of "1" digits may be zero (empty) if the maximum label length is one digit.

5.2.2 The `AssignInitialLabels_Sequential` Algorithm

Algorithm 5.1 is the SCOOTER `AssignInitialLabels_Sequential` Algorithm and assumes a prefix labeling scheme. The algorithm takes as input, the number of child nodes of a parent to be labeled, and assigns a unique label to every child node. The root node of the tree always has the label 2. The algorithm may be applied recursively to label every node in an XML tree. When the number of child nodes *nodeCount* is expressed as a positive integer in the base three, line 3 computes the minimum number of digits required to represent *nodeCount* in the base 3. The minimum number of digits required will determine the maximum label size (*maxLabelSize*) of all labels assigned by the `AssignInitialLabels_Sequential` Algorithm. The total number of labels that may be assigned with a maximum length of size *maxLabelSize* is computed using the formula: maxLabels = $3^{maxLabelSize} - 1$ (line 4).

Algorithm 5.1: SCOOTER `AssignInitialLabels_Sequential`

```
    /* Assign unique labels to all child nodes of a parent node.              */
    input  : nodeCount - the number of child nodes to be labeled.
    output: A unique SCOOTER code (self_label) for each child node.
 1  begin
 2      labelList ⟵ an empty array;
 3      maxLabelSize ⟵ Ceiling ($\log_3$(nodeCount + 1));
 4      maxLabels ⟵ $3^{maxLabelSize} - 1$;
 5      difference ⟵ maxLabels − nodeCount;
 6      numShorterLabels ⟵ Floor (difference / 2);
 7      if (numShorterLabels > 0) then
 8          labelSize ⟵ maxLabelSize − 1;
 9          numRemainingLabels ⟵ nodeCount − numShorterLabels;
10      else
11          labelSize ⟵ maxLabelSize;
12          numRemainingLabels ⟵ nodeCount − 1;
13      end
        /* Compute the SCOOTER self_label of the first child.                  */
14      self_label ⟵ ∅;
15      for (i=1; i < labelSize; i++) do
16          self_label ⟵ self_label ⊕ 1 ;            // ⊕ denotes concatenation
17      end
18      self_label ⟵ self_label ⊕ 2;
19      labelList.add (self_label);
        /* Now compute the SCOOTER self_labels for all subsequent children.    */
20      for (i=1; i < numShorterLabels; i++) do
21          self_label ⟵ Increment (self_label, labelSize);
22          labelList.add (self_label);
23      end
24      for (i=1; i ≤ numRemainingLabels; i++) do
25          self_label ⟵ Increment (self_label, maxLabelSize);
26          labelList.add (self_label);
27      end
28      return labelList;
29  end
```

After generating the first label, a naive approach to assigning every subsequent label is to lexicographically increment the preceding label. Although two thirds of labels available will be of length *maxLabelSize*, one third of the labels available will have shorter lengths varying from one digit to (maxLabelSize − 1). Specifically, given a maximum label size of length *maxLabelSize*, there are $3^1 - 1$ labels of length 1, 3^2 −

3^1 labels of length 2, $3^3 - 3^2$ labels of length 3,..., $3^{maxLabelSize} - 3^{maxLabelSize-1}$ labels of length *maxLabelSize*.

Given a maximum allowable SCOOTER label length of D digits, a shorter label is any SCOOTER label with a length less than or equal to D − 1. Our design goal is the assignment of compact labels, consequently we designed our `AssignInitialLabels_Sequential` Algorithm to guarantee that all shorter labels are assigned. In order to describe how this is possible, we must first present three properties that are exploited by the `AssignInitialLabels_Sequential` Algorithm to achieve this goal.

Theorem 5.1. *The first label assigned by the* ***`AssignInitialLabels_Sequential`*** *Algorithm will always be lexicographically less than any other label assigned by the* ***`AssignInitialLabels_Sequential`*** *Algorithm.*

Proof: The lexicographically least string encoding using the three digits "1", "2" and "3" is the string consisting of a single "1" digit. However, a SCOOTER label can never terminate with a "1" digit (from SR5.3). Hence, the lexicographically least string encoding is the string of length D digits that terminates with a "2" digit and is preceded with a sequence of "1" digits of length (D − 1). The `AssignInitialLabels_Sequential` Algorithm always generates the first label such that it is a string of length D digits that terminates with a "2" digit and is preceded with a sequence of "1" digits of length (D − 1). Hence, the first label assigned will always be lexicographically less than any other label assigned by the `AssignInitialLabels_Sequential` Algorithm. □

Theorem 5.2. *The* n^{th} *label assigned by the SCOOTER* ***`AssignInitialLabels_Sequential`*** *Algorithm will always be lexicographically greater than the* $(n-1)^{th}$ *label assigned by the* ***`AssignInitialLabels_Sequential`*** *Algorithm.*

Proof: From Theorem 5.1, we know that the first label assigned by the `AssignInitialLabels_Sequential` Algorithm is always the lexicographical smallest label. Thereafter, the second and subsequent label assigned by the `AssignInitialLabels_Sequential` Algorithm is generated by lexicographically incrementing the label immediately preceding it. Thus, the nth label assigned will always be lexicographically greater than the (n−1)th label. □

We now prove that of the total number of labels that may be assigned with a maximum length of size *maxLabelSize* (that is $3^{maxLabelSize} - 1$), every third label is a *shorter label.*

Theorem 5.3. *Given an arbitrary nth label from the total number of labels that may be assigned with a maximum length of D digits, if n is divisible by three, then the length of the nth label is less than D digits. If n is not divisible by three, then the length of the nth label is always D digits.*

Proof: From SR5.2, we know the first label will always be of length D digits. From SR5.1, we know that the first label always terminates with a "2" digit. From Theorem 5.2, we know the next label is always the lexicographical increment of the preceding label. Therefore, we know the second label must be of length D digits and terminate with a "3" digit (because "3" is the immediate lexicographical increment of "2"). In order to obtain the third label, the second label cannot be incremented by extending its length because it is already at the maximum allowable length of D digits. Therefore, the second label is lexicographically incremented by incrementing the rightmost non-three digit. For example, if maximum label length D = 3, then the label 113 lexicographically increments to 12. When performing a lexicographical increment operation, all digits occurring after the position of the modified digit are discarded (thus, 113 increments to 12; 113 does not increment to 123). Therefore, the third label always has a length less than D digits. Given any label *k* with a length equal to D digits and terminating with a "2" digit, the lexicographical increment operation generates the label *k+1* such that it has a length of D digits and terminates with a "3" digit. Then, as with lexicographically incrementing the second label assigned by the `AssignInitialLabels_Sequential` Algorithm, the label *k+2* is obtained by lexicographically incrementing the rightmost non-three digit in label *k+1* (for example, label 31333 is lexicographically incremented to 32). The label *k+2* will always have a length less than D digits precisely because the incremented digit occurs before position D in the label and all digits after position D are discarded. The lexicographical increment operation always generates the label *k+3* such that it has a length of D digits and terminates with a "2" digit because the increment operation on a label with less than D digits always generates a label with D digits and terminating with a "2" digit . Thus, the labels *k* and *k+3* have a length of D digits and terminate with a "2" digit. Therefore, given the nth label, if n is divisible by three, then the length of the nth label is less than D digits. If n is not divisible by three, then the length of the nth label is always D digits.

□

Assigning the Shorter labels

The number of nodes to be labeled (*nodeCount*) by the `AssignInitialLabels_Sequential` Algorithm may be less than the total number of labels available with a maximum length of *maxLabelSize*. By exploiting the lexicographical properties of Theorem 5.2 and given that every third label will have a length less than *maxLabelSize* from Theorem 5.3, we can guarantee to assign all shorter labels when generating *nodeCount* labels. The algorithm computes the difference between the total number of labels available with a maximum length of *maxLabelSize* and the number of nodes to be labeled (line 5) and then determines the number of shorter labels to initially assign (line 6). The algorithm initially assigns the shorter labels (whose ordinal values are divisible by 3 and have a length less than *maxLabelSize*) (lines 7–22) before necessarily assigning all remaining labels with a length less than or equal to *maxLabelSize* (lines 24–27). We know for certain that labels with length *maxLabelSize* will be assigned because *maxLabelSize* was computed to be the smallest possible length to represent *nodeCount* labels (line 3).

5.2.3 The `Increment` Algorithm

Algorithm 5.2 is the SCOOTER `Increment` Algorithm called by the `AssignInitialLabels_Sequential` Algorithm. The algorithm takes two input parameters: a node self_label, and maxLabelSize - the maximum number of digits allowed in the self_label. The output returned by the algorithm is a new self_label that is the lexicographical increment of the input self_label. The algorithm will never receive a

Algorithm 5.2: SCOOTER `Increment`

```
input  : $N_{left}$ - a node label;
         $maxLabelSize$ - maximum number of digits allowed in label.
output: $N_{new}$ - a new self_label such that $N_{left} \prec N_{new}$.
begin
    $N_{temp} \longleftarrow N_{left}$;
    if (length($N_{temp}$) == maxLabelSize) then
        if (last digit in $N_{temp}$ is '1') then
            $N_{new} \longleftarrow N_{temp}$ with last digit changed to '2';
        else if (last digit in $N_{temp}$ is '2') then
            $N_{new} \longleftarrow N_{temp}$ with last digit changed to '3';
        else if (last digit in $N_{temp}$ is '3') then
            while (last digit of $N_{temp}$ is '3') do
                $N_{temp} \longleftarrow N_{temp}$ with last digit removed;
            end
            if (last digit in $N_{temp}$ is '1') then
                $N_{new} \longleftarrow N_{temp}$ with last digit changed to '2';
            else if (last symbol in $N_{temp}$ is '2') then
                $N_{new} \longleftarrow N_{temp}$ with last digit changed to '3';
            end
        end
    else if (length($N_{temp}$) < maxLabelSize) then
        while (i = Length($N_{temp}$) + 1; i < maxLabelSize; i++) do
            $N_{temp} \longleftarrow N_{temp} \oplus 1$;
        end
        $N_{new} \longleftarrow N_{temp} \oplus 2$;
    end
    return $N_{new}$;
end
```

self_label longer than maxLabelSize. Furthermore, the algorithm will never receive a self_label with length maxLabelSize **and** consisting of all "3" digits by virtue of line 3 in Algorithm 5.1. Lastly, although the `Increment` Algorithm will never receive a self_label from the `AssignInitialLabels_Sequential` Algorithm that terminates with a "1" digit, we will pass substrings of labels that may terminate with a "1" digit to the `Increment` Algorithm in order to process dynamic node label insertions and deletions (discussed in Chapter 6).

5.2.4 The `AssignInitialLabels_NodeK` Algorithm

The `AssignInitialLabels_NodeK` Algorithm is a label assignment algorithm that facilitates random access. Given a positive integer k and the total number of nodes to be labeled n, this algorithm determines the label of the k^{th} arbitrary node without the need to compute any other label. The first part of this algorithm is similar to the `AssignInitialLabels_Sequential` Algorithm: maxLabelSize is computed (line 2); the total number of labels available for assignment computed (line 4); and then the number of shorter labels that may be initially assigned is determined (lines 5–6). The algorithm then determines if the k^{th} label is one of the shorter labels (lines 7–14).

Algorithm 5.3: SCOOTER `AssignInitialLabel_NodeK`

```
   /* Assign initial label to the K^{th} child node.                              */
   input  : N - a positive integer representing the total number of nodes to be labeled.
            K - a positive integer representing the node whose label we want, such that $1 \leq K \leq N$.
   output: A unique SCOOTER code (self_label) for the K^{th} child node.
1  begin
2      D ⟵ Ceiling ($\log_3$(N + 1));
3      divisor ⟵ $3^D$;
4      maxLabels ⟵ $3^D - 1$;
5      difference ⟵ maxLabels − N;
6      numShorterLabels ⟵ Floor (difference / 2);
7      if (K ≤ numShorterLabels) then
8          divisor ⟵ divisor / 3;
9          D ⟵ D − 1;
10     else
11         numRemainingLabels ⟵ K − numShorterLabels;
12         startIndexOfRemainingLabels = numShorterLabels * 3;
13         K ⟵ startIndexOfRemainingLabels + numRemainingLabels;
14     end
15     quotient ⟵ K;
16     self_label ⟵ ∅;
17     for (j=1; j ≤ D; j++) do
18         divisor ⟵ divisor / 3;
19         code ⟵ Floor (quotient / divisor) + 1;
20         self_label ⟵ self_label ⊕ code;
21         remainder ⟵ quotient mod divisor;
22         if (remainder == 0) then
23             return self_label;
24         else
25             quotient ⟵ remainder;
26         end
27     end
28 end
```

The final part of the algorithm (lines 15–27) performs a *decrease and conquer* search to identify the label of the k^{th} node. A decrease and conquer algorithm is similar to a divide and conquer algorithm. A divide and conquer algorithm splits the problem into multiple smaller problems. A decrease and conquer algorithm reduces the problem at each step to a single smaller instance of the same problem. At the first iteration, the FOR loop identifies the most significant digit (leftmost digit) in the label. At each subsequent iteration the search space is reduced by a constant factor of 3 (because our labels are encoded in the base 3) and the next digit in the label is identified. The FOR loop terminates when all digits in the label are identified (which requires at most *maxLabelSize* iterations).

5.2.5 Summary Analysis of `AssignInitialLabels` Algorithms

We now highlight four distinct characteristics of our `AssignInitialLabels` Algorithms.

1. Each Scooter label can be determined entirely using the label of the node to the immediate left (and immediate right, please refer to appendix §B.1 for the `Decrement` Algorithm). Consequently, the `AssignInitialLabels_Sequential` Algorithm is a *deterministic* algorithm as defined in Definition 4.1. This is a key property which we will exploit to enable and maintain compact labels in the presence of an arbitrary number of node insertions and deletions. This property also facilitates the reuse of deleted node labels.

2. One third of all labels available for assignment will have a length less than *maxLabelSize* (from Theorem 5.3). The `AssignInitialLabels_Sequential` Algorithm will use all of these shorter labels when initially assigning labels. Consequently, the `AssignInitialLabels_Sequential` Algorithm always assigns the most compact SCOOTER labels.

3. One significant limitation of existing approaches that require a sequential process to assign node labels is that to generate a label for node n, we must first generate all $n-1$ node labels. This limitation can be a significant bottleneck when processing large XML files. Hence, we presented the `AssignInitialLabels_NodeK` Algorithm that can determine an arbitrary k^{th} child node label without having to compute any other child node label. When parsing very large XML documents, both the `AssignInitialLabels_Sequential` Algorithm and the `AssignInitialLabels_NodeK` Algorithm can be employed together. For example, in a distributed environment supporting parallel processing, when initially labeling 100 million child nodes, the `AssignInitialLabels_NodeK` Algorithm running on a server could identify the 100 node labels at each millionth position and each of these 100 labels could be sent to 100 clients computers. Each client computer can run in parallel the *Increment*

Algorithm to determine the 1 million labels lexicographically ordered immediately after the input label. Each client can then send the list of 1 million labels back to the server. The ability to compute node labels both sequentially and independently of one another, opens up the possibility of distributed and parallel processing in a multi-threaded and multi-core environment and may offer significant performance benefits.

4. The SCOOTER labels are encoded using the numeric base 3. In mathematical numeral systems, the base or radix is the number of unique symbols that a positional numeral system uses to represent numbers. For example, the decimal system uses the *base 10*, because it uses the 10 symbols from 0 through 9. The highest symbol usually has the value of one less than the base of that numeric system. In [26], the authors demonstrate the most economical radix for a numbering system is e, the base of the natural logarithms, with a value of approximately 2.718. Economy is measured as the product of the radix and the number of digits needed to express a range of given values. Consequently the economy is also a measure of how compact is the numerical representation of a given radix. In [26], the authors also demonstrate that the integer 3, being the closest integer to e, is almost always the most economical integer radix or base. For this reason, we have chosen to use the numeric *base 3* and consequently, quaternary codes to represent SCOOTER labels.

5.3 Compact Adaptive Growth Method

In this section, we present a new Compact Adaptive Growth Method which provides for the generation of labels with a highly constrained label size growth rate. The Compact Adaptive Growth Method provides the foundation and mechanism by which the SCOOTER dynamic labeling scheme can maintain a highly constrained label size growth rate irrespective of the quantity of arbitrary and repeated node label insertions and deletions. The SCOOTER node label insertion algorithms exploiting the Compact Adaptive Growth Method will be presented in the next Chapter. We now introduce the Compact Adaptive Growth Method and begin with a simple example that provides an overview of the conceptual approach followed by a more detailed analysis of its underlying properties.

Consider an XML tree consisting of a root node R and two child nodes with selflabels '2' and '3'. A sequence of 100 nodes is inserted to the right of the rightmost child node. Table 5.2 illustrates the first 21 insertions. The first $(3^1 - 1)$ node labels generated consist of a *prefix* string '3' and a *postfix* string generated by the `AssignInitialLabels` Algorithm for a maxLabelSize of 1 digit (i.e.: '2' and '3'). However, we always select the first postfix string to be the label at the midpoint of all available labels (i.e.: '3') so as to ensure future node insertions before and after this node have the same growth rate in label size (please refer to Table 5.1 for examples of labels generated by the `AssignInitialLabels` Algorithm). For the next 4 $((3^2 - 1) / 2)$ insertions, from the second to the fifth insertion inclusive, the newly generated labels consist of a *prefix* string '33' and a *postfix* string that mirrors the labels generated for a maxLabelSize of 2 digits starting at the midposition (e.g.: '23', '3', '32' '33'). The midposition is calculated using the same formula used to determine the number of nodes available for insertion $((3^2 - 1) / 2)$. For the next 13 $((3^3 - 1) / 2)$ insertions, from the 6th to the 18th insertion inclusive, labels consist of a *prefix* string '3333' and a *postfix* string that mirrors the labels generated for a maxLabelSize of 3 digits starting from the midposition (e.g.: '223', '23', '232', '233' and so on). This process is repeated as many times as required.

We now provide an analysis of the underlying properties. Conceptually, we consider an inserted label as comprising of two components: a *prefix* and a *postfix*. We define eight rules that govern the operation of the Compact Adaptive Growth Method. The first two compact adaptive rules (CAR) determine the initial length of the prefix and postfix components. CAR5.3 and CAR5.4 specify the allowable value of the prefix and postfix. CAR5.5 specifies the maximum allowable label length. CAR5.6 and CAR5.7 govern the compact adaptive growth rate of the prefix and postfix. Finally, CAR5.8 determines the value of the first postfix immediately after an adaptive increase in the prefix and maximum allowable postfix lengths.

CAR 5.1. *The smallest allowable prefix length is 1 digit.*

Insert after right-most node	SCOOTER label
	2
	3
1	33
2	3323
3	333
4	3332
5	3333
6	3333223
7	333323
8	3333232
9	3333233
10	33333
11	3333312
12	3333313
13	333332
14	3333322
15	3333323
16	333333
17	3333332
18	3333333
19	33333332223
20	3333333223
21	33333332232

Table 5.2

Col. 1	Col. 2	Col. 3	Col. 4	Col. 5	Col. 6	Col. 7
Growth counter	Node Start	Node End	Prefix length	Max Post-fix length	Max selflabel length	Selflabel total bits
1	1	1	1	1	2	4
2	2	5	2	2	4	8
3	6	18	4	3	7	14
4	19	58	7	4	11	22
5	59	179	11	5	16	32
6	180	543	16	6	22	44
7	544	1,636	22	7	29	58
8	1,637	4,916	29	8	37	74
9	4,917	14,757	37	9	46	92
10	14,758	44,281	46	10	56	112
11	44,282	132,854	56	11	67	134
12	132,855	398,574	67	12	79	158
13	398,575	1,195,735	79	13	92	184
14	1,195,736	3,587,219	92	14	106	212
15	3,587,220	10,761,672	106	15	121	242
16	10,761,673	32,285,032	121	16	137	274
17	32,285,033	96,855,113	137	17	154	308
18	96,855,114	290,565,357	154	18	172	344
19	290,565,358	871,696,090	172	19	191	382
20	871,696,091	2,615,088,290	191	20	211	422
21	2,615,088,291	7,845,264,891	211	21	232	464
⋮	⋮	⋮	⋮	⋮	⋮	⋮
n	$\frac{3^n-2n+3}{4}$	$\frac{3^{n+1}-2n-3}{4}$	$\frac{n^2-n}{2}+1$	n	$\frac{n^2+n}{2}+1$	n^2+n+2

Table 5.3: Compact Adaptive Growth Rate

CAR 5.2. *When the prefix length is 1 digit, the maximum allowable postfix length is 1 digit.*

CAR 5.3. *The prefix string consists of one of two possible sequences: a sequence of one or more consecutive "3" digits; or a sequence of one or more consecutive "1" digits. The particular sequence chosen as prefix, depends on the insertion operation to be performed.*

CAR 5.4. *The length of the postfix string can be less than or equal to the maximum allowable postfix length.*

CAR 5.5. *The maximum label length is always equal to the sum of the prefix and the maximum allowable postfix length.*

CAR 5.6. *When generating a new rightmost label, we extend the length of the prefix if and only if the current rightmost label consists of all '3' symbols and the length* *of the current rightmost node label equals the sum of the current prefix length and maximum allowable postfix length.*

CAR 5.7. *The new prefix length is assigned the value of the previous maximum allowable label length; the new maximum postfix length is assigned the value of the previous maximum postfix length plus 1. This rule is codified in Algorithm 5.4.*

CAR 5.8. *When Algorithm 5.4 is invoked and an adaptive increase in the prefix and postfix lengths has been performed, the first postfix will always be assigned the label at the lexicographical midposition of all possible labels of length maximumAllowablePostfixLength.*

CAR5.6 specifies the conditions under which a growth in the label size is necessary. CAR5.7 specifies the rate at which the label size must grow. CAR5.6 defines the conditions under which CAR5.7 is invoked. For example, when inserting a new node after a current rightmost node label '33', the current rightmost node label has a prefix length of 1 and a maximum postfix length of 1, hence CAR5.6 is satisfied and CAR5.7 is invoked. The prefix length is set to the value of the previous maximum label length (2 digits) and the new maximum postfix length is assigned the value of the previous maximum postfix length plus 1 (2 digits). The prefix is set to a string of "3" digits of length 2, and the postfix is set to the label at the midposition of all possible labels of length 2, namely "23". Hence, the new rightmost node label is the concatenation of the prefix and postfix, namely '3323'.

Algorithm 5.4: SCOOTER `ComputePrefixPostfixLengths`

```
/* Compute the prefix length and the maximum allowable postfix length of a self_label.
   */
input  : numDigits - the number of consecutive digits of type prefix at start of self_label.
output: prefixLength - the prefix length of the self_label.
        maxPostfixLength - the maximum allowable postfix length of the self_label.
begin
    prefixLength ⟵ 1;
    maxPostfixLength ⟵ 1;
    labelLength ⟵ prefixLength + maxPostfixLength;
    while (labelLength ≤ numDigits) do
        prefixLength ⟵ labelLength;
        maxPostfixLength ⟵ maxPostfixLength + 1;
        labelLength ⟵ prefixLength + maxPostfixLength;
    end
    return prefixLength, maxPostfixLength;
end
```

All bit-string dynamic labeling schemes (including QED and CDBS [38]) have a label growth rate of one bit per node insertion. Therefore, after one thousand insertions, one million insertions and one billion insertions, the largest selflabel sizes are 1,000 bits, 1,000,000 bits and 1,000,000,000 bits respectively. In contrast, after one thousand insertions, one million insertions and one billion insertions, the largest SCOOTER selflabels are 58 bits, 184 bits and 422 bits respectively (please refer to columns 2, 3 and 7 of Table 5.3). Thus, SCOOTER labels are several orders of magnitude smaller than the labels of all existing bit-string dynamic labeling schemes when processing frequently skewed insertions. Furthermore, in contrast to all existing dynamic labeling schemes, SCOOTER generates compact labels without requiring advance knowledge of the number of nodes to be inserted. The Compact Adaptive Growth Method is made possible by virtue of the *deterministic* property of our `AssignInitialLabels` Algorithms. The Compact Adaptive Growth Method may also be applied when inserting new nodes before the leftmost node. However, in this case we count the number of consecutive '1' symbols to determine the length of the prefix.

5.4 Compact Adaptive Growth Evaluation

In this section, we present a comprehensive theoretical evaluation of our Compact Adaptive Growth Method, hereafter referred to using the acronym CAGM. We perform our evaluation in the context of the generation of node labels for our SCOOTER dynamic labeling scheme. Our overall objective is to quantify the relationship between the growth rate of the number of labels to be inserted with the corresponding growth rate in label size. In order to quantify this relationship, we must address a number of goals; For any positive integer N:

1. How many unique SCOOTER labels are there with a length less than or equal to N? We address this question in Theorem 5.4.

2. How many new labels are available for insertion after the Nth adaptive increase in the prefix length? We address this question in Theorem 5.5.

3. What is the length of the prefix after N adaptive increases in the prefix length? We address this question in Theorem 5.6.
4. What is the maximum length (in digits) of a SCOOTER self_label after N adaptive increases in the prefix length? We address this question in Theorem 5.7.
5. What is the total number of labels available for insertion after N adaptive increases in the prefix length? We address this question in Theorem 5.8.
6. What is the ordinal number of the first label available for use after N adaptive increases in the prefix length? We address this question in Theorem 5.9.
7. Finally, what is the maximum size in bits of a SCOOTER self_label after N adaptive increases in the prefix length? We address this question in Theorem 5.10.

Theorem 5.4. *For any positive integer N, the total number of unique SCOOTER labels with a length less than or equal to N is:* $(3^N - 1)$.

Proof: Given a SCOOTER label may contain only the three digits {"1", "2", and "3"}, then only two unique labels may be generated using one digit, namely "2" and "3" (because a SCOOTER label cannot terminate with a "1" digit). When assigning SCOOTER labels with a length of two digits, the first digit can be one of {"1", "2", "3"}, and the second digit can be one of {"2", "3"}. Therefore, there are $(3^1 * 2) = 6$ possible labels of length 2 digits. When assigning SCOOTER labels with a length of three digits, the first two digits can be one of {"1", "2", "3"}, and the third digit can only be one of {"2", "3"}. Therefore, there are $(3^2 * 2) = 18$ possible labels of length 3 digits. When assigning labels of length k, the first $k-1$ digits can be one of {"1", "2", "3"}, and the k^{th} digit can be one of {"2", "3"}. Therefore, the number of unique SCOOTER labels of length k is $(3^{k-1} * 2)$. Hence, the total number of unique SCOOTER labels with a length less than or equal to N is:

$$\sum_{k=1}^{N} 3^{k-1} * 2 \qquad = (3^0 * 2) + (3^1 * 2) + (3^2 * 2) + (3^3 * 2) + \cdots + (3^{N-1} * 2)$$

$$= 2 * (3^0 + 3^1 + 3^2 + 3^3 + \cdots + 3^{N-1}) \qquad (5.1)$$

We know from number theory that the sum of x to the power of n from 0 to (N − 1) is:

$$\sum_{n=0}^{N-1} x^n = \frac{x^N - 1}{x - 1}$$

Therefore equation 5.1 becomes:

$$\begin{aligned}\sum_{k=1}^{N} 3^{k-1} * 2 &= 2 * \frac{3^N - 1}{3 - 1}\\ &= 2 * \frac{3^N - 1}{2}\\ &= 3^N - 1\end{aligned}$$

□

Given that we now know how many unique SCOOTER labels there are with a length ≤ N (from Theorem 5.4), we are in a position to determine how many new labels are available for insertion after the Nth adaptive increase in the prefix length.

Theorem 5.5. *The number of new labels available for insertion after the Nth adaptive increase in the prefix length is (3^N − 1) / 2 labels.*

Proof:
From Theorem 5.4, we know the total number of unique SCOOTER labels with a length less than or equal to N that can be assigned by the `AssignInitialLabels_Sequential` Algorithm is $3^N - 1$. However, after the Nth adaptive increase in the prefix length, the number of new labels available for insertion is equal to the number of labels that can be assigned by the `AssignInitialLabels_Sequential` Algorithm with a length of *maximumPostfixLength*. From CAR5.7, we know the *maximumPostfixLength* after N adaptive increases in the prefix length is simply N. Also, from CAR5.8 we know the first postfix after an adaptive increase in the prefix length is assigned the label at the midpoint between 1 and $3^N - 1$. Consequently, the number of new labels available for insertion after the Nth adaptive increase in the

prefix length is $(3^N - 1)$ / 2 labels. $\square$

In order to determine the maximum length (in digits) of a label after N adaptive increases in the prefix length, we must first determine the length of the prefix itself after N adaptive increases in the prefix length.

Theorem 5.6. *The length of the prefix after N adaptive increases in the prefix length is:* $\dfrac{N^2 - N}{2} + 1$ *(please refer to last row of column four in Table 5.3)*

Proof: From CAR5.1, we know that the smallest prefix length is 1 digit. From CAR5.2, it is given that when the prefix has length 1, the maximum allowable postfix length is defined to be 1 digit. For the second and subsequent adaptive increases in the prefix length, it is given from CAR5.7 and CAR5.5 that the prefix length after N adaptive increases is equal to the prefix length after (N−1) adaptive increases plus the postfix length after (N−1) adaptive increases. However, it is given by CAR5.7 that the postfix length grows by 1 after each adaptive increase. Therefore, the sum of the postfix length after (N−1) adaptive increases is the sum of 1 to (N−1). We know from number theory that the sum of 1 to N is $\dfrac{N(N+1)}{2}$. Therefore, the length of the prefix after N adaptive increases to the prefix length is equal to the first prefix of length 1 plus the sum of the postfix from 1 to (N−1):

$$\begin{aligned}
&= 1 + \frac{N(N+1)}{2} - N \\
&= 1 + \frac{N^2 + N}{2} - N \\
&= 1 + \frac{N^2 + N - 2N}{2} \\
&= \frac{N^2 - N}{2} + 1
\end{aligned}$$

$\square$

Theorem 5.7. *The maximum length (in digits) of a SCOOTER self_label after N adaptive increases in the prefix length is:* $\dfrac{N^2 + N}{2} + 1$ *(please refer to last row of column six in Table 5.3)*

Proof: From Theorem 5.6, we know the length of the prefix after N adaptive increases in the prefix length is: $\dfrac{N^2 - N}{2} + 1$. From CAR5.7, we know that the maximum length of the postfix after N adaptive increases in the prefix length is simply N.

Therefore, given that the maximum self_label length is always computed as the sum of the prefix length and maximum postfix length (CAR5.5), then the maximum length (in digits) of a SCOOTER self_label after N adaptive increases in the prefix length is:

$$\begin{aligned}&= \frac{N^2 - N}{2} + 1 + N \\ &= \frac{N^2 - N + 2N}{2} + 1 \\ &= \frac{N^2 + N}{2} + 1\end{aligned}$$

□

We determined how many new labels are available for insertion after the Nth adaptive increase in the prefix length (from Theorem 5.5). This information allows us to determine the total number of all labels available for insertion after N adaptive increases in the prefix length.

Theorem 5.8. *The total number of labels available for insertion after N adaptive increases in the prefix length is:* $\frac{3^{N+1} - 2N - 3}{4}$ *(please refer to last row of column three in Table 5.3)*

Proof: From Theorem 5.5, we know that after each nth adaptive increase in the prefix length, the number of new labels available at that prefix length is $\frac{3^n-1}{2}$. Therefore, after N adaptive increases in the prefix length, the total number of labels available is:

$$\begin{aligned}\sum_{n=1}^{N} \frac{3^n - 1}{2} \quad &= \frac{3^1 - 1}{2} + \frac{3^2 - 1}{2} + \frac{3^3 - 1}{2} + \cdots + \frac{3^N - 1}{2} \\ &= \frac{3^1 + 3^2 + 3^3 + \cdots + 3^N}{2} - \frac{N}{2} \qquad (5.2)\end{aligned}$$

We know from number theory that the sum of x to the power of n from 1 to N is:

$$\sum_{n=1}^{N} x^n \quad = \frac{x^{N+1} - x}{x - 1}$$

Therefore equation 5.2 becomes

$$\sum_{n=1}^{N} \frac{3^n - 1}{2} = \frac{\frac{3^{N+1}-3}{3-1}}{2} - \frac{N}{2}$$
$$= \frac{3^{N+1} - 3}{4} - \frac{N}{2}$$
$$= \frac{3^{N+1} - 3}{4} - \frac{2N}{4}$$
$$= \frac{3^{N+1} - 2N - 3}{4}$$

□

Theorem 5.9. *The ordinal number of the first label available for use after N adaptive increases in the prefix length is* $\frac{3^N - 2N + 3}{4}$ *(please refer to last row of column two in Table 5.3)*

Proof: The ordinal number of the first label available for use after N adaptive increases in the prefix length is always one more than the total number of labels available after N−1 adaptive increases in the prefix length. From Theorem 5.8, we know the total number of labels available after N adaptive increases in the prefix length is $\frac{3^{N+1} - 2N - 3}{4}$ Thus, by replacing every occurrence of N with N−1 and adding one to the final result, we can determine the ordinal number of the first label available for use after N adaptive increases as:

$$= \frac{3^{(N-1)+1} - 2(N-1) - 3}{4} + 1$$
$$= \frac{3^N - 2N + 2 - 3}{4} + \frac{4}{4}$$
$$= \frac{3^N - 2N + 3}{4}$$

□

Theorem 5.10. *The maximum size in bits of a SCOOTER self_label after N adaptive increases in the prefix length is:* $N^2 + N + 2$ *(please refer to last row of column seven in Table 5.3)*

Proof: From Theorem 5.7, we know the maximum length (in digits) of a self_label after N adaptive increases in the prefix length is: $\frac{N^2 + N}{2} + 1$. Given that SCOOTER labels employ a quaternary encoding where each digit is represented using 2 bits, the maximum size in bits of a self_label after N adaptive increases in the prefix length is equal to the maximum length (in digits) multiplied by 2:

$$= (\frac{N^2+N}{2} + 1) * 2$$
$$= (\frac{N^2+N+2}{2}) * 2$$
$$= N^2 + N + 2$$

□

5.4.1 Analysis

All existing bit-string dynamic labeling schemes have a minimum of one-bit-per-node-insertion label growth rate. Therefore, as the number of label insertions increases linearly, the corresponding growth in label size is at least linear. It follows that as the number of labels to be inserted increases exponentially, the corresponding growth in label size is also exponential for all bit-string dynamic labeling schemes. **However, as the number of SCOOTER labels to be inserted increases exponentially ($O(3^N)$ from Theorem 5.8), the corresponding growth in label size is quadratic ($O(N^2)$ from Theorem 5.10)**. Consequently, the Compact Adaptive Growth Method ensures SCOOTER node labels have a highly constrained growth rate under frequent node insertions. For example, after ten thousand node insertions and one hundred thousand node insertions, the largest self_labels generated by all bit-string dynamic labeling schemes have a minimum length of 10,000 bits and 100,000 bits respectively. In contrast, after ten thousand node insertions and one hundred thousand node insertions, the largest SCOOTER self_labels have a maximum length of 92 bits and 134 bits respectively.

Chapter 6

SCOOTER Node Label Insertion Algorithms

In the previous Chapter, we introduced a dynamic labeling scheme called SCOOTER. Specifically, we presented two `AssignInitialLabels` Algorithms that take as input an XML tree and output a labeled XML tree in which each node has been assigned a unique, ordered and compact label. We also presented our Compact Adaptive Growth Method (CAGM) which provides a method for generating labels with a highly constrained growth rate. In this Chapter, we present the SCOOTER node insertion algorithms - a suite of algorithms exploiting CAGM rules to facilitate node label insertions in an XML tree. In particular, the insertion algorithms guarantee a highly constrained growth rate in label size under any node insertion scenario. Furthermore, the algorithms support the reuse of deleted node labels.

When we consider a node insertion, there are three high level cases to be processed: insertion of a new node after the current rightmost node; before the current leftmost node; and between two consecutive sibling nodes. This Chapter is structured as follows: In §6.1, we present and explain in detail the algorithm to insert a new node after the current rightmost sibling node. In §6.2, we introduce and analyze the algorithm to insert a new node before the current leftmost sibling node. Finally, in §6.3, we present four algorithms to facilitate the insertion of a new node between two consecutive sibling nodes.

6.1 Insertion After the Current Rightmost Node.

Algorithm 6.1 is the SCOOTER `InsertNewNodeAfterRightmostNode` Algorithm that supports the reuse of deleted node labels when inserting a new node after the rightmost sibling node. Given one input node label N_{left}, Algorithm 6.1 generates a node label N_{new} such that N_{new} is the immediate lexicographical increment of the input label. For example, if the current rightmost node label is “3”, the sequence of the next 21 new rightmost node labels is illustrated in Table 5.2. The label generated and assigned to N_{new} may be a deleted node label that is being reused, or a new label. The algorithm will always reuse the smallest possible deleted node label if one is available, or if the algorithm detects a valid prefix (a consecutive sequence of “3” digits) at the beginning of N_{left}, then N_{new} will always be assigned a label generated according to CAGM rules, and consequently the label size will grow very slowly. We now walk through the algorithm and provide a detailed explanation.

- *(lines 2–5). Process N_{left} when no valid prefix is detected.*
  If the first digit of the label N_{left} is “1”, then N_{new} is assigned the label “2”. If the first digit of N_{left} is “2”, then N_{new} is assigned the label “3”.

- *(lines 6–9). Compute the maximum prefix, postfix and label lengths.*
  If the first digit of N_{left} is “3”, then N_{left} contains a valid prefix and therefore, we must generate N_{new} according to CAGM rules. The prefix length and maximum allowable postfix length of N_{new} must be computed in order to determine the maximum allowable label length of N_{new}. The prefix and maximum allowable postfix lengths are identified by counting the number of consecutive “3” digits at the start of N_{left}, and passing the result as an input parameter to Algorithm 5.4 (`ComputePrefixPostfixLengths`). The maximum allowable label length of N_{new} according to CAR5.5 is computed as the sum of the prefix and maximum allowable postfix lengths.

- *(line 10) Obtain the initial postfix.*
  We know the prefix of N_{new} will always consist of a sequence of “3” digits of length prefixLength. Therefore, all that remains is to determine the postfix of

Algorithm 6.1: SCOOTER `InsertNewNodeAfterRightmostNode`

```
input  : $N_{left}$ - The current rightmost node self_label.
output: $N_{new}$ - A new self_label such that $N_{left} \prec N_{new}$.
begin
    if (first digit in $N_{left}$ is '1') then
        $N_{new} \longleftarrow$ '2';
    else if (first digit in $N_{left}$ is '2') then
        $N_{new} \longleftarrow$ '3';
    else if (first digit in $N_{left}$ is '3') then
        numConsecThrees $\longleftarrow$ the number of consecutive '3' digits at start of $N_{left}$;
        /* Compute the prefixLength and postfixLength based on numConsecThrees.      */
        prefixLength, postfixLength $\longleftarrow$ ComputePrefixPostfixLengths (numConsecThrees);
        labelLength $\longleftarrow$ prefixLength + postfixLength;
        postfix $\longleftarrow$ substring($N_{left}$, prefixLength + 1, labelLength);
        if (postfix is not empty) then
            if (last symbol in postfix is '1') then
                postfix $\longleftarrow$ postfix with last symbol changed to '2';
            else
                postfix $\longleftarrow$ Increment (postfix, postfixLength);
            end
        else if (postfix is empty) then
            postfix $\longleftarrow \emptyset$;
            while (i=1; i < postfixLength; i++) do
                postfix $\longleftarrow$ postfix $\oplus$ 2;
            end
            postfix $\longleftarrow$ postfix $\oplus$ 3;
        end
        prefix = null;
        while (i=1; i $\leq$ prefixLength; i++) do
            prefix $\longleftarrow$ prefix $\oplus$ 3;
        end
        $N_{new} \longleftarrow$ prefix $\oplus$ postfix;
    end
    return $N_{new}$;
end
```

N_{new} which is obtained from N_{left}. However, the length of N_{left} may be longer than the maximum allowable label length of N_{new}, which would indicate one of two possibilities: a very large number of node labels were initially assigned when the XML document was created, or an arbitrary number of nodes labels after N_{left} have been deleted. In either case, the postfix is determined by obtaining the substring of N_{left} from position (prefixLength + 1) upto the maximum allowable length of N_{new}. For example, if the rightmost node label N_{left} = "33332232212", then the prefix is 3333 and the postfix obtained from N_{left} is 223.

- *(lines 11–16). Generate the final postfix when initial postfix is not empty.*

  When the postfix terminates with a "1" digit, the final postfix is assigned the value of the current postfix with the last digit changed to "2". If the postfix terminates with a “2” or “3” digit, the final postfix is assigned the lexicographical increment of the current postfix value. Hence, the postfix will grow according to the growth rate of CAR5.7. **The key point is that once a valid prefix is identified in N_{left}, then the label assigned to N_{new} will always be generated according to CAGM rules**.

- *(lines 17–23). Generate the final postfix when initial postfix is empty.*
  If no postfix can be obtained from N_{temp}, then the postfix is selected to be the label at the lexicographical midpoint of all possible labels of length *postfixLength* (from CAR5.8 and Theorem 5.5). The lexicographical midpoint of all possible labels of length *postfixLength* is $(3^{postfixLength} - 1) / 2$. For example, if N_{left} = “3333”, then the prefix length of N_{new} is 4 and the maximum allowable postfix length of N_{new} is 3. However, we can obtain no postfix substring in N_{left} because the $len(N_{left})$ is equal to the prefixLength. Therefore, the postfix is selected to be the label at the midpoint $(3^{postfixLength} - 1) / 2$. When performing an insertion operation after a node, the label at the midpoint $(3^{postfixLength} - 1) / 2$ will always terminate with a “3” digit and be preceded by a sequence of “2” digits of length postfixLength $-$ 1, (e.g.: 3, 23, 223, 2223 and so on). Hence, if N_{left} = “3333”, then N_{new} = “3333223”.

- *(lines 24–27). Obtain the prefix of N_{new}.*
  The prefix of N_{new} is always generated as a sequence of “3” digits of length *maxPrefixLength.*

- *(line 28). Generate the label N_{new}.*
  Finally, the label N_{new} is created as a concatenation of the prefix and the postfix.

6.1.1 Analysis of `InsertNewNodeAfterRightmostNode` Algorithm

If the first digit of N_{left} is either “1” or “2”, then the first digit of N_{new} is assigned the lexicographical increment of the first digit of N_{left}. Thus, $N_{left} \prec N_{new}$. If the first digit of N_{left} is “3”, then the postfix used in the composition of N_{new} will always be lexicographically greater than the postfix obtained from N_{left}, because the postfix in N_{new} is the lexicographical increment of the postfix obtained from N_{left}. If the postfix in N_{new} was not obtained from N_{left}, N_{new} will always be lexicographically greater than N_{left} because N_{left} will be a prefix of N_{new}. Hence, the `InsertNewNodeAfterRightmostNode` Algorithm generates a new label N_{new} such that $N_{left} \prec N_{new}$.

6.2 Insertion Before the Current Leftmost Node.

Algorithm 6.2 is the SCOOTER `InsertNewNodeBeforeLeftmostNode` Algorithm that supports the reuse of deleted node labels when inserting a new node before the leftmost sibling node. Given one input node label N_{right}, Algorithm 6.2 generates a node label N_{new} such that N_{new} is the lexicographical decrement of the input label. For example, if the current leftmost node label is "2", the sequence of the next 21 new leftmost node labels are illustrated in Table 6.1 (they are ordered from bottom to top as a visual aid to viewing the lexicographical order). The algorithm will always reuse the smallest possible deleted node label if one is available, or if the algorithm detects a valid prefix (a consecutive sequence of "1" digits) at the beginning of N_{right}, then N_{new} will always be assigned a label generated according to CAGM rules, and consequently the label size will grow very slowly. We now walk through the algorithm and provide a detailed explanation.

- *(lines 2–5). Process N_{right} when no valid prefix is detected.*
  If the first digit of the label N_{right} is "3", then N_{new} is assigned the label "2". If the first digit of N_{right} is "2", then N_{new} is assigned the label "12".
- *(lines 6–12). Compute the maximum prefix, postfix and label lengths.*
  If the first digit of N_{right} is "1", then N_{right} contains a valid prefix, and therefore we must generate N_{new} according to CAGM rules. The prefix length and maximum allowable postfix length of N_{new} must be computed in order to determine the maximum allowable label length of N_{new}. The prefix and maximum allowable postfix lengths are identified by counting the number of consecutive "1" digits at the start of N_{right}, and passing the result as an input parameter to Algorithm 5.4 (`ComputePrefixPostfixLengths`). However,

Insert before leftmost node	SCOOTER label
21	11111112213
20	1111111222
19	11111112222
18	1111112
17	1111113
16	111112
15	1111122
14	1111123
13	111113
12	1111132
11	1111133
10	11112
9	1111212
8	1111213
7	111122
6	1111222
5	1112
4	1113
3	112
2	1122
1	12
	2
	3

Table 6.1: SCOOTER Labels When Inserting Before Leftmost Node

when counting the number of consecutive “1” digits at the start of N_{right}, if the first non “1” digit is a “2” digit and is also the last digit in N_{right}, then N_{right} is the lexicographically smallest label of length $len(N_{right})$. Consequently, we need to add one to the number of consecutive “1” digits because the new label must be lexicographically less than the input label. The maximum allowable label length of N_{new} according to CAR5.5 is computed as the sum of the prefix and maximum allowable postfix lengths.

- *(lines 13–14). Reuse a shorter deleted node label if available.*

  If the maximum allowable label length of N_{new} is less than the length of N_{right} **and** if the first *maximumAllowableLabelLength* digits of N_{right} is the smallest lexicographical label of length *maximumAllowableLabelLength* (that is it terminates with a “2” digit and is preceded by a sequence of consecutive “1” digits), then N_{new} is assigned the first *maximumAllowableLabelLength* digits of N_{right} (which as a prefix-string of N_{right}, is always lexicographical less than N_{right}). For example, given N_{right} = 11111121112, then N_{new} = 1111112.

- *(lines 16–18). Generate the postfix from N_{right}.*

Algorithm 6.2: SCOOTER `InsertNewNodeBeforeLeftmostNode`

```
input  : $N_{right}$ - the current leftmost node self_label.
output: $N_{new}$ - a new self_label such that $N_{new} \prec N_{right}$.
begin
    if (first digit in $N_{right}$ is '3') then
        $N_{new} \longleftarrow$ '2';
    else if (first digit in $N_{right}$ is '2') then
        $N_{new} \longleftarrow$ '12';
    else if (first digit in $N_{right}$ is '1') then
        numConsecOnes $\longleftarrow$ the number of consecutive '1' digits at start of $N_{right}$;
        if (numConsecOnes == (length($N_{right}$) − 1) and (last digit in $N_{right}$ is '2')) then
            numConsecOnes $\longleftarrow$ numConsecOnes + 1;
        end
        /* Compute the prefixLength and postfixLength based on numConsecOnes.        */
        prefixLength, postfixLength $\longleftarrow$ ComputePrefixPostfixLengths (numConsecOnes);
        labelLength $\longleftarrow$ prefixLength + postfixLength;
        if (numConsecOnes == (labelLength − 1)) and (length($N_{right}$) > labelLength) then
            $N_{new} \longleftarrow$ substring($N_{right}$, 1, labelLength);
        else
            postfix $\longleftarrow$ substring($N_{right}$, prefixLength + 1, labelLength);
            if (postfix is not empty) then
                postfix $\longleftarrow$ Decrement(postfix, postfixLength);
            else if (postfix is empty) then
                postfix $\longleftarrow \emptyset$;
                while (i=1; i $\leq$ postfixLength; i++) do
                    postfix $\longleftarrow$ postfix $\oplus$ 2;
                end
            end
            prefix = null;
            while (i=1; i $\leq$ prefixLength; i++) do
                prefix $\longleftarrow$ prefix $\oplus$ 1;
            end
            $N_{new} \longleftarrow$ prefix $\oplus$ postfix;
        end
    end
    return $N_{new}$;
end
```

We know the prefix of N_{new} will always consist of a sequence of "1" digits of length prefixLength. Therefore, what remains is to determine the postfix of N_{new}. The postfix is determined by obtaining the substring of N_{right} from position (prefixLength + 1) upto the maximum allowable length of N_{new} inclusive, and decrementing it. For example, if N_{right} = "1113", then the prefixLength of N_{new} is 2 and the maximum allowable postfix length is 2. Therefore, the prefix of N_{new} = "11", and the postfix identified in N_{right} is "13", which when decremented becomes "12". Hence, N_{new} = "1112".

- *(lines 19–24). Generate the postfix if N_{right} contains no postfix.*

  If no postfix can be obtained from N_{right}, then N_{new} will be assigned the first label to have a prefix of length *len(N_{right})*. For example, if N_{right} = "1112", then the prefix length of N_{new} is 4 and the maximum allowable postfix length of N_{new} is 3. However, we can obtain no postfix substring in N_{right}. Therefore, the postfix is selected to be the label at the midpoint ($3^{postfixLength} - 1$) / 2 (from Theorem 5.5 and CAR5.8). When performing an insertion operation *before* a node, the label at the midpoint ($3^{postfixLength} - 1$) / 2 will always consist of a sequence of "2" digits of length *postfixLength*, (e.g.: 2, 22, 222, 2222 and so on). Hence, if N_{right} = "1112", then N_{new} = "1111222".

- *(lines 25–28). Generate the prefix of N_{new}.*
  The prefix of N_{new} will always consist of a sequence of "1" digits of length prefixLength.

- *(line 29). Generate the label N_{new}.*
  Finally, the label N_{new} is created as a concatenation of the prefix and the postfix.

6.2.1 Analysis of `InsertNewNodeBeforeLeftmostNode` Algorithm

If the first digit of N_{right} is either "2" or "3", then the first digit of N_{new} is assigned the lexicographical decrement of the first digit of N_{right}. Thus, $N_{new} \prec N_{right}$. If the first digit of N_{right} is "3", then the postfix used in the composition of N_{new} will always be lexicographically less than the postfix obtained from N_{right} because the postfix in N_{new} is the lexicographical decrement of the postfix obtained from N_{right}. If the postfix in N_{new} was not obtained from N_{right}, then N_{new} will always be lexicographically less than N_{right} because the last digit in N_{right} is a "2" and the corresponding digit in N_{new} is a "1". Hence, the `InsertNewNodeBeforeLeftmostNode` Algorithm generates a new label N_{new} such that $N_{new} \prec N_{right}$.

6.3 Insertion between Two Consecutive Sibling Nodes

A node insertion between two consecutive sibling nodes is the most difficult node insertion scenario. Between any two consecutive nodes, there may have been an arbitrary number of node deletions. The ability to determine whether deletions have occurred must be determined from the information contained in the labels alone. In addition, there are 4 distinct insertion scenarios when inserting a new node between two consecutive sibling node labels:

1. The left label is longer than the right label.
2. The left label is the same length as the right label.
3. The left label is a prefix string of the right label.
4. The left label is shorter than the right label but not a prefix string of the right label.

By exploiting the Compact Adaptive Growth Method, the SCOOTER dynamic labeling scheme provides the same highly constrained growth rate in label size when processing node label insertions in all four scenarios. In the remainder of this section, we present four algorithms (one for each insertion scenario) and highlight some observations.

6.3.1 The `Insertion_LongerThan` Algorithm

Algorithm 6.3 is the `Insertion_LongerThan` Algorithm that inserts a new node between two consecutive sibling nodes when the label of N_{left} is longer than N_{right}. The algorithm takes in two input node labels N_{left} and N_{right} such that: N_{left} is lexicographically less than N_{right} **and** both N_{left} and N_{right} are not empty **and** the length of N_{left} is longer than the length of N_{right}. The algorithm generates a new node label N_{new} such that $N_{left} \prec N_{new} \prec N_{right}$. This algorithm will always reuse the smallest possible deleted node label if one is available, or if the algorithm detects a valid prefix (a consecutive sequence of "3" digits) in N_{left}, then N_{new} will always be assigned a label generated according to CAGM rules, and consequently the label size will grow very slowly. An explanation of the algorithm now follows.

- *(line 3). Determine the first position of difference between N_{left} and N_{right}.* The position at which the digits first differ between the two labels is identified and assigned to P. There is always a position of difference between the two labels because N_{left} is lexicographically less than N_{right}.

Algorithm 6.3: SCOOTER `Insertion_LongerThan`

```
/* Insert node between two consecutive sibling nodes; length($N_{left}$) > length($N_{right}$).
   */
input  : $N_{left}$ - the left node self_label.
         $N_{right}$ - the right node self_label.
output: $N_{new}$ - a new self_label such that $N_{left} \prec N_{new} \prec N_{right}$.
begin
    if (length($N_{left}$) > length($N_{right}$)) then
        P ⟵ first position of difference between $N_{left}$ and $N_{right}$;
        affix ⟵ substring($N_{left}$, 1, P);
        if (Symbol at position P in $N_{left}$ is '1') and (Symbol at position P in $N_{right}$ is '3') then
            $N_{new}$ ⟵ affix with last symbol changed to '2';
        else if (P < length($N_{right}$)) then
            $N_{new}$ ⟵ Increment(affix, length(affix));
        else if (P == length($N_{right}$)) then
            $N_{temp}$ ⟵ substring($N_{left}$, P + 1, length($N_{left}$));
            numConsecthrees ⟵ the number of consecutive '3' digits at start of $N_{temp}$;
            if (numConsecthrees == 0) then
                postfix ⟵ Increment(first symbol of $N_{temp}$, 1);
                $N_{new}$ ⟵ affix ⊕ postfix;
            else if (numConsecthrees > 0) then
                maxPrefixLength, maxPostfixLength ⟵
                ComputePrefixPostfixLengths(numConsecThrees);
                maxLabelLength ⟵ maxPrefixLength + maxPostfixLength;
                postfix ⟵ substring($N_{temp}$, maxPrefixLength + 1, maxLabelLength);
                if (postfix is not empty) then
                    if (last symbol in postfix is '1') then
                        postfix ⟵ postfix with last symbol changed to '2';
                    else
                        postfix ⟵ Increment(postfix, maxPostfixLength);
                    end
                else if (postfix is empty) then
                    postfix ⟵ ∅;
                    while (i=1; i < maxPostfixLength; i++) do
                        postfix ⟵ postfix ⊕ 2;
                    end
                    postfix ⟵ postfix ⊕ 3;
                end
                prefix = null;
                while (i=1; i ≤ maxPrefixLength; i++) do
                    prefix ⟵ prefix ⊕ 3;
                end
                $N_{new}$ ⟵ affix ⊕ prefix ⊕ postfix;
            end
        end
    end
    return $N_{new}$;
end
```

- *(line 4). Identify the shared common digits at start of N_{left} and N_{right}.*

  We assign the first P digits of N_{left} to affix. The affix contains the shared common digits at the beginning of N_{left} and N_{right}. The last digit of the affix also has the digit from N_{left} at the first position of difference between N_{left} and N_{right}. This is necessary to ensure the new label N_{new} will be lexicograpically less than N_{right}.

- *(lines 5–6). Reuse the shortest deleted node label if available.*

  If the digit at position P in N_{left} is "1" and the digit at position P in N_{right} is "3", then we can reuse the shortest possible deleted node label between N_{left} and N_{right} which is the label N_{left} with the digit at position P changed to "2". For example, given $N_{left} = 21232$ and $N_{right} = 23222$, then $N_{new} = 22$.

- *(lines 7–8). Reuse a shorter deleted node label if available.*

  If P is less than $len(N_{right})$, then we know for certain there is at least one deleted node label available for reuse. We know this because the properties of the `AssignInitialLabels_Sequential` Algorithm guarantees that given two consecutive node labels N_{left} and N_{right} such that $len(N_{left}) > len(N_{right})$, then the first position of difference between N_{left} and N_{right} must be $len(N_{right})$, but P is less than $len(N_{right})$. Thus, we reuse a shorter deleted node label which is the lexicographical increment of the affix with a maximum length of P digits.

- *(lines 9–11). Determine the length of the valid prefix in N_{left}.*

  If P is equal to $len(N_{right})$, then we know for certain that there are no deleted node labels available with a length less than or equal to $len(N_{right})$. From this point in the algorithm onward, we will generate the label N_{new} according to CAGM rules. The prefix and maximum allowable postfix lengths must be determined in order to compute the maximum allowable label length of N_{new}. In order to determine the prefix and maximum allowable postfix lengths, we must first determine the length of the valid prefix in N_{left}. The length of the valid prefix in N_{left} is determined by assigning all digits in N_{left} after position P to N_{temp} and counting the number of consecutive "3" digits at the beginning of N_{temp}.

- *(lines 12–14). Generate the shortest label possible if no valid prefix found.*

  If there are no consecutive "3" digits at the beginning of N_{temp}, then we know for certain there is a node label available for use with length $len(N_{right}) + 1$. Given that any new label generated according to CAGM rules will have a length of at least $len(N_{right}) + 2$, the algorithm uses the shorter node label with length $len(N_{right}) + 1$.

- *(lines 15–17). Compute the maximum prefix, postfix and label lengths.*

  If there is a valid prefix (one or more consecutive "3" digits) at the beginning of N_{temp}, then we compute the maximum prefix, postfix and label lengths based on the length of the prefix in N_{temp}.

- *(line 18). Obtain initial postfix from N_{left}.*

  We know the prefix of N_{new} will always consist of a sequence of consecutive "3" digits of length *maxPrefixLength.* Therefore, what remains is to determine the postfix of N_{new}. The postfix is initially obtained from N_{temp} (which was originally created from N_{left}) as the substring from position (maxPrefixLength + 1) up to the maximum allowable label length of N_{new} (*maxLabelLength*). For example, given N_{left} = 323322312 and N_{right} = 33, then P = 2, Affix = 32, N_{temp} = 3322312, maxPrefixLength = 2, maxPostfixLength = 2, and maxLabelLength = 4. Therefore, the prefix of N_{new} = 33, the postfix obtained from N_{temp} is 22.

- *(lines 19–21). Generate final postfix if initial postfix terminates with "1".*

  When the postfix terminates with a "1" digit, the final postfix is assigned the value of the current postfix with the last digit changed to "2" (in order to guarantee the new postfix is lexicographical greater than the input postfix). Observe that the algorithm did not increment the postfix. If the current postfix terminates with a "1" digit and has a length less than *maxPostfixLength*, the `Increment` Algorithm will generate a new postfix value lexicographically less than the input postfix, which is not the desired result (the new postfix should be lexicographically greater than the input postfix). This is not a limitation in the `Increment` Algorithm. The `Increment` Algorithm was explicitly designed to consume all available digits when incrementing a label. The `Insertion_LongerThan` Algorithm should never invoke the `Increment` Algorithm, with an input label consisting of a sequence of "1" digits **and** with a length less than *maxPostfixLength.* However, the `Increment` Algorithm can successfully process an input label consisting of a sequence of "1" digits with a length equal to *maxPostfixLength.*

- *(lines 22–24). Generate final postfix if initial postfix terminates with "2" or "3".*

  If the postfix terminates with a "2" or "3" digit, the final postfix is assigned the lexicographical increment of the current postfix value. Hence, the postfix will grow according to the growth rate of CAGM rules.

- *(lines 25–31). Generate the final postfix if initial postfix is empty.*

  If no postfix can be obtained from N_{temp}, then the first postfix to have a prefix of length *maxPrefixLength* will be assigned. For example, given $N_{left} = 3233$ and $N_{right} = 33$, then P = 2, Affix = 32, $N_{temp} = 33$, maxPrefixLength = 2, maxPostfixLength = 2, and maxLabelLength = 4, prefix = 33, and the postfix = 23. The postfix is always selected to be the label at the midpoint $(3^{maxPostfixLength} - 1) / 2$ of all possible labels with a length less than or equal to *maxPostfixLength* (from Theorem 5.5 and CAR5.8).

- *(lines 32–35). Generate the prefix of N_{new}.*

  The prefix is always generated as a sequence of "3" digits of length *maxPrefixLength.*

- *(line 36). Generate the label N_{new}.*

  Finally, the label N_{new} is created as a concatenation of the affix with the prefix and the postfix.

Analysis of `Insert_LongerThan` Algorithm

Recall that the affix contains the shared common digits at the beginning of N_{left} and N_{right}. The last digit of the affix also has the digit from N_{left} at the first position of difference between N_{left} and N_{right}, so as to guarantee that N_{new} (which is constructed using the affix) will be lexicographically less than N_{right}. The postfix used in the composition of N_{new} will always be lexicographically greater than the postfix obtained from N_{left} because the postfix in N_{new} is the lexicographical increment of the postfix obtained from N_{left}. If the postfix in N_{new} was not obtained from N_{left}, then N_{new} will always be lexicographically greater than N_{left} because N_{left} is a prefix of N_{new}. Hence, the `Insertion_LongerThan` Algorithm generates a new label N_{new} such that $N_{left} \prec N_{new} \prec N_{right}$.

6.3.2 The `Insertion_EqualTo` Algorithm

Algorithm 6.4 is the `Insertion_EqualTo` Algorithm that inserts a new node between two consecutive sibling nodes when both input labels have the same length. The algorithm takes two input parameters - node labels N_{left} and N_{right} such that: N_{left} is lexicographically less than N_{right} **and** both N_{left} and N_{right} are not empty **and** the lengths of both labels are the same. The algorithm generates a new node label N_{new} such that $N_{left} \prec N_{new} \prec N_{right}$. This algorithm is designed to reuse a shorter deleted node label if available. If there is no shorter deleted node label available, it will generate a label that is at most 1 digit longer than the length of the input labels.

Algorithm 6.4: SCOOTER `Insertion_EqualTo`

```
   /* Insert node between two consecutive sibling nodes; length($N_{left}$) = length($N_{right}$).
      */
   input  : $N_{left}$ - the left node self_label.
            $N_{right}$ - the right node self_label.
   output: $N_{new}$ - a new self_label such that $N_{left} \prec N_{new} \prec N_{right}$.
1  begin
2      if (length($N_{left}$) == length($N_{right}$)) then
3          P ⟵ first position of difference between $N_{left}$ and $N_{right}$;
4          if (P < length($N_{right}$)) then
5              affix ⟵ substring($N_{left}$, 1, P);
6              $N_{new}$ ⟵ Increment(affix, length(affix));
7          else if (P == length($N_{right}$)) then
8              $N_{new}$ ⟵ $N_{left} \oplus 2$;
9          end
10     end
11     return $N_{new}$;
12 end
```

- *(line 3). Determine the first position of difference between N_{left} and N_{right}.*
  The position at which the digits first differ between the two labels is identified and assigned to P.

- *(lines 4–6). Reuse the shortest deleted node label if available.*
  We first check if we can reuse a deleted node label with a length less than

$len(N_{left})$. If the first position of difference between N_{left} and N_{right} is not the last digit, and if both labels have the same length, then we are certain there is at least one deleted node label available for reuse because of the properties of the `AssignInitialLabels_Sequential` Algorithm. The `AssignInitialLabels_Sequential` Algorithm guarantees that given two consecutive sibling node labels N_{left} and N_{right} such that $len(N_{left}) = len(N_{right})$, then the first position of difference between N_{left} and N_{right} must be $len(N_{right})$. Even if an arbitrary number of nodes are inserted between N_{left} and N_{right}, the first position of difference between any newly inserted node and N_{right} must be at position $len(N_{right})$. However, the first position of difference is not at $len(N_{right})$. Therefore, we reuse the shortest deleted node label possible between N_{left} and N_{right} which is obtained by incrementing the substring of N_{left} from the first digit up to and including the digit at position P.

- *(lines 7–9). Generate the shortest possible label between N_{left} and N_{right}.* Given that the first position of difference between N_{left} and N_{right} is the last digit, and given that both labels have the same length, and given that a label must terminate with a "2" or a "3" digit, then we know for certain that there are no deleted node labels available for use between N_{left} and N_{right}. Therefore, we generate and assign to N_{new} the shortest possible label between N_{left} and N_{right}, which is N_{left} with a "2" digit concatenated at the end.

Analysis of `Insertion_EqualTo` Algorithm

If the position of difference between the two input labels is not the last digit in the labels, then:

- N_{new} must be lexicographical greater than N_{left} because it is generated by incrementing the digit at position P in N_{left}.

- N_{new} must be lexicographical less than N_{right} because the lexicographical increment operation will generate the prefix of N_{right} (which is always lexicographical less than N_{right}) before it can generate N_{right} itself. However, only one lexicographical increment operation is performed and thus, N_{new} must be lexicographical less than N_{right}.

If the position of difference between the two input labels is the last digit in both labels, then:

- N_{new} must be lexicographical greater than N_{left} because N_{left} is the prefix of N_{new}.
- N_{new} must be lexicographical less than N_{right} because the digit at position P in N_{new} is lexicographical less than the digit at position P in N_{right}.

Hence, the `Insertion_EqualTo` Algorithm generates a new label N_{new} such that $N_{left} \prec N_{new} \prec N_{right}$.

6.3.3 The `Insertion_LessThanPrefix` Algorithm

Algorithm 6.5 is the `Insertion_LessThanPrefix` Algorithm that inserts a new node between two consecutive sibling nodes when one label is the prefix of the other. The algorithm takes in two input node labels N_{left} and N_{right} such that:

- N_{left} is lexicographically less than N_{right}, and
- Both N_{left} and N_{right} are not empty, and
- len(N_{left}) < len(N_{right}), and
- N_{left} is a prefix of N_{right}.

The algorithm generates a new node label N_{new} such that $N_{left} \prec N_{new} \prec N_{right}$. This algorithm will always assign a label to N_{new} generated according to CAGM rules, and consequently the label size will grow very slowly. We now present a detailed description of the algorithm.

- *(lines 3–9). Determine the length of the valid prefix in N_{right}.*

  N_{temp} is the string remaining when N_{left} is removed from N_{right}. We count the number of consecutive "1" digits at the beginning of N_{temp} and store the count

Algorithm 6.5: SCOOTER `Insertion_LessThanPrefix`

```
/* Insert node between two consecutive sibling nodes; $N_{left}$ is a prefix of $N_{right}$.   */
input  : $N_{left}$ - the left node self_label.
         $N_{right}$ - the right node self_label.
output: $N_{new}$ - a new self_label such that $N_{left} \prec N_{new} \prec N_{right}$.
begin
    if (length($N_{left}$) < length($N_{right}$)) and ($N_{left}$ is a prefix of $N_{right}$) then
        $N_{temp}$ ⟵ substring($N_{right}$, length($N_{left}$) + 1, length($N_{right}$));
        numConsecOnes ⟵ the number of consecutive '1' digits at start of $N_{temp}$;
        if (digit at Ntemp[numConsecOnes + 1] is '2') then
            if (length($N_{left}$) + numConsecOnes) == (length($N_{right}$) − 1) then
                numConsecOnes ⟵ numConsecOnes + 1;
            end
        end
        minLabelLength ⟵ length($N_{left}$) + numConsecOnes + 1;
        maxPrefixLength, maxPostfixLength ⟵ ComputePrefixPostfixLengths(minLabelLength
        − 1);
        maxLabelLength ⟵ maxPrefixLength + maxPostfixLength;
        prefix ⟵ $N_{left}$;
        for (i = length($N_{left}$); i < maxPrefixLength; i++) do
            prefix ⟵ prefix ⊕ 1;
        end
        actualPrefixLength ⟵ length(prefix);
        allowablePostfixLength ⟵ maxLabelLength − actualPrefixLength;
        postfixDefault ⟵ a string of '2' digits of length allowablePostfixLength;
        postfix ⟵ substring($N_{right}$, actualPrefixLength + 1, maxLabelLength);

        if (length(postfix) == 0) then
            postfix ⟵ postfixDefault;
        else if (length(postfix) > 0) then
            numOnes ⟵ the number of consecutive '1' digits at start of postfix;
            if (numOnes == length(postfix) − 1) and (last digit in postfix is '2')
            and (length(postfix) == allowablePostfixLength) then
                postfix ⟵ postfix;
            else
                postfix ⟵ generateAdaptivePostfix(postfix, allowablePostfixLength);
            end
        end
        $N_{new}$ ⟵ prefix ⊕ postfix;
    end
    return $N_{new}$;
end
```

as numConsecOnes. We need to know the number of consecutive "1" digits at the beginning of N_{temp} because the new label N_{new} must be lexicographically less than N_{right}. Also, if the first non "1" digit in N_{temp} is a "2" digit, and is the last digit in N_{temp}, then we add one to numConsecOnes in order to ensure N_{new} will be lexicographically less than N_{right}.

- *(line 10). Compute the minimum label length of N_{new}.*

  Before we can begin constructing the new label N_{new}, the minimum label length of N_{new} must be computed. The minimum label length of N_{new} is computed as the sum of four components:

1. The length of the input label N_{left}.
2. The number of consecutive "1" digits commencing at position $len(N_{left})$ + 1 in label N_{right} (i.e.: numConsecOnes). numConsecOnes may be zero.
3. One extra digit must be added because the new label must be at least 1 digit longer than N_{left} (recall that N_{left} is a prefix of N_{right}).
4. A further digit, if and only if there is a "2" digit appearing immediately after the sequence of consecutive "1" digits identified in label N_{right}. This extra symbol is required because N_{new} cannot terminate with a "1" digit.

- *(lines 11–12) Compute the maximum prefix, postfix and label lengths.*

  The minimum label length is the input parameter to the `ComputePrefixPostfixLengths` Algorithm to determine the maximum prefix length and maximum allowable postfix length of N_{new}. The maximum label length of N_{new} is always computed as the sum of the maximum prefix length and maximum allowable postfix length of N_{new}.

- *(lines 13–16). Generate the prefix of N_{new}.*

  The prefix of N_{new} must be initialized to N_{left}, because N_{left} is a prefix of N_{right}, and N_{new} must be lexicographically less than N_{right}. However, the length of N_{left} may be less than the maximum allowable prefix length *maxPrefixLength.* If the length of N_{left} is less than *maxPrefixLength*, then we must continue to append "1" digits to the prefix until it reaches *maxPrefixLength.* For example, given $N_{left} = 313$, and $N_{right} = 31311112$, then *maxPrefixLength* = 7 and *maxPostfixLength* = 4. Consequently, the prefix is initialized to N_{left} (313) and then extended with a sequence of "1" digits up to length *maxPrefixLength.* Hence, the prefix becomes 3131111.

- *(lines 17–18). Recalibrate the maximum postfix length.*

  We are now certain that the length of the prefix is at least equal to *maxPrefixLength.* However, the length of the prefix may be greater than *maxPrefixLength* because the prefix is always initialized to N_{left} and the length of N_{left} may be greater than *maxPrefixLength.* If the length of the prefix is greater than *maxPrefixLength*, then we must adjust the size of the maximum allowable postfix length of N_{new} to ensure the length of N_{new} will never be larger than the maximum label length. For example, given $N_{left} = 31312$, and $N_{right} = 313122$, then *maxPrefixLength* $= 4$ and *maxPostfixLength* $= 3$. However, the prefix $= 31312$ (initialized to N_{left}) and consequently the *actualPrefixLength* $= 5$ and the *allowablePostfixLength* $= 2$.

- *(lines 19–22). Obtain the postfix from N_{right}.*

  The default postfix string is initialized to a sequence of "2" digits of length *allowablePostfixLength.* We extract a postfix string from the label N_{right}. We deploy the default postfix string if and only if no postfix string can be obtained from the label N_{right}. For example, given $N_{left} = 313$ and $N_{right} = 3132$, then *maxPrefixLength* $= 4$ and *maxPostfixLength* $= 3$. The prefix is initialized to N_{left} and extended to 3131. Hence, no postfix can be obtained from N_{right}, therefore the postfix is assigned the value of the default postfix string. Thus, the postfix $= 222$ and $N_{new} = 3131222$

- *(lines 23–27). Determine if N_{new} can use the postfix obtained from N_{right}.*

  We have obtained a postfix string from N_{right}. If the current postfix has the maximum allowable postfix length and consists of a sequence of zero or more consecutive "1" digits followed by and terminating with a single "2" digit, then there does not exist a postfix lexicographical less than the input postfix and with a length less than or equal to *allowablePostfixLength.* However, the maximum label length of N_{new} must be less than the length of N_{right} because the current postfix can not have the same length as *allowablePostfixLength.* The current postfix can not have the same length as *allowablePostfixLength* when the *maxLabelLength* of N_{new} is greater than or equal to the length of N_{right} by virtue of lines (5–10). Thus, we can simply use the postfix extracted from N_{right} because N_{new} will be a prefix of N_{right} and consequently N_{new} will be lexicographically less than N_{right}. For example, given $N_{left} = 3133$ and $N_{right} = 3133112112$, then N_{new} becomes 3133112.

- *(lines 28–30). Generate the adaptive postfix.*

In all other cases when the current postfix is not empty (has at least one digit), we generate a new postfix by invoking the `generateAdaptivePostfix` Algorithm (presented in the next section). This algorithm will guarantee that the postfix has a growth rate according to CAGM rules and will always generate a new postfix such that it is lexicographically less than the input postfix (obtained from N_{right}).

- *(line 32). Generate the label N_{new}.*

  Finally, the label N_{new} is created as a concatenation of the prefix and the postfix.

Analysis of `Insertion_LessThanPrefix` Algorithm

The label N_{new} will be lexicographically greater than N_{left} because N_{left} will always be a prefix of N_{new}. The label N_{new} will be lexicographically less than N_{right} because the postfix generated for N_{new} will always be lexicographically less than the corresponding digits in N_{right} (or the label N_{new} will be a prefix of N_{right}). Hence, the `Insertion_LessThanPrefix` Algorithm generates a new label N_{new} such that $N_{left} \prec N_{new} \prec N_{right}$.

6.3.4 The `generateAdaptivePostfix` Algorithm

Algorithm 6.6 is the `generateAdaptivePostfix` Algorithm invoked by the `Insertion_LessThanPrefix` Algorithm. This algorithm receives two input parameters: a postfix (extracted from N_{right} by the `Insertion_LessThanPrefix` Algorithm), and the maximum allowable length of the new postfix (determined by the Compact Adaptive Growth Method). The algorithm generates a new postfix such that it is lexicographically less than the input postfix. Specifically, the new postfix is generated such that it is lexicographically centered between the most significant digit of the input postfix and the least (smallest) lexicographical postfix possible. However, if the most significant digit of the input postfix is “1”, then the new postfix is obtained by lexicographically decrementing the input postfix. This is to ensure the adaptive postfix has a growth rate dictated by CAGM rules. It is worth noting that the `Insertion_LessThanPrefix` Algorithm ensures this algorithm never receives an input postfix that is the smallest lexicographical postfix with the maximum allowable postfix length (e.g.: 2, 12, 112, 1112 and so on). We now provide a detailed explanation of the algorithm.

Algorithm 6.6: SCOOTER `generateAdaptivePostfix`

```
/* Generate a new postfix according to the compact adaptive growth method.  */
input  : postfix - a postfix (extracted from $N_{right}$ by the Insertion_LessThanPrefix algorithm).
         maxPostfixLength - the maximum allowable length of the new postfix.
output: newPostfix - a new postfix that is lexicographically less than the input postfix.
1  begin
2      if (first digit in postfix is '3') then
3          newPostfix $\longleftarrow$ 2;
4      else if (first digit in postfix is '2') then
5          newPostfix $\longleftarrow$ 1 $\oplus$ a sequence of '2' digits of length (maxPostfixLength $-$ 1);
6      else if (first digit in postfix is '1') then
7          newPostfix $\longleftarrow$ Decrement(postfix, maxPostfixLength);
8      end
9      return newPostfix;
10 end
```

- *(lines 2–3). Generate the adaptive postfix when most significant digit is "3".* If the most significant digit of the input postfix is "3", the new postfix is generated such that it is lexicographically centered between the smallest lexicographical postfix possible and "3". Thus, the new postfix = "2".

- *(lines 4–5). Generate the adaptive postfix when most significant digit is "2".* Given that the first digit of the input postfix is "2", we know for certain that *maxPostfixLength* $\geq$ 2, because a valid postfix lexicographically less than "2" must have at least 2 digits. If *maxPostfixLength* were equal to 1, then this algorithm would never have been invoked by the `Insertion_LessThanPrefix` Algorithm with a postfix = "2". The postfix lexicographically centered between "2" and the smallest lexicographical postfix possible is always a sequence of "2" digits of length *maxPostfixLength* $-$ 1 preceded by a "1" digit (e.g.: 12, 122, 1222, 12222 and so on).

- *(lines 6–7). Generate the adaptive postfix when most significant digit is "1".* If the input postfix begins with a "1" digit, then the new postfix is the lexicographical decrement of the input postfix. This ensures the postfix has a growth rate that corresponds to CAGM rules. Furthermore, frequently skewed insertions are processed gracefully.

6.3.5 The `Insertion_LessThanNotPrefix` Algorithm

Algorithm 6.7 is the `Insertion_LessThanNotPrefix` Algorithm that inserts a new node between two consecutive sibling nodes when one label is shorter than the other, but not a prefix of the other. The algorithm receives two input parameters - node labels N_{left} and N_{right} such that:

- N_{left} is lexicographically less than N_{right}, and
- N_{left} and N_{right} are not empty, and
- len(N_{left}) < len(N_{right}), and
- N_{left} is **not** a prefix of N_{right}.

The algorithm generates a node label N_{new} such that $N_{left} \prec N_{new} \prec N_{right}$. This algorithm will always reclaim and reuse a deleted node label and ensure that the reused label will have a maximum label size determined by CAGM rules. An explanation of the algorithm now follows.

- *(line 2–3). Determine the first position of difference between N_{left} and N_{right}.* The position at which the digits first differ between the two labels is identified and assigned to P. We know that the digit at position P in N_{left} can never be a "3", because "3" is the lexicographically largest digit that can appear at position P in N_{right} and P is the position of difference between the two labels. Given that N_{left} is shorter than N_{right}, and given that N_{left} is not a prefix of N_{right}, then we know for certain there is at least one deleted node label available for reuse between N_{left} and N_{right}. For example, a label consisting of the first P digits of N_{right} will always be lexicographically greater than N_{left} (because the digit at position P in both labels differ) and lexicographically less than N_{right} (because the length of N_{right} is longer than P) and therefore, the label will be available for reuse. Thus, this algorithm will always reuse a deleted node label.

Algorithm 6.7: SCOOTER `Insertion_LessThanNotPrefix`

```
/* Insert node between two consecutive sibling nodes; length($N_{left}$) < length($N_{right}$);
   $N_{left}$ not a prefix of $N_{right}$.                                                   */
input  : $N_{left}$ - the left node self_label.
         $N_{right}$ - the right node self_label.
output: $N_{new}$ - a new self_label such that $N_{left} \prec N_{new} \prec N_{right}$.
begin
    if (length($N_{left}$) < length($N_{right}$)) and ($N_{left}$ is NOT a prefix of $N_{right}$) then
        P $\longleftarrow$ first position of difference between $N_{left}$ and $N_{right}$;
        if (P == 1) then
            Nnew $\longleftarrow$ Increment(first digit in $N_{left}$, 1);
        else if (P > 1) then
            maxPrefixLength $\longleftarrow$ maxPostfixLength $\longleftarrow$ 1;
            maxLabelLength $\longleftarrow$ maxPrefixLength + maxPostfixLength;
            while (maxLabelLength < P) do
                maxPrefixLength $\longleftarrow$ maxLabelLength;
                maxPostfixLength $\longleftarrow$ maxPostfixLength + 1;
                maxLabelLength $\longleftarrow$ maxPrefixLength + maxPostfixLength;
            end
            prefix $\longleftarrow$ substring($N_{left}$, 1, P $-$ 1);
            actualPrefixLength $\longleftarrow$ length(prefix);
            actualPostfixLength $\longleftarrow$ maxLabelLength $-$ actualPrefixLength;
            $N_{temp}$ $\longleftarrow$ substring($N_{left}$, P, P + (actualPostfixLength $-$ 1));
            postfix $\longleftarrow$ Increment($N_{temp}$, actualPostfixLength);
            $N_{new}$ $\longleftarrow$ prefix $\oplus$ postfix;
        end
    end
    return $N_{new}$;
end
```

- *(lines 4–5). Generate N_{new} when labels do not have shared prefix digits.*

  In the scenario where there are no shared prefix digits between N_{left} and N_{right}, then the maximum label size of N_{new} must be 1 digit (the shortest label), and therefore the value of N_{new} is obtained by lexicographically incrementing the first digit in N_{left}. For example, given N_{left} = 2 and N_{right} = 32, then N_{new} = 3.

- *(lines 6–13). Compute maximum prefix, postfix and label lengths.*

  Given that P is greater than 1, we know the shared prefix digits between N_{left} and N_{right} have length P$-$1. Thus, we compute the maximum prefix length of N_{new} based on the length of the shared prefix between N_{left} and N_{right}. The maximum prefix length of N_{new} is used to determine the maximum postfix length and maximum label length of N_{new}.

- *(lines 14–15). Generate the prefix of N_{new}.*

  The prefix of N_{new} can be obtained from either N_{left} or N_{right} and consists of the shared digits at the beginning of N_{left} and N_{right}.

- *(lines 16–18). Generate the postfix of N_{new}.*

  The length of the prefix string of N_{new} may be longer than the maximum prefix length. Therefore, it is necessary to compute the actual postfix length by subtracting the actual prefix length from the maximum allowable label length of N_{new}. The postfix is initially obtained from N_{left} starting from the first position of difference P until the *allowablePostfixLength* − 1 (inclusive). The final postfix is the lexicographical increment of the initial postfix.

- *(line 19). Generate the label of N_{new}.*

  Finally, the label N_{new} is created as a concatenation of the prefix and the postfix.

Analysis of `Insertion_LessThanNotPrefix` Algorithm

When N_{left} and N_{right} do not have shared prefix digits at the beginning of their labels (the first position of difference between the two labels is the first digit), then:

- The label N_{new} is guaranteed to be lexicographically greater than N_{left} because N_{new} is the lexicographical increment of the first digit in N_{left}.

- The label N_{new} is guaranteed to be lexicographically less than N_{right} because, given that N_{left} is shorter than N_{right} and given that the first digit differs in both labels, then a minimum of two lexicographical increment operations on N_{left} are required to generate N_{right}; the first increment operation must generate the prefix of N_{right} before a second increment operation can generate N_{right} itself. However, only one lexicographical increment operation is performed on N_{left} when N_{left} and N_{right} do not have shared prefix digits. Thus, the label N_{new} is guaranteed to be lexicographically less than N_{right}

When N_{left} and N_{right} have shared prefix digits at the beginning of their labels, then:

- The label N_{new} is guaranteed to be lexicographically greater than N_{left} because the prefix of N_{new} is obtained from N_{left} and the postfix of N_{new} is the lexicographical increment of the corresponding digits in N_{left}.

- The label N_{new} is guaranteed to be lexicographically less than N_{right} because N_{left} must first be lexicographically incremented to be a prefix of N_{right}, before it can be lexicographically incremented to be equal to N_{right}. However, this algorithm only performs one lexicographical increment operation on the postfix obtained from N_{left} and used in the construction of the label N_{new}. Consequently, the label N_{new} is guaranteed to be lexicographically less than N_{right}.

Hence, the `Insertion_LessThanNotPrefix` Algorithm generates a node label N_{new} such that $N_{left} \prec N_{new} \prec N_{right}$.

6.4 Summary

In this Chapter, we presented a suite of algorithms to enable dynamic node label insertions in an XML tree. The algorithms were designed to exploit the Compact Adaptive Growth Method presented in the previous Chapter, and to guarantee a highly constrained label size growth rate under various node insertion scenarios, including frequently skewed insertions. Furthermore, all of the algorithms support the reuse of deleted node labels.

In Chapter 8, we will present a comprehensive experimental evaluation of the SCOOTER node insertion algorithms presented in this Chapter. However, in Chapter 7, we first will present a binary-encoded adaption of the Compact Adaptive Growth Method and a new binary-encoded bit-string dynamic labeling scheme that demonstrates the compact adaptive growth method is completely independent of the quaternary encoding employed in this Chapter. We will also present in Chapter 7 a new label storage scheme to enable binary-encoded bit-string dynamic labeling schemes to completely avoid the relabeling of nodes under arbitrary node insertions and deletions scenarios.

Chapter 7

A Scalable Binary-Encoded Dynamic Labeling Scheme

In the previous two Chapters, we presented the Compact Adaptive Growth Method (CAGM) - a method for generating labels with a highly constrained growth rate, and SCOOTER - a dynamic labeling scheme for XML that exploits the CAGM to provide dynamic node label insertions and deletions while maintaining a highly constrained label size. However, SCOOTER completely avoids the relabeling of nodes by employing the separator label storage scheme. The use of the separator label storage scheme was made possible because SCOOTER uses quaternary-encoded labels. To date, there does not exist a binary-encoded bit-string dynamic labeling scheme that can completely avoid the relabeling of nodes.

This Chapter is structured as follows: In §7.1, we introduce the binary-encoded version of our SCOOTER dynamic labeling scheme which we call SCOOBER. In §7.2, we introduce two `AssignInitialLabels` Algorithms that describes how XML trees are initially labeled. In §7.3, we present the Compact Adaptive Growth Method for a *binary* encoding of labels. We introduce our comprehensive theoretical evaluation of the Compact Adaptive Growth Method in §7.4 and the SCOOBER node insertion algorithms in §7.5. Finally, as one does not exist, in §7.6, we introduce a scalable label storage scheme for binary-encoded bit-string labels based on the Fibonacci sequence that enables SCOOBER to completely avoid the relabeling of nodes under any update scenario.

7.1 The SCOOBER Dynamic Labeling Scheme

The SCOOBER dynamic labeling scheme may be described as the binary-encoded version of the quaternary-encoded SCOOTER dynamic labeling scheme. The name SCOOBER, encapsulates the core properties - Scalable, Compact, Ordered, Orthogonal, Binary-Encoded, Reusable dynamic labeling scheme. The labeling scheme shares all of the properties and advantages of the SCOOTER dynamic labeling scheme except scalability, which is made possible in SCOOTER by employing the separator label storage scheme. SCOOBER supports scalability by employing the Fibonacci label storage scheme to be introduced in §7.6.

There are several reasons why we choose to support binary-encoded bit-string labels.

1. The QED and SCOOTER labeling schemes are the only bit-string dynamic labeling schemes to date that employ a quaternary encoding of labels. However, there are several bit-string dynamic labeling schemes that employ a binary-encoding of labels. The properties and concepts underpinning the Compact Adaptive Growth Method and the SCOOTER labeling scheme are not specific to SCOOTER or the quaternary encoding. Hence, it is important for the wider applicability of the contribution in this dissertation that they be shown to work for binary-encoded bit-string labels.

2. Although we highlighted in §5.2.5 that the base 3 is the most compact radix or base, there is a computational overhead in converting binary or decimal to base 3 and back.

3. When processing a bit-string label encoded using the separator label storage scheme, the label must be read until a separator is encountered in order to determine and identify each individual component of the label. A binary-encoding of bit-string labels can take advantage of existing bit-level and byte-level optimizations when performing label comparison operations and consequently can determine structural relationships between labels more quickly.

7.1.1 SCOOBER Lexicographical Order

SCOOBER compares node labels using lexicographical order.

Definition 7.1. *(Lexicographical order) Given two binary strings S_{left} and S_{right} (S_{left} represents the left string, S_{right} represents the right string), S_{left} is said to be lexicographical equal to S_{right} iff they are identical. S_{left} is said to be lexicographically less than S_{right} ($S_{left} \prec S_{right}$) iff:*

1. *the comparison of S_{left} and S_{right} is bit by bit from left to right. If the current bit of S_{left} is less than the current bit of S_{right} then $S_{left} \prec S_{right}$ and stop the comparison, or*
2. *S_{left} is a prefix of S_{right}.*

Example 7.1. *Given two binary strings 101 and 111, $101 \prec 111$ because the comparison is from left to right and the 2^{nd} bit of 101 is a zero and the 2^{nd} bit of 111 is 1. Given two binary strings 101 and 1011, $101 \prec 1011$ because 101 is a prefix of 1011. Given two binary strings 11 and 011, $11 \succ 011$ because the comparison is from left to right and the 1^{st} bit of 11 is a one and the 1^{st} bit of 011 is a zero.*

A SCOOBER label must end in a "1" bit in order to maintain lexicographical order in the presence of dynamic insertions.

7.2 Assigning Initial Labels

In this section, we present the binary-encoded equivalent of the two algorithms for assigning labels at document initialization - a sequential assignment algorithm and a random access assignment algorithm. We first present the rules governing the assignment of SCOOBER labels.

7.2.1 Rules for Assigning Labels

There are a small number of rules used to determine the assignment of labels in order to ensure a compact label size and to maintain lexicographical order between labels.

Column 1	Column 2	Column 3	Column 4	Column 5	Column 6	Column 7
Decimal	SCOOBER all labels with max length of 1 bit	SCOOBER all labels with max length of 2 bits	SCOOBER all labels with max length of 3 bits	SCOOBER all labels with max length of 4 bits	SCOOBER assign 20 labels	CDBS assign 20 labels
1	1	01	001	0001	0001	00001
2		1	01	001	001	0001
3		11	011	0011	0011	001
4			1	01	01	00101
5			101	0101	0101	0011
6			11	011	011	01
7			111	0111	0111	01001
8				1	1	0101
9				1001	1001	011
10				101	101	0111
11				1011	1011	1
12				11	10111	10001
13				1101	11	1001
14				111	11001	101
15				1111	1101	1011
16					11011	11
17					111	11001
18					11101	1101
19					1111	111
20					11111	1111
Total Size					74 bits	74 bits

Table 7.1: SCOOBER and CDBS Sample Labels

SR$_b$ 7.1. *The first label consists of a sequence of "0" bits of length k and terminates with a "1" bit, where the length of the label is k + 1.*

SR$_b$ 7.2. *The first label will always be the maximum allowable length.*

SR$_b$ 7.3. *No label can terminate with a "0" bit.*

SR$_b$ 7.4. *The second and remaining labels can be of any allowable length.*

In Table 7.1, columns 2, 3, 4 and 5 illustrate the labels assigned by the SCOOBER `AssignInitialLabels` Algorithms when the maximum label length is 1, 2, 3, and 4 bits respectively. Columns 6 and 7 illustrate the labels generated by the `AssignInitialLabels` Algorithms of the SCOOBER and CDBS labeling schemes respectively when initially labeling 20 child nodes (they are presented to facilitate a comparison).

7.2.2 The `AssignInitialLabels_Sequential` Algorithm

Algorithm 7.1 is the SCOOBER `AssignInitialLabels_Sequential` Algorithm. The algorithm is functionally identical to the SCOOTER `AssignInitialLabels_Sequential` Algorithm, except in this algorithm, we generate labels encoded in the base 2, instead of the base 3. The total number of labels that may be assigned with a maximum length of size *maxLabelSize* is computed using the formula: maxLabels $= 2^{maxLabelSize} - 1$.

Algorithm 7.1: SCOOBER `AssignInitialLabels_Sequential`

```
/* Assign unique labels to all child nodes of a parent node.                 */
input  : nodeCount - the number of child nodes to be labeled.
output: A unique self_label (binary string) for each child node.
begin
    labelList ⟵ an empty array;
    maxLabelSize ⟵ Ceiling($\log_2$(nodeCount + 1));
    maxLabels ⟵ $2^{maxLabelSize} - 1$;
    numShorterLabels ⟵ maxLabels − nodeCount;
    if (numShorterLabels > 0) then
        labelSize ⟵ maxLabelSize − 1;
        numRemainingLabels ⟵ nodeCount − numShorterLabels;
    else
        labelSize ⟵ maxLabelSize;
        numRemainingLabels ⟵ nodeCount − 1;
    end
    /* Compute the self_label of the first child.                            */
    self_label ⟵ ∅;
    for (i=1; i < labelSize; i++) do
        self_label ⟵ self_label ⊕ 0 ;              // ⊕ denotes concatenation
    end
    self_label ⟵ self_label ⊕ 1;
    labelList.add(self_label);
    /* Now compute the self_labels for all subsequent children.              */
    for (i=1; i < numShorterLabels; i++) do
        self_label ⟵ Increment(self_label, labelSize);
        labelList.add(self_label);
    end
    for (i=1; i ≤ numRemainingLabels; i++) do
        self_label ⟵ Increment(self_label, maxLabelSize);
        labelList.add(self_label);
    end
    return labelList;
end
```

Half of all labels available will be of length *maxLabelSize* with the other half having shorter lengths varying from one digit to (maxLabelSize − 1). Specifically, given a maximum label size of length *maxLabelSize*, there are $2^1 - 1$ labels of length 1, $2^2 - 2^1$ labels of length 2, $2^3 - 2^2$ labels of length 3,. . . , $2^{maxLabelSize} - 2^{maxLabelSize-1}$

labels of length *maxLabelSize*.

In a similar way to SCOOTER, we designed our `AssignInitialLabels_Sequential` Algorithm to guarantee that all shorter labels are assigned. A shorter label is any SCOOBER label with a length less than the maximum allowable length. The three properties that are exploited by the `AssignInitialLabels_Sequential` Algorithm to achieve this goal are now presented.

Theorem 7.1. *The first label assigned by the `AssignInitialLabels_Sequential` Algorithm will always be lexicographically less than any other label assigned by the `AssignInitialLabels_Sequential` Algorithm.*

Proof: The lexicographically least string encoding using the two bits "0" and "1" is the string consisting of a single "0" bit. However, a SCOOBER label can never terminate with a "0" bit (from SR_b7.3). Hence, the lexicographically least string encoding is the string of length D bits that terminates with a "1" bit and is preceded with a sequence of "0" bits of length (D $-$ 1). The `AssignInitialLabels_Sequential` Algorithm always generates the first label such that it is a string of length D bits that terminates with a "1" bit and is preceded with a sequence of "0" bits of length (D $-$ 1). Hence, the first label assigned will always be lexicographically less than any other label assigned by the `AssignInitialLabels_Sequential` Algorithm. □

Theorem 7.2. *The n^{th} label assigned by the SCOOBER `AssignInitialLabels_Sequential` Algorithm will always be lexicographically greater than the $(n-1)^{th}$ label assigned by the `AssignInitialLabels_Sequential` Algorithm.*

Proof: From Theorem 7.1, we know that the first label assigned by the `AssignInitialLabels_Sequential` Algorithm is always the lexicographical smallest label. Thereafter, the second and subsequent label assigned by the `AssignInitialLabels_Sequential` Algorithm is generated by lexicographically incrementing the label immediately preceding it. Thus, the n^{th} label assigned will always be lexicographically greater than the $(n-1)^{th}$ label. □

We now prove that of the total number of labels that may be assigned with a maximum length of size *maxLabelSize*, every second label is a *shorter label.*

Theorem 7.3. *Given an arbitrary nth label from the total number of labels that may be assigned with a maximum length of D bits, if n is divisible by two, then the length of the nth label is less than D bits. If n is not divisible by two, then the length of the nth label is always D bits.*

Proof: From SR$_b$7.2, we know the first label will always be of length D bits. From SR$_b$7.1, we know that the first label always terminates with a "1" bit. In order to obtain the second label, the first label cannot be incremented by extending its length because it is already at the maximum allowable length of D bits and it terminates with a "1" bit. Therefore, the first label is lexicographically incremented by incrementing the rightmost "0" bit. For example, if maximum label length D = 3, then the label 001 lexicographically increments to 01. When performing a lexicographical increment operation, all bits occurring after the position of the modified bit are discarded (thus, 001 increments to 01 and not to 011). Therefore, the second label always has a length less than D bits. Given any label *k* with a length equal to D bits and terminating with a "1" bit, the label *k+1* is obtained by lexicographically incrementing the rightmost "0" bit in label *k* (for example, label 10111 is lexicographically incremented to 11). The label *k+1* will always have a length less than D bits precisely because the incremented bit occurs before position D in label *k* and all bits after the incremented bit in label *k* are discarded. The lexicographical increment operation always generates the label *k+2* such that it has a length of D bits and terminates with a "1" bit because the increment operation on a label with less than D bits always generates a label with D bits and terminating with a "1" bit . Thus, the labels *k* and *k+2* have a length of D bits and terminate with a "1" bit. Therefore, given the nth label, if n is divisible by two, the length of the nth label is less than D bits. If n is not divisible by two, then the length of the nth label is always D bits. □

Assigning the Shorter labels

The number of nodes to be labeled (*nodeCount*) by the `AssignInitialLabels_Sequential` Algorithm may be less than the total number of labels available with a maximum length of *maxLabelSize*. By exploiting the lexicographical properties of theo-

Algorithm 7.2: SCOOBER `Increment`

```
input  : $N_{left}$ - a node label;
         maxLabelSize - maximum number of digits allowed in label.
output: $N_{new}$ - a new self_label such that $N_{left} \prec N_{new}$.
begin
    $N_{temp} \longleftarrow N_{left}$;
    if (length($N_{temp}$) == maxLabelSize) then
        while (last digit of $N_{temp}$ is '1') do
            $N_{temp} \longleftarrow N_{temp}$ with last digit removed;
        end
        $N_{new} \longleftarrow N_{temp}$ with last digit changed to '1';
    else if (length($N_{temp}$) < maxLabelSize) then
        while (i = Length($N_{temp}$) + 1; i < maxLabelSize; i++) do
            $N_{temp} \longleftarrow N_{temp} \oplus 0$;
        end
        $N_{new} \longleftarrow N_{temp} \oplus 1$;
    end
    return $N_{new}$;
end
```

rem 7.2 and given that every second label will have a length less than *maxLabelSize* from Theorem 7.3, we can guarantee to assign all shorter labels when generating *nodeCount* labels.

7.2.3 The `Increment` Algorithm

Algorithm 7.2 is the `Increment` Algorithm called by the `AssignInitialLabels_Sequential` Algorithm. The algorithm is very similar to the SCOOTER `Increment` Algorithm, but operates on binary-encoded labels and not on labels encoded to the base 3.

7.2.4 The `AssignInitialLabels_NodeK` Algorithm

The `AssignInitialLabels_NodeK` Algorithm is a label assignment algorithm that facilitates random access. Given a positive integer k and the total number of nodes to be labeled n, this algorithm determines the label of the k^{th} arbitrary node without the need to compute any other label. This algorithm is functionally identical to the SCOOTER `AssignInitialLabels_NodeK` Algorithm except it generates binary-encoded labels and not labels encoded to the base 3.

Algorithm 7.3: SCOOBER `AssignInitialLabel_NodeK`

```
/* Assign initial label to the K^th child node.                    */
input  : N - a positive integer representing the total number of nodes to be labeled.
         K - a positive integer representing the node whose label we want, such that $1 \leq K \leq N$.
output: A unique self_label (binary string) for the K^th child node.
begin
    D ⟵ Ceiling($\log_2$(N + 1));
    divisor ⟵ $2^D$;
    maxLabels ⟵ $2^D - 1$;
    numShorterLabels ⟵ maxLabels − N;
    if (K ≤ numShorterLabels) then
        divisor ⟵ divisor / 2;
        D ⟵ D − 1;
    else
        K ⟵ (numShorterLabels * 2) + (K − numShorterLabels);
    end
    quotient ⟵ K;
    self_label ⟵ ∅;
    for (j=1; j ≤ D; j++) do
        divisor ⟵ divisor / 2;
        code ⟵ Floor(quotient / divisor);
        self_label ⟵ self_label ⊕ code;
        remainder ⟵ quotient mod divisor;
        if (remainder == 0) then
            return self_label;
        else
            quotient ⟵ remainder;
        end
    end
end
```

7.2.5 Summary Analysis of `AssignInitialLabels` Algorithms

We now highlight three distinct characteristics of our `AssignInitialLabels` Algorithms.

1. The `AssignInitialLabels_Sequential` Algorithm is a *deterministic* algorithm (as defined in Definition 4.1) whereby each SCOOBER label can be determined entirely using the label of the node to the immediate left and immediate right. This property is exploited to enable the labeling scheme to generate compact labels at document initialization and to maintain compact labels under arbitrary node insertions and deletions.

2. One half of all labels available for assignment will have a length less than *maxLabelSize* (from Theorem 7.3). The `AssignInitialLabels_Sequential` Algorithm uses all of the shorter labels when initially assigning labels. Consequently, the `AssignInitialLabels_Sequential` Algorithm always assigns

the most compact SCOOBER labels.

3. The `AssignInitialLabels_NodeK` Algorithm can determine an arbitrary k^{th} child node label without having to compute any other child node label. The ability of the SCOOBER dynamic labeling scheme to compute node labels both sequentially and independently of one another, opens up the possibility of distributed and parallel processing in a multi-threaded and multi-core environment and may offer significant performance benefits.

7.3 Compact Adaptive Growth Method (Binary)

In §5.3, we presented the Compact Adaptive Growth Method which provided for the generation of labels with a highly constrained label size growth rate encoded to the base 3. However, our Method is independent of the numeric base 3. In this Chapter, we have introduced thus far, a binary-encoded bit-string dynamic labeling scheme called SCOOBER. We now briefly present the Compact Adaptive Growth Method using labels encoded in the base 2. The Compact Adaptive Growth Method provides a mechanism by which compact labels are generated. However, the Compact Adaptive Growth Method is tree-unaware. It generates bit-string labels that are compact, unique, *deterministic* and lexicographically ordered. It is the suite of node insertion algorithms (in conjunction with the `AssignInitialLabels` Algorithms that make up SCOOBER) that exploit the Compact Adaptive Growth Method to maintain a highly constrained label size growth rate irrespective of the quantity of arbitrary and repeated node label insertions and deletions. For clarity of exposition, we begin with a simple example providing an overview of the conceptual approach of the Compact Adaptive Growth Method.

Consider an XML tree consisting of a root node R and one child node with selflabel '1'. We insert a sequence of 100 nodes to the right of the rightmost child node. Table 7.2 illustrates the first 35 insertions. The first (2^0) node label generated consists of a *prefix* string '1' and a *postfix* string generated by the `AssignInitialLabels` Algorithm for a maxLabelSize of 1 digit (i.e.: '1'). For the next 2 (2^1) insertions, from the second to the third insertion inclusive, the newly generated labels consist

Insert after right-most node	SCOOBER label
	1
1	11
2	111
3	1111
4	11111
5	1111101
6	111111
7	1111111
8	11111111
9	11111111001
10	1111111101
11	11111111011
12	111111111
13	11111111101
14	1111111111
15	11111111111
16	111111111111
17	1111111111110001
18	111111111111001
19	1111111111110011
20	11111111111101
21	1111111111110101
22	111111111111011
23	1111111111110111
24	1111111111111
25	1111111111111001
26	111111111111101
27	1111111111111011
28	11111111111111
29	1111111111111101
30	111111111111111
31	1111111111111111
32	11111111111111111
33	1111111111111111100001
34	111111111111111110001
35	1111111111111111100011

Table 7.2

Col. 1	Col. 2	Col. 3	Col. 4	Col. 5	Col. 6
Growth counter	Node Start	Node End	Prefix length	Max Post-fix length	SCOOBER Selflabel total bits
1	1	1	1	1	2
2	2	3	2	2	4
3	4	7	4	3	7
4	8	15	7	4	11
5	16	31	11	5	16
6	32	63	16	6	22
7	64	127	22	7	29
8	128	255	29	8	37
9	256	511	37	9	46
10	512	1,023	46	10	56
11	1,024	2,047	56	11	67
12	2,048	4,095	67	12	79
13	4,096	8,191	79	13	92
14	8,192	16,383	92	14	106
15	16,384	32,767	106	15	121
16	32,768	65,535	121	16	137
17	65,536	131,071	137	17	154
18	131,072	262,143	154	18	172
19	262,144	524,287	172	19	191
20	524,288	1,048,575	191	20	211
21	1,048,576	2,097,151	211	21	232
22	2,097,152	4,194,303	232	22	254
23	4,194,304	8,388,607	254	23	277
24	8,388,608	16,777,215	277	24	301
25	16,777,216	33,554,431	301	25	326
26	33,554,432	67,108,863	326	26	352
27	67,108,864	134,217,727	352	27	379
28	134,217,728	268,435,455	379	28	407
29	268,435,456	536,870,911	407	29	436
30	536,870,912	1,073,741,823	436	30	466
31	1,073,741,824	2,147,483,647	466	31	497
32	2,147,483,648	4,294,967,295	497	32	529
⋮	⋮	⋮	⋮	⋮	⋮
n	2^{n-1}	$2^n - 1$	$\frac{n^2-n}{2}+1$	n	$\frac{n^2+n}{2}+1$

Table 7.3: Compact Adaptive Growth Rate (Binary)

of a *prefix* string '11' and a *postfix* string that mirrors the labels generated for a maxLabelSize of 2 bits starting at the midposition (e.g.: '1', '11') (please refer to Table 7.1 for examples of labels generated by the `AssignInitialLabels` Algorithm). The midposition is calculated using the same formula to determine the number of nodes available for insertion (2^1). For the next 4 (2^2) insertions, from the 4th to the 8th insertion inclusive, the labels consist of a *prefix* string '1111' and a *postfix* string that mirrors the labels generated for a maxLabelSize of 3 bits starting from the midposition (e.g.: '1', '101', '11', '111'). For the next 8 (2^3) insertions, from the

9th to the 16th insertion inclusive, the labels consist of a *prefix* string '1111111' (7 ones) and a *postfix* string that mirrors the labels generated for a maxLabelSize of 4 bits starting from the midposition (e.g.: '1', '1001', '101', '1011' and so on). This process is repeated as many times as is required.
The underlying properties of the Compact Adaptive Growth Method for binary-encoded labels are very similar to the properties for labels encoded in the base 3. In fact, the only difference is in the allowable values of the prefix. For clarity, we present the full compact adaptive binary rules (CAR_b) now.

$\mathbf{CAR}_b$ 7.1. *The smallest allowable prefix length is 1 bit.*

$\mathbf{CAR}_b$ 7.2. *When the prefix length is 1 bit, the maximum allowable postfix length is 1 bit.*

$\mathbf{CAR}_b$ 7.3. *The prefix string consists of one of two possible sequences: a sequence of one or more consecutive "0" bits; or a sequence of one or more consecutive "1" bits. The particular sequence chosen as prefix depends on the insertion operation to be performed.*

$\mathbf{CAR}_b$ 7.4. *The length of the postfix string can be less than or equal to the maximum allowable postfix length.*

$\mathbf{CAR}_b$ 7.5. *The maximum label length is always equal to the sum of the prefix and the maximum allowable postfix length.*

$\mathbf{CAR}_b$ 7.6. *When generating a new rightmost label, the length of the prefix is extended if and only if the current rightmost label consists of all '1' bits and the length of the current rightmost node label equals the sum of the current prefix length and maximum allowable postfix length.*

$\mathbf{CAR}_b$ 7.7. *The new prefix length is assigned the value of the previous maximum allowable label length; the new maximum postfix length becomes the previous maximum postfix length plus 1. This rule is codified in Algorithm 5.4 and illustrated in columns 4, 5 and 6 of Table 7.3.*

CAR$_b$ 7.8. *When CAR$_b$7.7 is invoked and an adaptive increase in the prefix and postfix lengths has been performed, the first postfix will always be assigned the label at the lexicographical midposition of all possible labels of length maximumAllowable-PostfixLength.*

Using the same comparison that was highlighted in §5.3, all binary-encoded bit-string dynamic labeling schemes have a best case label size growth rate of one bit per node insertion. Thus, after one thousand insertions, one million insertions and one billion insertions, the largest self_label sizes are 1,000 bits, 1,000,000 bits and 1,000,000,000 bits respectively. For the same insertions using SCOOBER, the largest self_labels are 56 bits, 211 bits and 466 bits respectively (please refer to columns 2, 3 and 6 of Table 7.3). Thus, SCOOBER labels are several orders of magnitude smaller than the labels of all existing bit-string dynamic labeling schemes when processing frequently skewed insertions. Furthermore, SCOOBER generates compact labels without requiring advance knowledge of the number of nodes to be inserted.

7.4 Compact Adaptive Growth (Binary) Evaluation

We have performed a comprehensive theoretical evaluation of the Compact Adaptive Growth Method for binary-encoded labels. However, the evaluation is almost identical to the evaluation provided for labels encoded to the base 3 in §5.4. Thus, to avoid a high degree of repetition, we have included the comprehensive theoretical evaluation of the Compact Adaptive Growth Method for binary-encoded labels in Appendix C.

7.5 SCOOBER Node Label Insertion Algorithms

We have provided a suite of SCOOBER node label insertion algorithms that exploit the Compact Adaptive Growth Method for binary-encoded bit-string labels to support arbitrary node insertions and deletions while maintaining a highly constrained label size growth rate. All of the algorithms support the reuse of shorter deleted node labels when available and process frequently skewed insertions gracefully. However, the SCOOBER algorithms are almost identical to the SCOOTER algorithms presented in Chapter 6, and provide the same functionality. Therefore, to avoid a high degree of repetition, we have included the full suite of SCOOBER node insertions algorithms in Appendix D.

7.6 Label Storage Scheme for Binary-Encoded Bit-String Labels

In §2.6, we reviewed all four classifications of label storage schemes proposed to date: length fields, control tokens, separators and prefix-free codes. A label storage scheme specifies how a label is encoded in binary so as to be stored on disk or any other physical digital medium. Control tokens and prefix-free label storage schemes have been deployed by dynamic labeling schemes that numerically or alphanumerically encoded their labels. In contrast, length field and separator label storage schemes have been deployed by bit-string dynamic labeling schemes. Figure 3.1 showed the type of label storage schemes employed by all dynamic labeling schemes reviewed in our literature review.

One of the key problems we sought to address in this dissertation, is the provision of scalability - that is a dynamic labeling scheme will never require the relabeling of existing nodes under any arbitrary combination of repeated node insertions and deletions. SCOOTER (and the QED labeling scheme) successfully provide this feature by employing a quaternary encoding of their labels in conjunction with the separator label storage scheme. However, SCOOBER is a binary-encoded bit-string dynamic labeling scheme and is therefore, unable to avail of the separator storage scheme. Given that all existing control tokens and prefix-free label storage schemes encode numeric and alphanumeric labels and not bit-string labels, the only classification of label storage scheme that may be employed by bit-string dynamic labeling schemes is the length-field label storage scheme. Recall that the concept underlying length fields is to store the length of the label immediately before the label itself. However, all fixed and variable length label storage schemes are subject to relabeling, or require a very large fixed-length bit code to indicate the label size which results in a significant wastage of storage space. Consequently, there does not exist a label storage scheme for binary-encoded bit-string dynamic labeling schemes that completely avoids the relabeling of nodes. We address this problem now.

7.6.1 Fibonacci Label Storage Scheme

We now introduce the Fibonacci label storage scheme which may be employed by any binary-encoded bit-string dynamic labeling scheme and facilitates a labeling scheme to completely avoid the relabeling of nodes. The Fibonacci label storage scheme is a hybrid of the control token and length field classifications. Before we describe the label storage scheme, we provide a brief overview of the Fibonacci sequence [56] and the Zeckendorf representation [57].

Definition 7.2. *Fibonacci Sequence.*
The Fibonacci sequence is given by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ *with* $F_0 = 0$ *and* $F_1 = 1$ *such that* $n \geq 2$.

The first 10 terms of the Fibonacci sequence are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34. Each term in the sequence is the sum of the previous two terms.

Definition 7.3. *Zeckendorf Representation.*
Every positive integer n has a unique representation as the sum of one or more distinct non-consecutive Fibonacci numbers. This may be more formally stated as:
$n = \sum_{k=0}^{N} \epsilon_k F_k$ *where* ϵ_k *are 0 or 1, and* $\epsilon_k * \epsilon_{k+1} = 0$.

It should be noted that although there are several ways to represent a positive integer n as the sum of Fibonacci numbers, only one representation is the Zeckendorf representation of n. For example, the positive integer 111 may be represented as the sum of Fibonacci numbers in the following way:

1. 111 = 89 + 21 + 1

2. 111 = 55 + 34 + 13 + 8 + 1

3. 111 = 89 + 13 + 5 + 3 + 1

However, only the first expression is the Zeckendorf representation of 111, because the second expression contains two consecutive Fibonacci terms (55 + 34) as does the third expression (5 + 3). We will exploit the property that no two consecutive Fibonacci terms occur in the Zeckendorf representation of a positive integer to construct the Fibonacci label storage scheme.

7.6.2 Encoding and Decoding the Length of the Label

We begin with a simple example providing an overview of the encoding process for the Fibonacci label storage scheme before we present our algorithms. Given a binary-encoded bit-string label $N_{new} = 110101$,

- We first determine the length of N_{new}. It has 6 bits.
- We then obtain the Zeckendorf representation of the length of the label. The Zeckendorf representation of 6 is 5 + 1.
- We then encode the Zeckendorf representation of the label length as a Fibonacci coded binary string. Specifically, starting from the Fibonacci term F_2 (recall $F_0 = 0$ and $F_1 = 1$), if the Fibonacci term F_{k+1} occurs in the Zeckendorf representation of the label length, then the k^{th} bit in the Fibonacci coded binary string is set to "1". If the Fibonacci term F_{k+1} does *not* occur in the Zeckendorf representation, then the k^{th} bit in the Fibonacci coded binary string is set to "0".
- For example, the first term F_2 (1) occurs in the Zeckendorf representation of 6 and thus, the first bit in the binary string is "1". The second term F_3 (2) does not occur in the Zeckendorf representation of 6 and thus, the binary string is now "10". The third term F_4 (3) does not occur in the Zeckendorf representation of 6 and thus, the binary string is now "100". The fourth term F_5 (5) occurs in the Zeckendorf representation of 6 and thus, the binary string is now "1001". There are no more terms in the Zeckendorf representation of the length of N_{new} (6 bits), therefore stop.

- The Fibonacci coded binary string of the Zeckendorf representation will never contain two consecutive “1” bits, precisely because it is a Fibonacci encoding of an Zeckendorf representation. Also, given that the construction of the Fibonacci coded binary string stops after processing the last term in the Zeckendorf representation, we are certain the last bit in the Fibonacci coded binary string must be “1”. We subsequently append an extra “1” bit to the end of the Fibonacci coded binary string to act as a control token or delimiter. Thereafter, we know the only place two consecutive “1” bits can occur in the Fibonacci coded binary string is at the end of the string. Thus the binary string is now “10011”.

- The Fibonacci label storage scheme adopts a length field storage approach, which means we encode and store the size of the label immediately before the label itself. The last two bits of the Fibonacci coded binary string will always consist of two consecutive “1” bits and they act as a control token separating the length field of the label from the label itself. To complete our example, the label N_{new} (110101) is encoded and stored using the Fibonacci label storage scheme as 10011 110101 (the space is provided as a visual aid).

It can be seen from above that the Fibonacci label storage scheme is a hybrid of the control token and length field label storage schemes. In [5] and [19], the authors exploit a Fibonacci coding of the Zeckendorf representation of variable-length binary strings for synchronization and error correction during the transmission of codes. However, to the best of our knowledge, Fibonacci coded binary strings have never been proposed as a foundation for a label storage scheme nor have they been proposed to provide scalabilty to dynamic labeling schemes to completely avoid the relabeling of nodes in the presence of repeated and arbitrary node insertions and deletions.
Algorithm 7.4 outlines the label length encoding process. It receives as input a positive integer n representing the label length and outputs a Fibonacci coded binary string of the Zeckendorf representation of n. Algorithm 7.5 outlines the label length decoding process which is the reverse transformation of Algorithm 7.4

Algorithm 7.4: `EncodeLabelLength`

```
/* Encode n to Fibonacci coded binary string of Zeckendorf representation of n.      */
input  : n - a positive integer representing the length of a label.
output: fibStr - a Fibonacci coded binary string of the Zeckendorf representation of n.
begin
    $F_0 \longleftarrow 0$;
    $F_1 \longleftarrow 1$;
    $F_{start} \longleftarrow F_0 + F_1$;
    $F_{end} \longleftarrow$ the largest Fibonacci number $\leq$ n;
    fibArray $\longleftarrow$ the Fibonacci sequence from $F_{start}$ to $F_{end}$ inclusive;
    fibStr $\longleftarrow$ '1';
    for (i=length(fibArray); i=1; i−−) do
        if (n $\geq$ fibArray[i]) then
            fibStr $\longleftarrow$ '1' $\oplus$ fibStr;
            n $\longleftarrow$ n $-$ fibArray[i];
        else
            fibStr $\longleftarrow$ '0' $\oplus$ fibStr;
        end
    end
    return fibStr;
end
```

Algorithm 7.5: `DecodeLabelLength`

```
/* Decode a Fibonacci coded binary string of a Zeckendorf representation to n.       */
input  : fibStr - a Fibonacci coded binary string of the Zeckendorf representation of n.
output: n - a positive integer representing the length of a label.
begin
    $F_0 \longleftarrow 0$;
    $F_1 \longleftarrow 1$;
    $F_{start} \longleftarrow F_0 + F_1$;
    fibCount $\longleftarrow$ length(fibStr);
    fibArray $\longleftarrow$ the first fibCount terms of the Fibonacci sequence from $F_{start}$ inclusive;
    n $\longleftarrow$ 0;
    for (i=1; i < length(fibArray); i++) do
        if (fibStr[i] == '1') then
            n $\longleftarrow$ n + fibArray[i];
        end
    end
    return n;
end
```

7.6.3 Fibonacci Label Storage Scheme Size Analysis

In Table 7.4, we illustrate the relationship between the growth rate of the Fibonacci coded binary string (of the Zeckendorf representation) of the length of the label and the corresponding growth in the quantity of labels that may be encoded with that length. Given a label encoding length n, the quantity of labels that may be encoded with length n is equal to the Fibonacci term F_{n-1}. In [50], the authors prove that the average value of the n^{th} term of a sequence defined by the general recurrence relation $G_n = G_{n-1} +- G_{n-2}$ increases exponentially. *Therefore, as the number of labels to be encoded using the Fibonacci label storage scheme increases exponentially,*

Growth Counter	Label Encoding Length	Num of labels
1	2	1
2	3	1
3	4	2
4	5	3
5	6	5
6	7	8
7	8	13
8	9	21
9	10	34
10	11	55
11	12	89
12	13	144
13	14	233
14	15	377
⋮	⋮	⋮
n	n + 1	F_n

Table 7.4: Fibonacci Coded Label Length Growth Rate

the corresponding growth in the size of the Fibonacci coded binary string is linear. This demonstrates that when processing a large quantity of labels the Fibonacci label storage scheme scales gracefully.

7.7 Summary

In this Chapter, we presented the Compact Adaptive Growth Method for the generation of binary-encoded bit-string labels. We also presented a binary-encoded bit-string dynamic labeling scheme, namely SCOOBER. The labeling scheme exploits the CAGM to facilitate an arbitrary number of node label insertions and deletions while maintaining a highly constrained growth rate in label size. This labeling scheme is very similar to SCOOTER and shares all of the properties and advantages of SCOOTER save one, scalability. To that end, we presented the Fibonacci label storage scheme which may be used in conjunction with SCOOBER to completely avoid the relabeling of nodes under any node insertion or deletion scenario. Furthermore, the Fibonacci label storage scheme provides a highly compact representation of node label sizes when processing large node labels. In the next Chapter, we provide a comprehensive experimental evaluation and analysis of the various contributions made in this dissertation.

Chapter 8

Experiments

In this Chapter, we provide a comprehensive evaluation of SCOOTER and SCOOBER dynamic labeling schemes for XML and the Fibonacci label storage scheme. The primary goal of this dissertation is the specification of a dynamic labeling scheme that provides a high degree of functionality while maintaining compact labels at all times. To that end, we focus on both the storage costs and the computational processing costs of maintaining compact labels in the provision of tree-based update functionality.

This Chapter is structured as follows: In §8.1, we provide an evaluation of the Fibonacci label storage scheme; in §8.2 through §8.4, we evaluate the dynamic labeling schemes at each stage of the document lifecycle; in §8.2, we evaluate the assignment of labels when a document is initially labeled; in §8.3, we focus on the generation of node labels under various node insertion scenarios, including frequently skewed node insertions; and in §8.4, we consider the impact of deleted node label reuse when performing node insertions in a dynamic environment.

Experimental Method and Setup

In this section, we outline our evaluation method and describe our experimental setup.

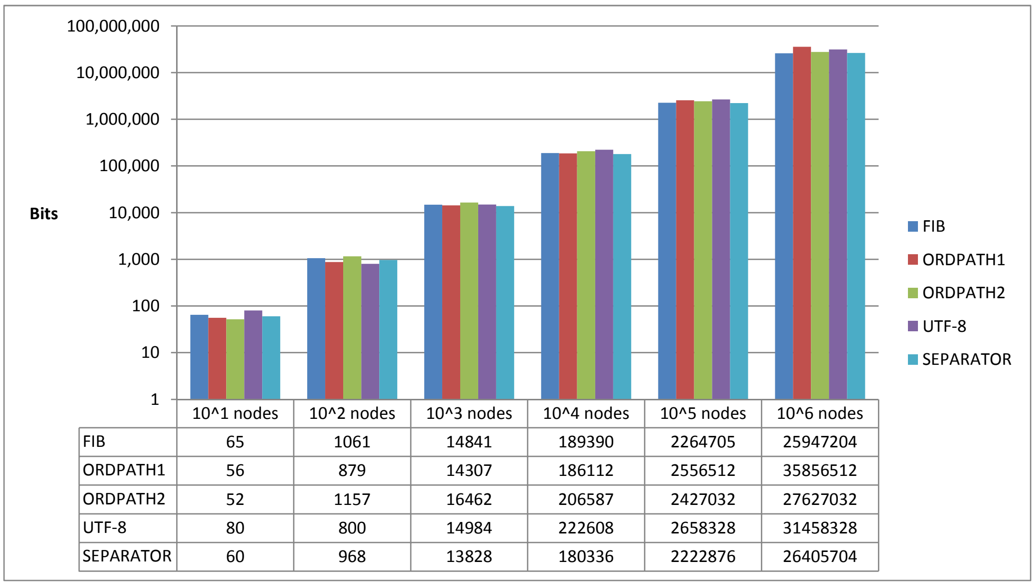
We evaluate and compare our dynamic labeling schemes with four other dynamic labeling schemes, namely QED (also known as CDQS), CDBS, ORDPATH and the

Vector labeling scheme. These four labeling schemes were chosen because collectively, they offer some of the properties that are present in our labeling schemes - scalability and compact labels. QED is the only bit-string dynamic labeling scheme that offers a compact label encoding at document initialization, while overcoming the overflow problem and completely avoiding node relabeling. However, QED does not offer compact labels under frequently skewed node insertions. The Vector labeling scheme is the only dynamic labeling scheme that has as one of its design goals, the ability to process frequently skewed insertions efficiently. The CDBS labeling scheme provides an encoding at document initialization that is as compact as the binary encoding of integers. Although CDBS is subject to the overflow problem and will require the relabeling of all nodes under certain update scenarios, we wanted to compare our labeling schemes with the most compact binary-encoded bit-string dynamic labeling scheme available. Lastly, ORDPATH is an industrial strength labeling scheme currently deployed in Microsoft SQL Server. All of the labeling schemes were implemented in Java version 6.38 and all experiments were carried out on a 2.66Ghz Intel(R) Core(TM)2 DUO CPU and 4GB of RAM. The experiments were performed 11 times, the time from the first run was discarded and the results of the subsequent 10 experiments averaged. For all experiments, the unit of storage is in *bits* and the unit of time is in milliseconds (ms). The ORDPATH prefix-free code tables and the array of Fibonacci numbers from 1 to N are computed once (in advance) and not each time a label is encoded/decoded, so as to reflect a real-world implementation scenario.

8.1 Label Storage Scheme Evaluation

In this section, we evaluate the Fibonacci label storage scheme by comparing it with three other label storage schemes, namely ORDPATH Compressed binary format, UTF-8 and the Separator label storage schemes. The three label storage schemes were reviewed in §2.6.

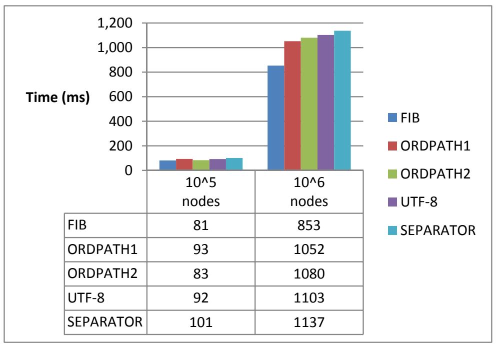
The Fibonacci and Separator label storage schemes were designed to encode bit-string labels, whereas the ORDPATH and UTF-8 label storage schemes were designed to encode integer-based labels. Consequently, to ensure an equitable and fair experimental evaluation, all four label storage schemes encode the positive integers from 1 to 10^n where n has the values from 1 to 6 inclusive. Given that the Fibonacci and Separator label storage schemes expect a bit-string label to encode, the integer is converted from base 10 to base 2 (binary) and the binary string representation of the integer is encoded. In Figure 8.1, we illustrate the storage costs for all four label storage schemes using labels derived from the integer encodings from 1 to 10^6. The ORDPATH compressed binary format provides a choice of two encoding tables to use; we present both encodings to enable a comprehensive evaluation and analysis. In this remainder of this Chapter, "FIB" is used to denote the Fibonacci label storage scheme.



	10^1 nodes	10^2 nodes	10^3 nodes	10^4 nodes	10^5 nodes	10^6 nodes
FIB	65	1061	14841	189390	2264705	25947204
ORDPATH1	56	879	14307	186112	2556512	35856512
ORDPATH2	52	1157	16462	206587	2427032	27627032
UTF-8	80	800	14984	222608	2658328	31458328
SEPARATOR	60	968	13828	180336	2222876	26405704

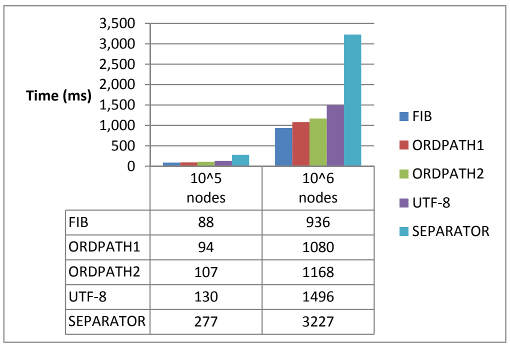
Figure 8.1: Storage Costs of Encoding Integers for the Label Storage Schemes

ORDPATH2 provides the most compact storage representation when encoding less than 10 integers, followed closely by ORDPATH1 in second place and SEPARATOR in third place. UTF-8 provides the most compact storage representation when encoding 10^2 integers and SEPARATOR when encoding 10^3 through 10^5 inclusive. As the size of the label grows, the size of the delimiter employed by SEPARATOR remains constant (2 bits). When encoding 10^6 integers, FIB provides the most compact storage representation. This result is in line with our theoretical evaluation of FIB in §7.6.3 which observed that as the number of labels to be encoded using FIB increases exponentially, the corresponding growth rate in the size of the Fibonacci coded binary string is linear. Hence, although the storage costs of FIB is average for small to medium sized labels, FIB provides a highly compact storage representation for large labels. However, unlike ORDPATH and UTF-8, FIB is not subject to the overflow problem and will never require existing labels to be relabeled.



	10^5 nodes	10^6 nodes
FIB	81	853
ORDPATH1	93	1052
ORDPATH2	83	1080
UTF-8	92	1103
SEPARATOR	101	1137

(a) Label Storage Scheme Encoding Times



	10^5 nodes	10^6 nodes
FIB	88	936
ORDPATH1	94	1080
ORDPATH2	107	1168
UTF-8	130	1496
SEPARATOR	277	3227

(b) Label Storage Scheme Decoding Times

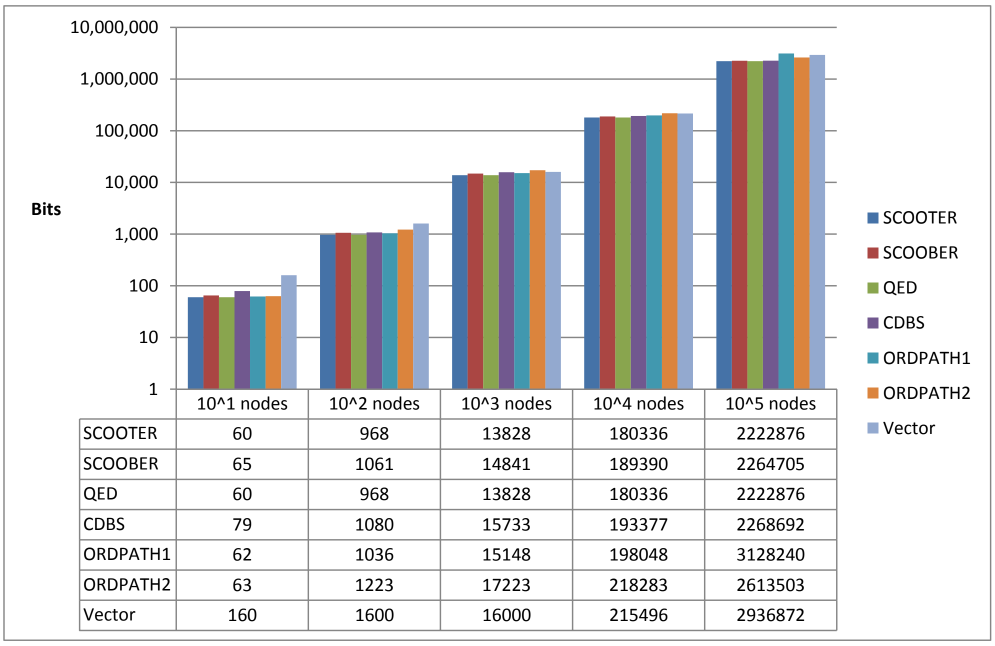
Figure 8.2: Integer Encoding and Decoding Times for Label Storage Schemes

In Figure 8.2, we illustrate the computation processing times to encode and decode 10^5 and 10^6 integer labels. The times for 10^4 (or less) integer encodings are not shown because they are single digit results with negligible differences between them. FIB is the fastest label storage scheme at both encoding and decoding. FIB is the fastest because as a length field label storage scheme it only has to encode and decode the length of the label. The actual bit-string label is stored immediately after the Fibonacci coded binary string and can be read and written without having to process each individual bit. All of the other label storage schemes must process the entire label to generate their encoding. ORDPATH1 has similar encode and decode computational processing costs. ORDPATH2 encodes more quickly than it decodes because the key size that maps to the range of the integer encoding grows more quickly than the encoding table employed by ORDPATH1. UTF-8 decodes approximately 30 percent slower than it encodes because when decoding, it must parse each individual byte in the multi-byte label and strip away the 2-bit control token at the start of each byte. SEPARATOR is the slowest at both encoding and

decoding because it must parse every bit in the bit-string label in order to locate the predefined bit sequence representing the separator and thus, cannot perform byte-size processing.
In summary, FIB is the fastest label storage scheme at both encoding and decoding, it is not subject to the overflow problem and will never required the relabeling of existing node labels.

8.2 AssignInitialLabels Evaluation

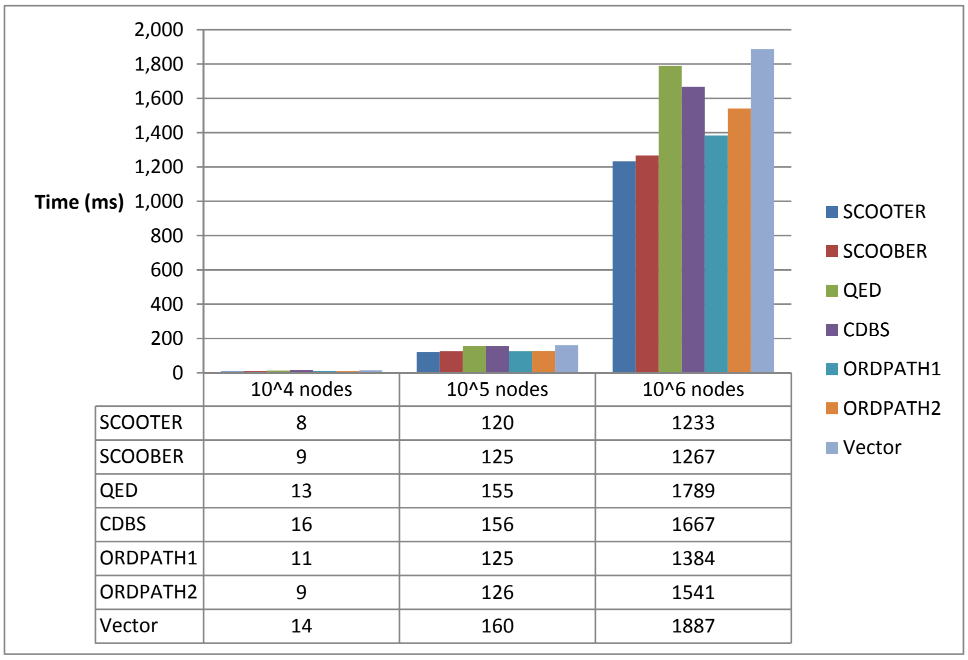
In this section, we evaluate the SCOOTER and SCOOBER dynamic labeling schemes when processing the first stage of the document lifecycle - that is the generation of and assignment of node labels when an XML document tree is initially labeled.



	10^1 nodes	10^2 nodes	10^3 nodes	10^4 nodes	10^5 nodes
SCOOTER	60	968	13828	180336	2222876
SCOOBER	65	1061	14841	189390	2264705
QED	60	968	13828	180336	2222876
CDBS	79	1080	15733	193377	2268692
ORDPATH1	62	1036	15148	198048	3128240
ORDPATH2	63	1223	17223	218283	2613503
Vector	160	1600	16000	215496	2936872

Figure 8.3: Algorithms `AssignInitialLabels` Label Storage Costs

In Figure 8.3, we illustrate the total node label storage costs of 10^1 through 10^5 initially assigned node labels for each labeling scheme. It is clear from the results that there is a direct relationship between the labels generated by the SCOOTER and QED `AssignInitialLabels` Algorithms - they have identical storage costs. This may appear odd given that the `AssignInitialLabels` Algorithms of both labeling schemes generate different labels. However, recall that both algorithms guarantee to assign the most compact labels for any given n nodes to be labeled, and provide independently developed and materially different processes to achieve this goal. It should also be noted that SCOOBER and CDBS also generate and assign the most compact binary-encoded bit-string labels and consequently have the same storage costs at a labeling scheme level. However, SCOOBER labels employ the Fibonacci label storage scheme whereas CDBS labels are encoded using ORDPATH1 compressed binary format as recommended by the authors of CDBS in [38]. Consequently their label storage costs illustrated in Figure 8.3 are different. The SCOOTER and QED labeling scheme generate the most compact labels of all the labeling schemes evaluated. The SCOOTER and QED labeling scheme employ the Separator label storage scheme and thus, as the label size grows, the size of the separator remains constant - just 2 bits. No other label storage scheme offers this property and consequently, as the label size grows, both the label and the label encodings grow in size. From 10^3 upwards, the SCOOBER labeling scheme offers the second most compact initially assigned labels.



	10^4 nodes	10^5 nodes	10^6 nodes
SCOOTER	8	120	1233
SCOOBER	9	125	1267
QED	13	155	1789
CDBS	16	156	1667
ORDPATH1	11	125	1384
ORDPATH2	9	126	1541
Vector	14	160	1887

Figure 8.4: Time Taken (Milliseconds) by the `AssignInitialLabels` Algorithms

In Figure 8.4, we illustrate the computational runtime costs of generating the labels where values represent the time taken in milliseconds. The Vector labeling scheme presents the slowest times followed by QED and CDBS. The SCOOTER labeling scheme consistently offers the quickest processing times regardless of the number of labels to be generated, followed closely in second place by SCOOBER. The most significant factor affecting the processing times of the QED, CDBS and Vector labeling schemes is the recursive algorithm employed by their `AssignInitialLabels` Algorithms and subsequently requires the construction in memory of large encoding tables to keep track of all previously assigned labels. In contrast, the SCOOTER and SCOOBER labeling schemes employ deterministic algorithms that assigns labels in a sequential manner and have a minimal memory overhead.

8.3 Node Insertion Evaluation

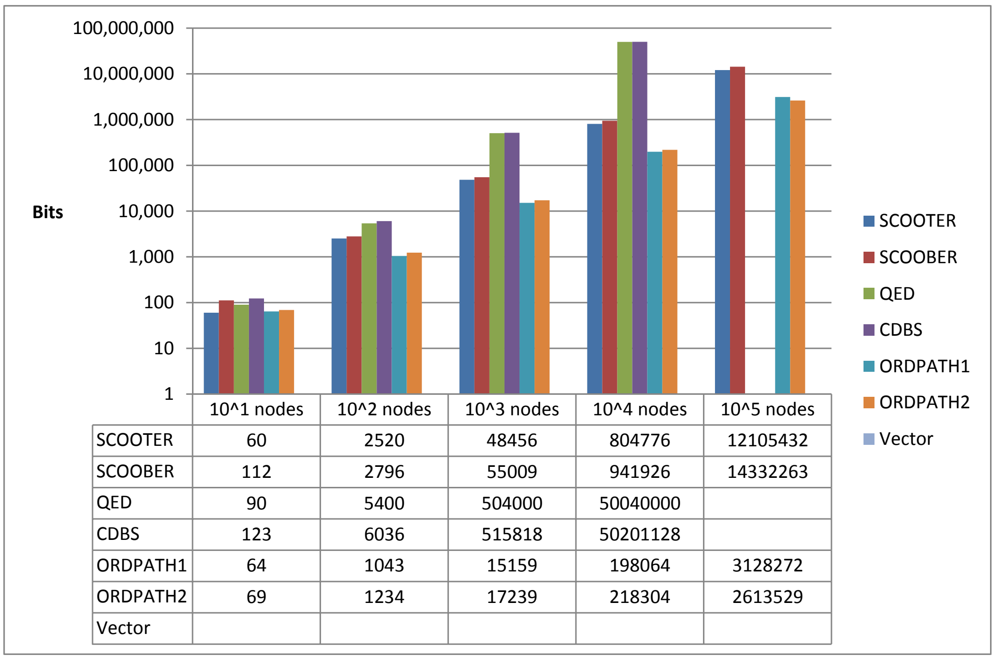
In this section, we provide an evaluation of the various node label insertion scenarios. This includes both node insertions after a rightmost node and frequently skewed node insertions.

8.3.1 Insertion After Rightmost Node Evaluation

We begin by presenting an analysis of the simplest node insertion scenario - an insertion of a new node after the current rightmost node. This is an ideal update scenario to demonstrate the growth characteristics of an update operation provided by a dynamic labeling scheme because it does not require a label comparison operation. Therefore, the update operation is not influenced by the particular values of two arbitrary labels.

Figure 8.5 presents the label storage costs for the `InsertNewNodeAfterRightmostNode` Algorithms under various update loads. For less than 10 node insertions, SCOOTER offers the most compact labels. However, for 10^2 node labels and greater, the ORDPATH labeling scheme offers the most compact labels because the labels are made up of integer components and the integer component value increments by 2 for each node insertion. Hence, the encoding of the label grows very slowly.

The results for the Vector labeling scheme are not available because, although the

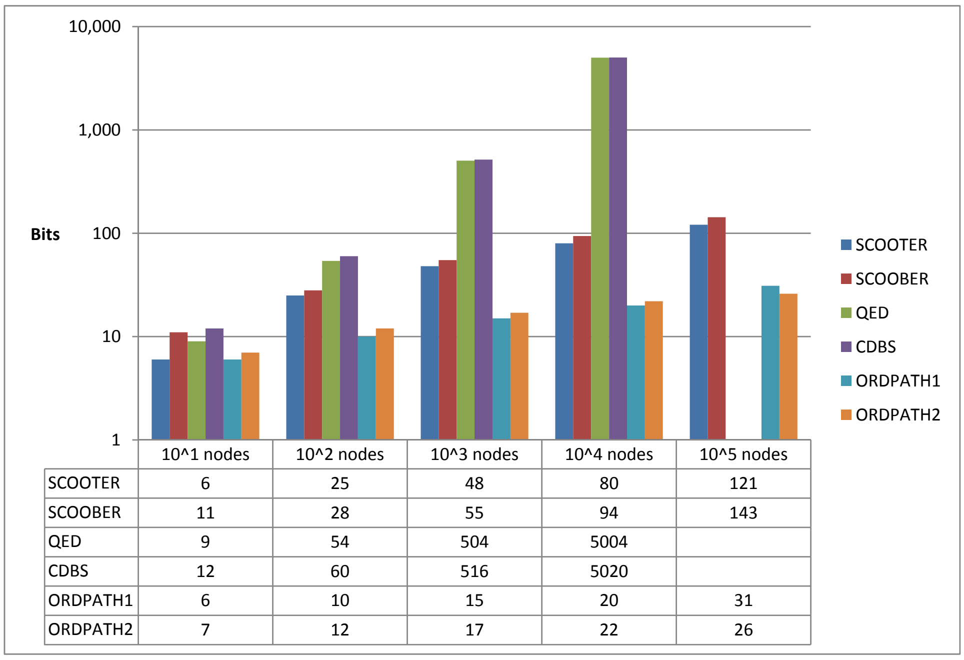


	10^1 nodes	10^2 nodes	10^3 nodes	10^4 nodes	10^5 nodes
SCOOTER	60	2520	48456	804776	12105432
SCOOBER	112	2796	55009	941926	14332263
QED	90	5400	504000	50040000	
CDBS	123	6036	515818	50201128	
ORDPATH1	64	1043	15159	198064	3128272
ORDPATH2	69	1234	17239	218304	2613529
Vector					

Figure 8.5: Algorithm `InsertNewNodeAfterRightmostNode` Label Storage Costs

authors claim the Vector labeling scheme may be deployed as a prefix labeling scheme, it is unclear how to insert a new node after the rightmost node label. Document order is maintained among vector node labels based on the numerical order of their gradients [67]. However, when initially assigning one or more labels, the rightmost vector node label is always (0, 1) which has a gradient of infinity. It is unclear how to insert a new label after the rightmost vector node label (0, 1) such that it has a gradient numerically ordered greater than infinity. Thus, we omit the Vector labeling scheme from further experimental evaluation because it is unsuitable for use as a prefix dynamic labeling scheme for XML.

The remaining four dynamic label schemes (SCOOTER, SCOOBER, QED and CDBS) are all bit-string dynamic labeling schemes. QED and CDBS have a one-bit per node insertion growth rate in label size and hence, their labels grow rapidly. The results when inserting 10^5 node labels using QED and CDBS are omitted because the Java virtual machine consumed all of the available memory and we were unable to generate the labels. In contrast, the SCOOTER and SCOOBER dynamic labeling scheme have a highly constrained label size growth rate as dictated by the Compact Adaptive Growth Method. The storage requirements of SCOOTER and SCOOBER for 10^4 node insertions is less than two percent of that required by CDBS and QED. Hence, SCOOTER and SCOOBER offer significant savings in storage costs compared to the best bit-string dynamic label schemes available to date.



	10^1 nodes	10^2 nodes	10^3 nodes	10^4 nodes	10^5 nodes
SCOOTER	6	25	48	80	121
SCOOBER	11	28	55	94	143
QED	9	54	504	5004	
CDBS	12	60	516	5020	
ORDPATH1	6	10	15	20	31
ORDPATH2	7	12	17	22	26

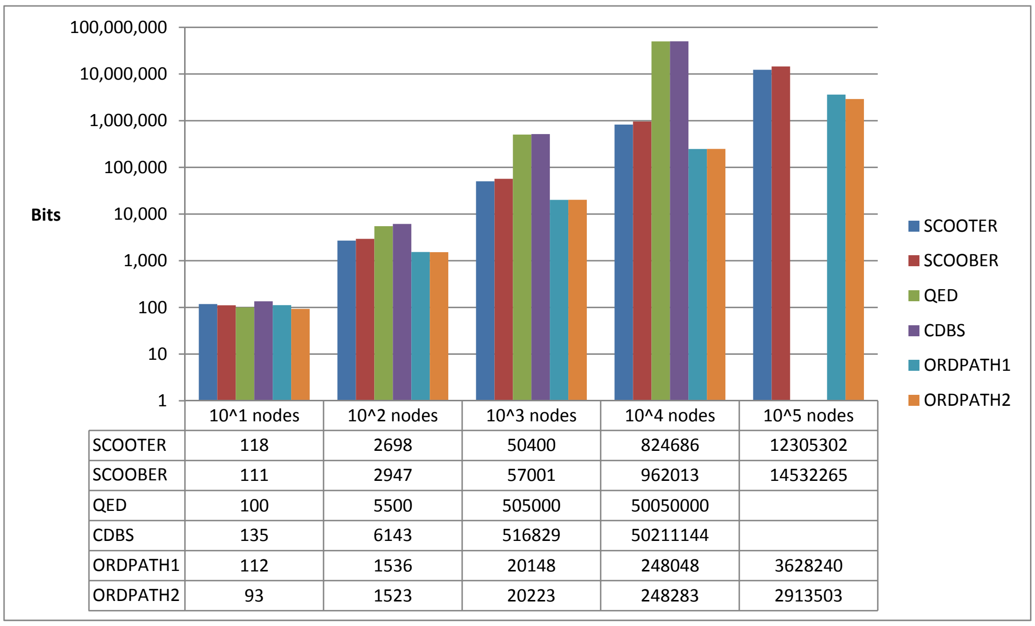
Figure 8.6: Algorithm `InsertNewNodeAfterRightmostNode` Average Label Sizes

One of our design goals was to maintain a compact label size at all times. Hence, a good metric by which we can analyze and measure if we have been successful in obtaining this objective is to identify the average node label size after a sequence of node insertions. Figure 8.6 illustrates the average node label size for the node insertions illustrated in Figure 8.5. The results confirm that SCOOBER and SCOOTER maintain average label sizes that are more than an order of magnitude smaller than QED or CDBS. The results also confirm that SCOOTER outperforms SCOOBER.

8.3.2 Frequently Skewed Insertions Evaluation

We now evaluate the performance of the labeling schemes under frequently skewed node insertions. There are two types of frequently skewed node insertions possible: Bulk Insertions and Fixed Point insertions. We describe and analyze both now.

Bulk Insertions



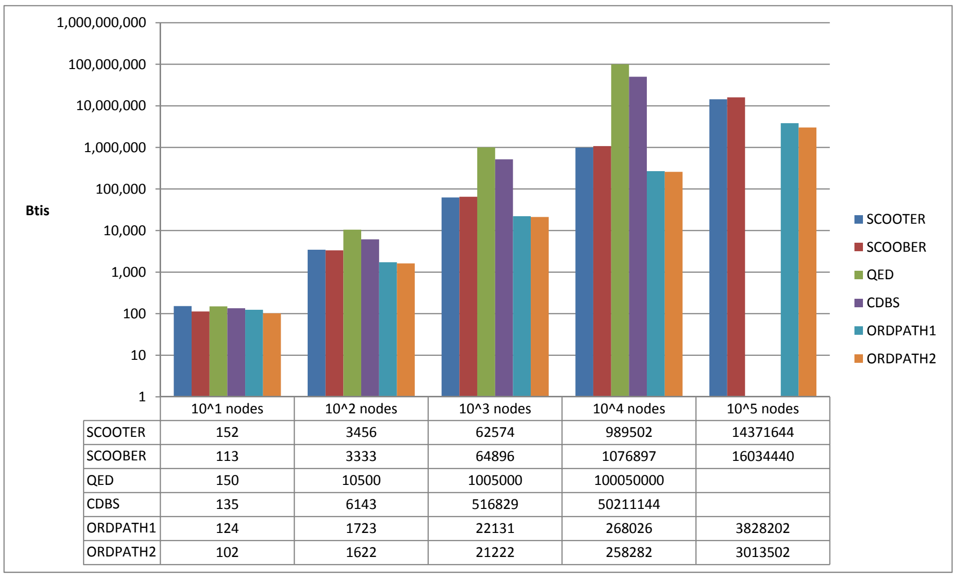
	10^1 nodes	10^2 nodes	10^3 nodes	10^4 nodes	10^5 nodes
SCOOTER	118	2698	50400	824686	12305302
SCOOBER	111	2947	57001	962013	14532265
QED	100	5500	505000	50050000	
CDBS	135	6143	516829	50211144	
ORDPATH1	112	1536	20148	248048	3628240
ORDPATH2	93	1523	20223	248283	2913503

Figure 8.7: Bulk Node Insertions Label Storage Costs

Bulk Insertions are a continuous sequence of two or more node insertions between two consecutive sibling nodes such that each new node is inserted immediately after the previously inserted node. In Figure 8.7, we illustrate the total label storage costs for 10^1 through 10^5 bulk node insertions. ORDPATH offers the most compact label size. However, although a bulk insertion is a similar update operation to the insertion after rightmost node, the ORDPATH labels are approximately 20 percent larger under bulk insertions. This is a result of the *careting-in* technique employed by the ORDPATH labeling scheme when inserting a new node between two consecutive sibling node labels. All newly inserted labels contain an additional integer component which contributes to the 20 percent increase in total label size. The SCOOTER, SCOOBER, QED and CDBS dynamic labeling schemes have results broadly comparable with the results from an insertion after rightmost node.

Fixed Point Insertions

Fixed Point insertions describe a continuous sequence of two or more node insertions between two consecutive sibling nodes N_{left} and N_{right} such that each new node is



	10^1 nodes	10^2 nodes	10^3 nodes	10^4 nodes	10^5 nodes
SCOOTER	152	3456	62574	989502	14371644
SCOOBER	113	3333	64896	1076897	16034440
QED	150	10500	1005000	100050000	
CDBS	135	6143	516829	50211144	
ORDPATH1	124	1723	22131	268026	3828202
ORDPATH2	102	1622	21222	258282	3013502

Figure 8.8: Fixed Point Insertions Label Storage Costs

inserted at the fixed point immediately after N_{left}. Figure 8.8 illustrates the label storage costs for labels generated after fixed point node insertions with an update load ranging from 10^1 through 10^5 nodes. The most significant observation is the speed at which the QED label sizes grow. The QED labeling scheme generates labels that grow at twice the speed compared with the corresponding growth rate resulting from a bulk node insertion (as illustrated in Figure 8.7). QED must add two new bits to each label generated by a fixed point node insertion compared to an average of one bit added to each label generated by a bulk node insertion. After just 10^4 fixed point node insertions, the label storage cost using QED labels is over 100 million bits. The label storage costs under the same update scenario for the SCOOTER labeling scheme is under 1 million bits.

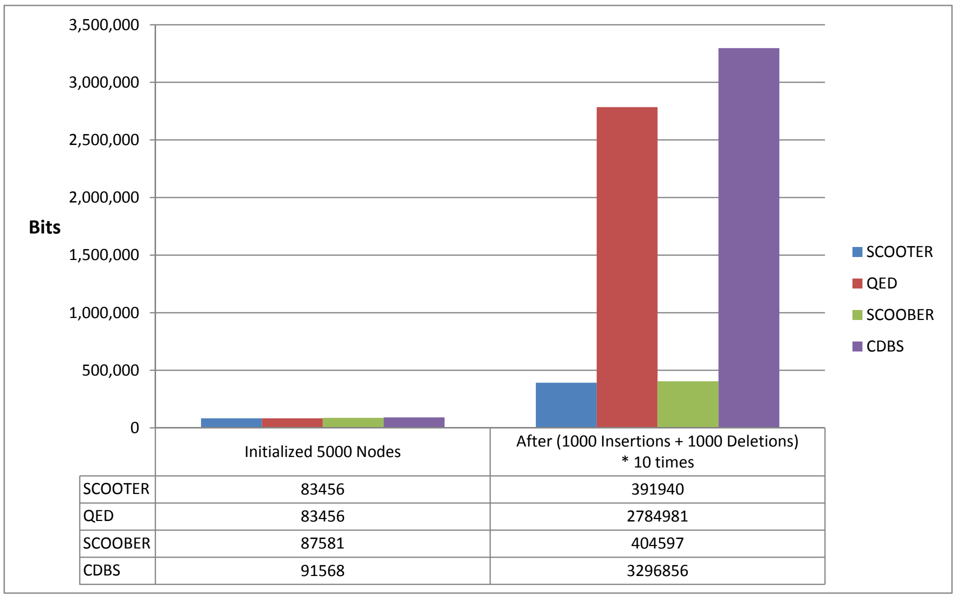
8.4 Node Insertion with Reuse Evaluation

In our final experiment, we focus on the ability of the SCOOTER and SCOOBER dynamic labeling schemes to support the reuse of deleted node labels when inserting new labels in a dynamic environment. Recall that no dynamic labeling scheme to date offers a complete solution to the reuse of deleted node labels.

In Figure 8.9, we present the details of an experiment whereby 5000 child nodes were initialized using the four bit-string dynamic labeling schemes, namely SCOOTER, SCOOBER, QED and CDBS (the label storage costs are shown in column 1). The following process was repeated 10 times:

1. A random number R was chosen between 1 and 4000
2. 1000 nodes beginning at position R were deleted.
3. 1000 new nodes were then inserted (bulk insertion) at position R.

The total label storage size of the SCOOTER and SCOOBER labels increased by approximately 4.7 times. In contrast, the total label storage size of the QED labels increased by a multiple of 33 times. In a similar manner, the total label storage size of the CDBS labels increased by a multiple of 36 times. Consequently, the ability to support the reuse of deleted node labels in a dynamic environment may offer potentially significant reductions in storage costs while maintaining compact labels and thus facilitating more efficient label comparison operations.



	Initialized 5000 Nodes	After (1000 Insertions + 1000 Deletions) * 10 times
SCOOTER	83456	391940
QED	83456	2784981
SCOOBER	87581	404597
CDBS	91568	3296856

Figure 8.9: Insertion with Deleted Node Label Reuse Storage Costs

8.5 Evaluation Framework

In Chapter 3, we presented an evaluation framework that encapsulated the template of properties that are representative of the characteristics of a *good* and holistic dynamic labeling scheme for XML. We now present the evaluation framework updated to incorporate the SCOOTER and SCOOBER dynamic labeling schemes.

Labelling Schemes	Document Order	Encoding Rep.	Xpath Eval.	Division Comp.	Recursion Alg.	Orthogonal	Compact Initial	Compact Update	Overflow Prob.	Label Reuse
DeweyID	**Hybrid**	**CT**	**F**	**F**	**F**	**N**	**N**	**N**	**N**	**N**
ORDPATH	**Hybrid**	**PF**	**F**	**N**	**F**	**N**	**N**	**N**	**N**	**N**
DLN	**Hybrid**	**CT**	**F**	**F**	**F**	**N**	**N**	**N**	**F**	**N**
LSDX	**Hybrid**	**LF**	**F**	**F**	**F**	**N**	**N**	**N**	**N**	**N**
ImprovedBinary	**Hybrid**	**LF**	**F**	**N**	**N**	**F**	**P**	**N**	**N**	**N**
QED	**Hybrid**	**S**	**F**	**N**	**N**	**F**	**F**	**N**	**F**	**N**
CDBS	**Hybrid**	**LF**	**F**	**N**	**N**	**F**	**F**	**N**	**N**	**N**
Vector	**Hybrid**	**CT**	**F**	**F**	**N**	**F**	**P**	**N**	**N**	**N**
DDE	**Hybrid**	**PF**	**F**	**N**	**N**	**N**	**P**	**N**	**N**	**N**
EXEL	**Hybrid**	**LF**	**F**	**F**	**F**	**F**	**N**	**N**	**N**	**N**
Enhanced EXEL	**Hybrid**	**LF**	**F**	**F**	**F**	**F**	**P**	**N**	**N**	**N**
SCOOTER	**Hybrid**	**S**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
SCOOBER	**Hybrid**	**LF**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**

Figure 8.10: Evaluation Framework (With SCOOTER and SCOOBER)

Both SCOOTER and SCOOBER fully satisfy all of the properties of the evaluation framework. In particular, both labeling schemes assign compact labels at document initialization and maintain compact labels under arbitrary node insertions and deletions. Neither SCOOTER nor SCOOBER are subject to the overflow problem and will never require the relabeling of existing nodes. Lastly, both SCOOTER and SCOOTER support the reuse of deleted node labels.

8.6 Summary

In this Chapter, we supplemented the theoretical evaluation of the Compact Adaptive Growth Method (in Chapter 5) with a comprehensive experimental evaluation of the SCOOTER and SCOOBER dynamic node labeling schemes. We demonstrated that both labeling schemes performed well when compared to the state-of-the-art in bit-string dynamic labeling schemes. The label storage costs are kept to a minimum and the average node label size after an arbitrary number of nodes label insertions is compact. Our evaluation demonstrates the benefits to be obtained from the adoption of SCOOTER and SCOOBER dynamic labeling schemes for XML updates.

Chapter 9

Conclusions

In this dissertation, we presented a compact and scalable encoding for updating XML based on node labeling schemes. The Compact Adaptive Growth Method facilitates the generation of highly compact, unique and ordered labels in a dynamic environment. The SCOOTER and SCOOBER labeling schemes exploit the Compact Adaptive Growth Method to assign compact node labels under various node insertion scenario, including frequently skewed insertions. This final Chapter is structured as follows: in §9.1, we review the approach and outcomes presented in this dissertation and in §9.2, we propose areas of future work.

9.1 Thesis Summary

The hypothesis put forward at the beginning of this dissertation was that it is possible to provide a dynamic labeling scheme for XML that is both *compact* and *scalable*. The overall research goal was to provide a dynamic labeling scheme that supports a high degree of functionality while minimizing the size of the node labels in the provision of this functionality. We now review our research objectives to achieve this goal.

1. We identified a set of properties that are representative of the characteristics of a *good* holistic dynamic labeling scheme for XML. For the remainder of the dissertation, we focused on four key properties that are open-research problems

at present: Label Reuse, Compact Labels, Scalability and Label Storage.

2. *Label Reuse.* We identified two attributes (*deterministic* and *consistent*) that a dynamic labeling scheme must incorporate in order to *guarantee* that a deleted node label can be reused under any node insertion scenario. We designed and specified the EBSL dynamic labeling scheme for XML that offers the first complete solution to fully supporting the reuse of deleted node labels.

3. *Compact Labels.* We presented the SCOOTER and SCOOBER dynamic label schemes that support the assignment of compact labels at document initialization. We also developed the Compact Adaptive Growth Method, a process that enables the generation of highly compact, unique and ordered labels in a dynamic environment. We then provided a suite of node insertion algorithms that exploit the Compact Adaptive Growth Method to guarantee compact node labels under any node insertion scenario.

4. *Scalability.* In order for a dynamic labeling scheme to provide scalability, two conditions must be satisfied: the dynamic labeling scheme node insertion algorithms must never require the relabeling of existing nodes; and the label storage scheme employed by the dynamic labeling scheme must not be subject to the overflow problem. The SCOOTER and SCOOBER node insertion algorithms never require the relabeling of existing nodes and both labeling schemes employ label storage schemes not subject to the overflow problem.

5. *Label Storage.* We presented the Fibonacci Label Storage Scheme, the first label storage scheme for binary-encoded bit-string labels that completely overcomes the overflow problem.

In Chapter 1, the XML data model was introduced and the challenges to updating XML were outlined. We discussed the importance of minimizing the node label size and the positive impact compact node labels have on both XML query and update services. We observed that a current obstacle in the provision of an XML update service is the limited functionality provided by existing dynamic labeling schemes and the rapid growth in node label size in a dynamic scenario.

In Chapter 2, we provided a comprehensive survey and analysis of the principle dynamic labeling schemes proposed to date. We determined the orthogonal dynamic labeling schemes to be the most versatile because most of them overcome the overflow problem. They also allow the database designer to choose whether to deploy a labeling scheme as either a prefix labeling scheme or a containment labeling scheme, depending on the type of data to be stored and the type of queries and updates to be performed. However, all orthogonal dynamic labeling schemes have a rapid label size growth rate under dynamic node insertions, and each individual orthogonal labeling scheme suffers from their own unique and distinct limitations. We also provided a detailed analysis of all four dynamic labeling schemes that claim to guarantee the reuse of deleted node labels and we demonstrated that none of these approaches offer a complete solution. In the final part of the Chapter, we provided an overview of the four approaches underlying the implementation of all existing label storage schemes to date. Each approach was shown to offer their own advantages and limitations. We concluded that no label storage scheme currently exists to enable binary-encoded bit-string dynamic labeling schemes overcome the overflow problem.

In Chapter 3, we identified the set of core properties that are representative of the characteristics of a *good* holistic dynamic labeling scheme for XML. These core properties constitute the principle components of our Evaluation framework by which we can evaluate all new and existing dynamic labeling schemes. With the aid of our Evaluation Framework, we set the scope of this dissertation to focus on four key characteristics that directly concern label size: 1) Label Reuse; 2) Compact Labels; 3) Scalability; and 4) Label Storage. The Label Reuse and Compact Labels characteristics are open research problems with no know solutions. The Scalability and Label Storage characteristics are open research questions for binary-encoded bit-string dynamic labeling schemes. These four problems were addressed in Chapters 4 through 7 in this dissertation.

In Chapter 4, we provided a detailed description of the deleted node label reuse problem. We defined two new attributes (*deterministic* and *consistent*) and demonstrated how the absence of these attributes are the underlying reason why all existing approaches to solving this problem offer incomplete solutions. We presented the EBSL dynamic labeling scheme which offers a complete solution to fully support the reuse of deleted node labels under any node insertion scenario. However, the size of the node labels assigned at document initialization increases linearly. Consequently, the EBSL node labels are not compact. Therefore, though the solution for the reuse of node labels provided by EBSL is beneficial, there are comparatively greater gains to be obtained by ensuring the size of the labels are compact both at document initialization and during node insertions.

In Chapter 5, we focused on the problem of Compact Labels. We introduced the SCOOTER dynamic labeling scheme and presented two *deterministic* algorithms for the generation and assignment of compact labels at document initialization. These algorithms facilitate sequential and parallel processing implementations. We then presented the Compact Adaptive Growth Method which provides a process for the generation of highly compact, unique and ordered labels in a dynamic environment. The labels generated by the Compact Adaptive Growth Method may be several orders of magnitudes smaller than labels generated by all existing bit-string dynamic labeling schemes under identical update scenarios. We also provided a comprehensive theoretical evaluation of the Compact Adaptive Growth Method in this Chapter.

In Chapter 6, we presented the suite of SCOOTER node label insertion algorithms that exploit the Compact Adaptive Growth Method to guarantee the generation of node labels with a highly constrained growth rate in label size under various node insertions scenarios, including frequently skewed node insertions. The node label insertion algorithms will never require the relabeling of existing nodes under any node insertion or deletion scenario. Also, given that SCOOTER labels are encoded using quaternary codes and employ the separator label storage scheme, the SCOOTER labeling scheme fully supports the Scalability characteristic.

Furthermore, all of the SCOOTER node insertion algorithms support the reuse of deleted node labels. However, although the EBSL label reuse strategy always guarantees a deleted node label will be reused if one is available at the position of insertion, the SCOOTER label reuse strategy guarantees that the label selected for reuse will never be larger than the maximum allowable label length determined by the rules of the Compact Adaptive Growth Method. The SCOOTER labeling scheme may select a deleted node label for reuse that is smaller than the maximum allowable label length, or indeed the smallest possible deleted node label available. However, it is not the existence of a deleted node label that determines whether it is reused or not (as in the EBSL reuse strategy), but rather the rules of the Compact Adaptive Growth Method governing label size that determine when and which deleted node label should be reused. The priority of the SCOOTER node insertions algorithms is always to ensure that node labels generated have a label size growth rate determined by the Compact Adaptive Growth Method, and consequently will have a highly constrained growth rate.

In Chapter 7, we presented the SCOOBER dynamic labeling scheme, which may be described as a binary-encoded version of the quaternary-encoded SCOOTER labeling scheme. The SCOOBER labeling scheme inherits all the properties and advantages of the SCOOTER labeling scheme except scalability (because it is unable to employ the separator label storage scheme). To overcome this problem, we introduced the Fibonacci label storage scheme that exploits the properties of the Fibonacci sequence and the Zeckendorf Representation to generate a Fibonacci coded binary string to encode the length of a label. The Fibonacci coded binary string may be stored immediately before a binary-encoded bit-string label to construct a length field label storage scheme that completely avoids the relabeling of nodes under any update scenario.

In Chapter 8, we provided a comprehensive evaluation of the contributions presented in this dissertation. We evaluated both the storage costs and the computational costs of assigning node labels at document initialization for several state-of-the-art dynamic labeling schemes. We also evaluated the storage and computational costs of the labeling schemes under various node insertion and deletion scenarios and under various update loads. The SCOOTER and SCOOBER dynamic labeling schemes provided very good results across a wide range of evaluation scenarios and demonstrate the positive benefits to be obtained from the contributions presented in this dissertation.

9.1.1 Impact on the Field

The contributions summarized above have a number of important benefits for the field of XML Database Management Systems and dynamic updates for XML. In [12], the authors point out that *"the length of the assigned labels is an important criterion in the quality of any such labeling scheme. This length determines the size of the index structure that contains the labels and thereby the feasibility of keeping this index in main memory."* The ability to determine structural relationships between nodes from indices in main memory provides significant performance benefits for XML query and update services over alternative approaches that require a database lookup, table-scan or file system access.

In [38], the authors of the QED and CDBS labeling schemes highlight as part of their future work the most significant problem yet to be solved - *"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"*. This is precisely one of the problems we have solved in this dissertation. Both SCOOTER and SCOOBER labeling schemes completely avoid the relabeling of existing nodes while assigning labels with a highly constrained growth rate in label size under frequently skewed node insertions.

In [25], the authors discuss the need for fast node identification in the management of XML documents and conclude *"For dominant processing tasks such as declarative, index-based query evaluation, tree navigation, and concurrency control, fine-grained access to the documents is indispensable. Thus, efficient and effective node labeling resilient to arbitrary document modifications is of outmost importance."* The SCOOTER and SCOOBER dynamic labeling schemes are resilient to arbitrary document modifications by never requiring the relabeling of existing nodes, offer a functionality-rich set of features and provide for efficient label comparison operations by maintaining compact labels that may be stored and processed in main memory.

9.2 Future Work

In this section, we highlight a number of research avenues for further exploration. We classify them as either short-term or long-term.

9.2.1 Short Term Research Avenues

The following research tasks could be reasonably completed in the short term.

A Dynamic Labeling Scheme for Streaming Data. Almost all existing dynamic labeling schemes to date need to know in advance the number of nodes in an XML tree before labels can be assigned by their `AssignInitialLabels` Algorithm. Both SCOOTER and SCOOBER labeling schemes suffer from this limitation. Hence, they cannot be deployed as a labeling scheme for on-the-fly labeling of streaming XML data. However, the SCOOTER `InsertNewNodeAfterRightMostNode` Algorithm could potentially be employed as an `AssignInitialLabels` Algorithm for streaming XML data. One problem to be overcome is that the size of the initially assigned labels would grow at the rate dictated by the Compact Adaptive Growth Method (CAGM). The CAGM growth rate is highly constrained when generating node labels during dynamic node insertions, but comparatively not so compact when compared to the label sizes generated by the SCOOTER `AssignInitialLabels` Algorithm. Therefore, the first short term research avenue is to adapt and refine the Compact Adaptive Growth Method such that the `InsertNewNodeAfterRightMostNode` Algorithm grows at a much slower rate when assigning labels to streaming XML data.

An Efficient Random Access `AssignInitialLabels` Algorithm. In this dissertation, we presented two algorithms for assigning labels at document initialization - a sequential assignment algorithm and a random access assignment algorithm. However, during our experimental evaluation, we determined the runtime performance of the random access assignment algorithm to be approximately five times slower than the sequential assignment algorithm. The second short term goal is to identify a more efficient method for the random access `AssignInitialLabels`

Algorithm using string processing and thus, avoid the costly division and modulus operations that must be performed l times for each label, where l is the length of the label.

9.2.2 Long Term Research Avenues

The long term research avenues for our research seek to extend the functionality provided by our labeling schemes to new areas and to try and support new features without the loss of existing functionality.

XML Version Control at the Node level. In [18], the authors present a unified framework for the version control of XML data and documents that can support fine-grained temporal query capabilities. They recognize the need to work at the level of the XQuery Data Model (XDM) and to extend it with a time dimension. In their prototype, they define the concept of a *node timeline* which is a succession of all node items sharing a given identity. A node timeline is identified by a URI. A *tree timeline* is a succession of trees whose roots share a given identity. A tree timeline is identified by the URI of its node timeline. We believe it may be possible to extend the Compact Adaptive Growth Method and SCOOBER dynamic labeling scheme to replace the URI with SCOOBER labels that will provide all of the functionality of the existing URI but with the added benefits of the SCOOBER labels.

Node-Level XML Locking Protocols. In [6] and [24], the authors provide an in-depth overview of the issues and challenges to be overcome in the provision of locking protocols for XML databases. The purpose of an XML lock protocol is to facilitate read and write access by two or more concurrent transactions to the same documents. Most existing approaches support updates at the document level. However, with the growth in the adoption of XML databases and XML repositories, efficient concurrent transaction processing and updates is desirable. In [24], the authors built a prototype native XML database system called XML Transaction Coordinator (XTC) with the principle design goal of supporting fine-grained lock protocols for the collaborative processing of XML documents. They concluded that *the node labeling scheme is of utmost importance for the internal processing of XML trees and, in particular, for locking.* The authors employ the DeweyID labeling scheme and compress the labels using Huffman encoding. The resultant codes preserve their order when compared at byte level and thus facilitate fast label comparison operations. A very useful research avenue would be to investigate the use of bit-level or byte level compression for SCOOTER and SCOOBER labels such that they preserve their byte order and can thus, facilitate label comparisons operations based on partial labels.

Appendix A

Related Peer Reviewed Publications

1. **FibLSS: A Scalable Label Storage Scheme for Dynamic XML Updates.**

   Martin F. O'Connor and Mark Roantree

   In: 17th East-European Conference on Advances in Databases and Information Systems, (ADBIS 2013), Genoa, Italy, 1–4 September 2013, LNCS 8133, pp. 218–231, Springer Proceedings.

2. **SCOOTER: A Compact and Scalable Dynamic Labeling Scheme for XML Updates.**

   Martin F. O'Connor and Mark Roantree

   In: 23rd International Conference on Database and Expert Systems Applications, (DEXA 2012), Vienna, Austria, 3–7 September 2012, LNCS 7446, Part 1 pp. 26–40, Springer Proceedings.

3. **Data Transformation and Query Management in Personal Health Sensor Networks.**

   Mark Roantree, Jie Shi, Paolo Cappellari, Martin F. O'Connor, Michael Whelan and Niall Moyna

   In: Journal of Network and Computer Applications, Vol. 35, Issue. 4, ISSN

1084–8045, pp.1191–1202, July 2012.

4. **EBSL: Supporting Deleted Node Label Reuse in XML.**
   Martin F. O'Connor and Mark Roantree
   In: 7th International XML Database Symposium, (XSym 2010), Singapore, 17 September 2010, LNCS 6309, pp.73–87, Springer Proceedings.

5. **Desirable Properties for XML Update Mechanisms.**
   Martin F. O'Connor and Mark Roantree
   In: Updates in XML: Proceedings of the EDBT/ICDT 2010 Workshops, Lausanne, Switzerland, 22–26 March 2010, Vol. 426, Article No.: 23, ACM Proceedings.

6. **Querying XML Data Streams from Wireless Sensor Networks: An Evaluation of Query Engines.**
   Martin F. O'Connor, Kenneth Conroy, Mark Roantree, Alan F. Smeaton and Niall Moyna
   In: 3rd International Conference on Research Challenges in Information Science (RCIS 2009), Fès, Morocco, 22–24 April 2009, pp.23–30, IEEE Proceedings.

7. **Query Management in a Sensor Environment.**
   Martin F. O'Connor, Vincent Andrieu, and Mark Roantree
   In: 14th International Conference on Parallel and Distributed Systems (ICPADS 2008), Melbourne, Australia, 8–10 December 2008, pp.835-840, IEEE Proceedings.

Appendix B

SCOOTER Algorithms

B.1 The SCOOTER `Decrement` Algorithm

Algorithm B.1 is the SCOOTER `Decrement` Algorithm. Given a SCOOTER self_label N_{right} and the maximum allowable self_label length *maxSize*, this algorithm identifies the label that is the immediate lexicographical decrement of the input label N_{right}. There are three conditions governing this algorithm. This algorithm should never receive:

1. An input label with a length greater than *maxSize*.
2. An input label consisting of all "1" digits.
3. An input label of length *maxSize* **and** an input label with zero or more consecutive "1" digits followed by and ending with a single "2" digit. For example, "2", "12", "112", 1112" and so on.

Algorithm B.1: SCOOTER `Decrement`

```
/* Decrement lexicographically a node self_label such that maximum permissible length
   is maxSize.                                                                       */
input  : $N_{right}$ - a node self_label;
         $maxSize$ - maximum number of digits allowed in self_label.
output: $N_{new}$ - a new self_label that is the immediate lexicographical decrement of $N_{right}$, that is
         $N_{new} \prec N_{right}$.
begin
    $N_{temp} \longleftarrow N_{right}$;
    if $(length(N_{right}) == maxSize)$ then
        if $(last\ digit\ in\ N_{temp}\ is\ '3')$ then
            $N_{temp} \longleftarrow N_{temp}$ with last digit changed to '2';
        else if $(last\ digit\ in\ N_{temp}\ is\ '2')$ then
            $N_{temp} \longleftarrow N_{temp}$ with last digit removed;
            while $(last\ digit\ in\ N_{temp}\ is\ '1')$ do
                $N_{temp} \longleftarrow N_{temp}$ with last digit removed;
            end
        else if $(last\ digit\ in\ N_{temp}\ is\ '1')$ then
            while $(last\ digit\ of\ N_{temp}\ is\ '1')$ do
                $N_{temp} \longleftarrow N_{temp}$ with last digit removed;
            end
        end
    else if $(length(N_{right}) < maxSize)$ then
        if $(last\ digit\ in\ N_{temp}\ is\ '3')$ then
            $N_{temp} \longleftarrow N_{temp}$ with last digit changed to '2';
        else if $(last\ digit\ in\ N_{temp}\ is\ '2')$ then
            $N_{temp} \longleftarrow N_{temp}$ with last digit changed to '1';
        else if $(last\ digit\ in\ N_{temp}\ is\ '1')$ then
            while $(last\ digit\ of\ N_{temp}\ is\ '1')$ do
                $N_{temp} \longleftarrow N_{temp}$ with last digit removed;
            end
            if $(last\ digit\ in\ N_{temp}\ is\ '3')$ then
                $N_{new} \longleftarrow N_{temp}$ with last digit changed to '2';
            else if $(last\ digit\ in\ N_{temp}\ is\ '2')$ then
                $N_{new} \longleftarrow N_{temp}$ with last digit changed to '1';
            end
        end
        for $(i = length(N_{temp}) + 1;\ i \leq maxSize;\ i{+}{+})$ do
            $N_{temp} \longleftarrow N_{temp} \oplus 3$;
        end
    end
    $N_{new} \longleftarrow N_{temp}$;
    return $N_{new}$;
end
```

Appendix C

Compact Adaptive Growth (Binary) Evaluation

C.1 Compact Adaptive Growth (Binary) Evaluation

In this section, we present a comprehensive theoretical evaluation of our Compact Adaptive Growth Method for binary encodings. We perform our evaluation in the context of the generation of node labels for the SCOOBER dynamic labeling scheme. Our overall objective is to quantify the relationship between the growth rate of the number of labels to be inserted with the corresponding growth rate in label size. In order to quantify this relationship, we must address a number of goals; For any positive integer N:

1. How many unique SCOOBER labels are there with a length less than or equal to N? We address this question in Theorem C.1.
2. How many new labels are available for insertion after the Nth adaptive increase in the prefix length? We address this in Theorem C.2.
3. What is the length of the prefix after N adaptive increases in the prefix length? We address this question in Theorem C.3.
4. What is the total number of labels available for insertion after N adaptive increases in the prefix length?. We address this question in Theorem C.4.

5. What is the ordinal number of the first label available for use after N adaptive increases in the prefix length? We address this question in Theorem C.5.

6. Finally, what is the maximum size in bits of a SCOOBER self_label after N adaptive increases in the prefix length? We address this question in Theorem C.6.

Theorem C.1. *For any positive integer N, the total number of unique SCOOBER labels with a length less than or equal to N is:* $(2^N - 1)$.

Proof: Given a SCOOBER label may contain only the two bits {"0" and "1"}, then only one unique label may be generated using one bit, namely "1" (because a SCOOBER label cannot terminate with a "0" bit). When assigning SCOOBER labels with a length of two bits, the first bit can be one of {"0", "1"}, and the second bit can only be "1". Therefore, there are $(2^1 * 1) = 2$ possible labels of length 2 bits. When assigning SCOOBER labels with a length of three bits, the first two bits can be one of {"0", "1"}, and the third bit can only be "1". Therefore, there are $(2^2 * 1) = 4$ possible labels of length 3 bits. When assigning labels of length k, the first $k-1$ bits can be one of {"0", "1"}, and the k^{th} bit can only be "1". Therefore, the number of unique SCOOBER labels of length k is $(2^{k-1} * 1)$. Hence, the total number of unique SCOOBER labels with a length less than or equal to N is:

$$\sum_{k=1}^{N} 2^{k-1} = (2^0) + (2^1) + (2^2) + (2^3) + \cdots + (2^{N-1}) \tag{C.1}$$

We know from number theory that the sum of x to the power of n from 0 to (N−1) is:

$$\sum_{n=0}^{N-1} x^n = \frac{x^N - 1}{x - 1}$$

Therefore equation C.1 becomes:

$$\begin{aligned}\sum_{k=1}^{N} 2^{k-1} &= \frac{2^N - 1}{2 - 1} \\ &= 2^N - 1\end{aligned}$$

□

Given that we now know how many unique SCOOBER labels there are with a length $\leq$ N (from Theorem C.1), we are in a position to determine how many new labels are available for insertion after the Nth adaptive increase in the prefix length.

Theorem C.2. *The number of new labels available for insertion after the N^{th} adaptive increase in the prefix length is 2^{N-1} labels.*

Proof:

From Theorem C.1, we know the total number of unique SCOOBER labels with a length less than or equal to N that can be assigned by the `AssignInitialLabels_Sequential` Algorithm is $2^N - 1$. However, after the Nth adaptive increase in the prefix length, the number of new labels available for insertion is equal to the number of labels that can be assigned by the `AssignInitialLabels_Sequential` Algorithm with a length of *maximumPostfixLength*. From CAR$_b$7.7, we know the *maximumPostfixLength* after N adaptive increases in the prefix length is simply N. Also, from CAR$_b$7.8 we know the first postfix after the Nth adaptive increase in the prefix length is assigned the label at the midpoint between 1 and $2^N - 1$. The midpoint for binary-encoded labels between 1 and $2^N - 1$ is computed as $\frac{2^N}{2}$ (which can be rewritten as 2^{N-1}). Consequently, the number of new labels available for insertion after the Nth adaptive increase in the prefix length is 2^{N-1} labels. □

In order to determine the maximum length (in bits) of a label after N adaptive increases in the prefix length, we must first determine the length of the prefix itself after N adaptive increases in the prefix length.

Theorem C.3. *The length of the prefix after N adaptive increases in the prefix length is:* $\dfrac{N^2 - N}{2} + 1$ *(please refer to last row of column four in Table 7.3)*

Proof: The proof is provided by Theorem 5.6.

□

We determined how many new labels are available for insertion after the Nth adaptive increase in the prefix length (from Theorem C.2). This information allows us to determine the total number of all labels available for insertion after N adaptive increases in the prefix length.

Theorem C.4. *The total number of labels available for insertions after N adaptive increases in the prefix length is:* $2^N - 1$ *(please refer to last row of column three in Table 7.3)*

Proof: From Theorem C.2, we know that after each nth adaptive increase in the prefix length, the number of new labels available at that prefix length is 2^{n-1}. Therefore, after N adaptive increases in the prefix length, the total number of labels available is:

$$\sum_{n=1}^{N} 2^{n-1} \qquad = (2^0) + (2^1) + (2^2) + (2^3) + \cdots + (2^{N-1}) \tag{C.2}$$

We know from number theory that the sum of x to the power of n from 0 to (N−1) is:

$$\sum_{n=0}^{N-1} x^n \qquad = \frac{x^N - 1}{x - 1}$$

Therefore equation C.2 becomes:

$$\begin{aligned}\sum_{n=1}^{N} 2^{n-1} \qquad &= \frac{2^N - 1}{2 - 1}\\ &= 2^N - 1\end{aligned}$$

□

Theorem C.5. *The ordinal number of the first label available for use after N adaptive increases in the prefix length is* 2^{N-1} *(please refer to last row of column two in Table 7.3)*

Proof: The ordinal number of the first label available for use after N adaptive increases in the prefix length is always one more than the total number of labels available after N−1 adaptive increases in the prefix length. From Theorem C.4, we know the total number of labels available after N adaptive increases in the prefix length is $2^N - 1$ Thus, by replacing every occurance of N with N−1 and adding one to the final result, we can determine the ordinal number of the first label available for use after N adaptive increases as:

$$= 2^{N-1} - 1 + 1$$

$$= 2^{N-1}$$

□

Theorem C.6. *The maximum size in bits of a SCOOBER self_label after N adaptive increases in the prefix length is:* $\dfrac{N^2+N}{2} + 1$ *(please refer to last row of column six in Table 7.3)*

Proof: There is a one-to-one mapping between the digits in a binary-encoded bit-string label and the bits used to physically store the binary-encoded bit-string label. Consequently, the proof is provided by Theorem 5.7.

□

C.1.1 Analysis

All existing bit-string dynamic labeling schemes have a minimum of a one-bit-per-node-insertion label growth rate. Therefore, as the number of label insertions increases linearly, the corresponding growth in label size is at least linear. It follows that as the number of labels to be inserted increases exponentially, the corresponding growth in label size is also exponential for all bit-string dynamic labeling schemes. However, as the number of SCOOBER labels to be inserted increases exponentially ($O(2^N)$ from Theorem C.4), the corresponding growth in label size is quadratic ($O(N^2)$ from Theorem C.6). Consequently, the Compact Adaptive Growth Method ensures SCOOBER node labels have a highly constrained growth rate under frequent node insertions. For example, after ten thousand node insertions and one hundred thousand node insertions, the largest self_labels generated by all binary-encoded bit-string dynamic labeling schemes have a minimum length of 10,000 bits and 100,000 bits respectively. In contrast, after ten thousand node insertions and one hundred thousand node insertions, the largest SCOOBER self_labels have a maximum length of 106 bits and 154 bits respectively.

Appendix D

SCOOBER Algorithms

D.1 The SCOOBER `InsertNewNodeAfterRightmostNode` Algorithm

D.2 The SCOOBER `InsertNewNodeBeforeLeftmostNode` Algorithm

D.3 The SCOOBER `Insertion_LongerThan` Algorithm

D.4 The SCOOBER `Insertion_EqualTo` Algorithm

D.5 The SCOOBER `Insertion_LessThanPrefix` Algorithm

D.6 The SCOOBER `generateAdaptivePostfix` Algorithm

D.7 The SCOOBER `Insertion_LessThanNotPrefix` Algorithm

D.8 The SCOOBER `Decrement` Algorithm

Algorithm D.1: SCOOBER `InsertNewNodeAfterRightmostNode`

```
input  : $N_{left}$ - the current rightmost node self_label.
output: $N_{new}$ - a new self_label such that $N_{left} \prec N_{new}$.
begin
    if (first digit in $N_{left}$ is '0') then
        $N_{new} \longleftarrow$ '1';
    else if (first digit in $N_{left}$ is '1') then
        numConsecOnes $\longleftarrow$ the number of consecutive '1' digits at start of $N_{left}$;
        /* Compute the prefixLength and postfixLength based on numConsecThrees.      */
        prefixLength, postfixLength $\longleftarrow$ ComputePrefixPostfixLengths(numConsecOnes);
        labelLength $\longleftarrow$ prefixLength + postfixLength;
        postfix $\longleftarrow$ substring($N_{left}$, prefixLength + 1, labelLength);
        if (postfix is not empty) then
            if (last symbol in postfix is '0') then
                postfix $\longleftarrow$ postfix with last symbol changed to '1';
            else
                postfix $\longleftarrow$ Increment(postfix, postfixLength);
            end
        else if (postfix is empty) then
            postfix $\longleftarrow$ 1;
        end
        prefix = null;
        while (i=1; i $\leq$ prefixLength; i++) do
            prefix $\longleftarrow$ prefix $\oplus$ 1;
        end
        $N_{new} \longleftarrow$ prefix $\oplus$ postfix;
    end
    return $N_{new}$;
end
```

Algorithm D.2: SCOOBER `InsertNewNodeBeforeLeftmostNode`

input : N_{right} - the current leftmost node self_label.
output: N_{new} - a new self_label such that $N_{new} \prec N_{right}$.

```
begin
    if (first digit in $N_{right}$ is '1') then
        $N_{new}$ ⟵ '01';
    else if (first digit in $N_{right}$ is '0') then
        numConsecZeros ⟵ the number of consecutive '0' digits at start of $N_{right}$;
        if (numConsecZeros == (length($N_{right}$) − 1)) then
            numConsecZeros ⟵ numConsecZeros + 1;
        end
        /* Compute the prefixLength and postfixLength based on numConsecZeros.   */
        prefixLength, postfixLength ⟵ ComputePrefixPostfixLengths(numConsecZeros);
        labelLength ⟵ prefixLength + postfixLength;
        if (numConsecZeros == (labelLength − 1)) and (length($N_{right}$) > labelLength) then
            $N_{new}$ ⟵ substring($N_{right}$, 1, labelLength);
        else
            postfix ⟵ substring($N_{right}$, prefixLength + 1, labelLength);
            if (postfix is not empty) then
                postfix ⟵ Decrement(postfix, postfixLength);
            else if (postfix is empty) then
                postfix ⟵ 1;
            end
            prefix = null;
            while (i=1; i ≤ prefixLength; i++) do
                prefix ⟵ prefix ⊕ 0;
            end
            $N_{new}$ ⟵ prefix ⊕ postfix;
        end
    end
    return $N_{new}$;
end
```

Algorithm D.3: SCOOBER `Insertion_LongerThan`

```
/* Insert node between two consecutive sibling nodes; length($N_{left}$) > length($N_{right}$).
   */
input  : $N_{left}$ - the left node self_label.
         $N_{right}$ - the right node self_label.
output: $N_{new}$ - a new self_label such that $N_{left} \prec N_{new} \prec N_{right}$.
begin
    if (length($N_{left}$) > length($N_{right}$)) then
        P ⟵ first position of difference between $N_{left}$ and $N_{right}$;
        affix ⟵ substring($N_{left}$, 1, P);
        if (P < length($N_{right}$)) then
            $N_{new}$ ⟵ Increment(affix, length(affix));
        else if (P == length($N_{right}$)) then
            $N_{temp}$ ⟵ substring($N_{left}$, P + 1, length($N_{left}$));
            numConsecOnes ⟵ the number of consecutive '1' digits at start of $N_{temp}$;
            if (numConsecOnes == 0) then
                $N_{new}$ ⟵ affix ⊕ 1;
            else if (numConsecOnes > 0) then
                maxPrefixLength, maxPostfixLength ⟵
                ComputePrefixPostfixLengths(numConsecOnes);
                maxLabelLength ⟵ maxPrefixLength + maxPostfixLength;
                postfix ⟵ substring($N_{temp}$, maxPrefixLength + 1, maxLabelLength);
                if (postfix is not empty) then
                    if (last symbol in postfix is '0') then
                        postfix ⟵ postfix with last symbol changed to '1';
                    else
                        postfix ⟵ Increment(postfix, maxPostfixLength);
                    end
                else if (postfix is empty) then
                    postfix ⟵ 1;
                end
                prefix = null;
                while (i=1; i ≤ maxPrefixLength; i++) do
                    prefix ⟵ prefix ⊕ 1;
                end
                $N_{new}$ ⟵ affix ⊕ prefix ⊕ postfix;
            end
        end
    end
    return $N_{new}$;
end
```

Algorithm D.4: SCOOBER `Insertion_EqualTo`

```
/* Insert node between two consecutive sibling nodes; length($N_{left}$) = length($N_{right}$).
   */
input  : $N_{left}$ - the left node self_label.
         $N_{right}$ - the right node self_label.
output: $N_{new}$ - a new self_label such that $N_{left} \prec N_{new} \prec N_{right}$.
begin
    if (length($N_{left}$) == length($N_{right}$)) then
        P ⟵ first position of difference between $N_{left}$ and $N_{right}$;
        $N_{new}$ ⟵ substring($N_{right}$, 1, P));
    end
    return $N_{new}$;
end
```

Algorithm D.5: SCOOBER `Insertion_LessThanPrefix`

```
/* Insert node between two consecutive sibling nodes; $N_{left}$ is a prefix of $N_{right}$.    */
input : $N_{left}$ - the left node self_label.
        $N_{right}$ - the right node self_label.
output: $N_{new}$ - a new self_label such that $N_{left} \prec N_{new} \prec N_{right}$.
begin
    if (length($N_{left}$) < length($N_{right}$)) and ($N_{left}$ is a prefix of $N_{right}$) then
        $N_{temp}$ ⟵ substring($N_{right}$, length($N_{left}$) + 1, length($N_{right}$));
        numConsecZeros ⟵ the number of consecutive '0' digits at start of $N_{temp}$;
        if ((length($N_{left}$) + numConsecZeros) == (length($N_{right}$) − 1) then
            numConsecZeros ⟵ numConsecZeros + 1;
        end
        minLabelLength ⟵ length($N_{left}$) + numConsecZeros + 1;
        maxPrefixLength, maxPostfixLength ⟵ ComputePrefixPostfixLengths(minLabelLength − 1);
        maxLabelLength ⟵ maxPrefixLength + maxPostfixLength;
        prefix ⟵ $N_{left}$;
        for (i = length($N_{left}$); i < maxPrefixLength; i++) do
            prefix ⟵ prefix ⊕ 0;
        end
        actualPrefixLength ⟵ length(prefix);
        allowablePostfixLength ⟵ maxLabelLength − actualPrefixLength;
        postfixDefault ⟵ 1;
        postfix ⟵ substring($N_{right}$, actualPrefixLength + 1, maxLabelLength);

        if (length(postfix) == 0) then
            postfix ⟵ postfixDefault;
        else if (length(postfix) > 0) then
            numZeros ⟵ the number of consecutive '0' digits at start of postfix;
            if (numZeros == length(postfix) − 1) and (length(postfix) == allowablePostfixLength) then
                postfix ⟵ postfix;
            else
                postfix ⟵ generateAdaptivePostfix(postfix, allowablePostfixLength);
            end
        end
        $N_{new}$ ⟵ prefix ⊕ postfix;
    end
    return $N_{new}$;
end
```

Algorithm D.6: SCOOBER `generateAdaptivePostfix`

```
/* Generate a new postfix according to the compact adaptive growth method.    */
input : postfix - a postfix (extracted from $N_{right}$ by the Insertion_LongerThan algorithm).
        maxPostfixLength - the maximum allowable length of the new postfix.
output: newPostfix - a new postfix that is lexicographically less than the input postfix.
begin
    if (first digit in postfix is '1') then
        if (length(postfix) == maxPostfixLength) then
            newPostfix ⟵ 1;
        else if (length(postfix) > 1) then
            newPostfix ⟵ a sequence of '0' digits of length(maxPostfixLength − 2) ⊕ 1;
        else if (length(postfix) == 1) then
            newPostfix ⟵ 01;
        end
    else if (first digit in postfix is '0') then
        newPostfix ⟵ Decrement(postfix, maxPostfixLength);
    end
    return newPostfix;
end
```

Algorithm D.7: SCOOBER `Insertion_LessThanNotPrefix`

```
/* Insert node between two consecutive sibling nodes; length($N_{left}$) < length($N_{right}$);
   $N_{left}$ not a prefix of $N_{right}$.                                              */
input  : $N_{left}$ - the left node self_label.
         $N_{right}$ - the right node self_label.
output: $N_{new}$ - a new self_label such that $N_{left} \prec N_{new} \prec N_{right}$.
begin
    if (length($N_{left}$) < length($N_{right}$)) and ($N_{left}$ is NOT a prefix of $N_{right}$) then
        P ⟵ first position of difference between $N_{left}$ and $N_{right}$;
        if (P == 1) then
            Nnew ⟵ 1;
        else if (P > 1) then
            maxPrefixLength ⟵ maxPostfixLength ⟵ 1;
            maxLabelLength ⟵ maxPrefixLength + maxPostfixLength;
            while (maxLabelLength < P) do
                maxPrefixLength ⟵ maxLabelLength;
                maxPostfixLength ⟵ maxPostfixLength + 1;
                maxLabelLength ⟵ maxPrefixLength + maxPostfixLength;
            end
            prefix ⟵ substring($N_{left}$, 1, P − 1);
            actualPrefixLength ⟵ length(prefix);
            actualPostfixLength ⟵ maxLabelLength − actualPrefixLength;
            $N_{temp}$ ⟵ substring($N_{left}$, P, P + (actualPostfixLength − 1));
            postfix ⟵ Increment($N_{temp}$, actualPostfixLength);
            $N_{new}$ ⟵ prefix ⊕ postfix;
        end
    end
    return $N_{new}$;
end
```

Algorithm D.8: SCOOBER `Decrement`

```
/* Decrement lexicographically a node self_label such that maximum length is maxSize */
input  : $N_{right}$ - a node self_label;
         maxSize - maximum number of digits allowed in self_label.
output: $N_{new}$ - a new self_label that is the immediate lexicographical decrement of $N_{right}$, such
        that $N_{new} \prec N_{right}$.
begin
    $N_{temp}$ ⟵ $N_{right}$;
    if (length($N_{right}$) == maxSize) then
        if (last digit in $N_{temp}$ is '1') then
            $N_{temp}$ ⟵ $N_{temp}$ with last digit removed;
        end
        while (last digit in $N_{temp}$ is '0') do
            $N_{temp}$ ⟵ $N_{temp}$ with last digit removed;
        end
    else if (length($N_{right}$) < maxSize) then
        if (last digit in $N_{temp}$ is '1') then
            $N_{temp}$ ⟵ $N_{temp}$ with last digit changed to '0';
        else if (last digit in $N_{temp}$ is '0') then
            while (last digit of $N_{temp}$ is '0') do
                $N_{temp}$ ⟵ $N_{temp}$ with last digit removed;
            end
            $N_{temp}$ ⟵ $N_{temp}$ with last digit changed to '0';
        end
        for (i = length($N_{temp}$) + 1; i ≤ maxSize; i++) do
            $N_{temp}$ ⟵ $N_{temp}$ ⊕ 1;
        end
    end
    $N_{new}$ ⟵ $N_{temp}$;
    return $N_{new}$;
end
```

Bibliography

[1] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'89)*, pages 253–262, Portland, Oregon, 31 May - 02 June 1989. ACM.

[2] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, pages 141–152, San Jose, CA, 26 February - 01 March 2002. IEEE Computer Society.

[3] Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. QRS: A Robust Numbering Scheme for XML Documents. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'03)*, pages 705–707, Bangalore, India, 05-08 March 2003. IEEE Computer Society.

[4] Dong Chan An, So Mi Park, and Seog Park. Efficient Secure Labeling Method under Dynamic XML Data Streams. In *Proceeding of the Third International Workshop on Security (IWSEC'08)*, volume 5312 of *LNCS*, pages 246–260, Kagawa, Japan, 25-27 November 2008. Springer-Verlag.

[5] Alberto Apostolico and Aviezri S. Fraenkel. Robust Transmission of Unbounded Strings Using Fibonacci Representations. *IEEE Transactions on Information Theory*, 33(2):238–245, March 1987.

[6] Sebastian Bächle, Theo Härder, and Michael Peter Haustein. Implementing and Optimizing Fine-Granular Lock Management for XML Document Trees. In *Proceedings of the 14th International Conference on Database Systems for Advanced Applications (DASFAA'09)*, volume 5463 of *LNCS*, pages 631–645, Brisbane, Australia, 21-23 April 2009. Springer-Verlag.

[7] Timo Böhme and Erhard Rahm. Supporting Efficient Streaming and Insertion of XML Data in RDBMS. In *Proceedings of the 3rd International Workshop of Data Integration over the Web (DIWeb'04) (held in conjunction with CAiSE'04)*, pages 70–81, Riga, Latvia, 08 June 2004.

[8] Federico Cavalieri, Giovanna Guerrini, and Marco Mesiti. Dynamic Reasoning on XML Updates. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT'11)*, pages 165–176, Uppsala, Sweden, 21-24 March 2011. ACM.

[9] Federico Cavalieri, Giovanna Guerrini, and Marco Mesiti. Reverting the Effects of XQuery Update Expressions. In *Proceedings of the 28th British National Conference on Databases (BNCOD'11)*, volume 7051 of *LNCS*, pages 167–181, Manchester, UK, 12-14 July 2011. Springer-Verlag.

[10] Don Chamberlin, Denise Draper, Mary Fernández, Michael Kay, Jonathan Robie, Michael Rys, Jérôme Siméon, Jim Tivy, and Philip Wadler. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley, 2004.

[11] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling Dynamic XML trees. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'02)*, pages 271–281, Wisconsin, USA, 03-06 June 2002. ACM.

[12] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling Dynamic XML Trees. *SIAM Journal on Computing (SICOMP)*, 39(5):2048–2074, 2010.

[13] Melvil Dewey. Dewey Decimal Classification (DDC) system. Available from: `http://www.oclc.org/dewey/` [Accessed 06 January 2013].

[14] Paul F. Dietz. Maintaining Order in a Linked List. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC'82)*, pages 122–127, California, USA, 05-07 May 1982. ACM.

[15] Maggie Duong and Yanchun Zhang. LSDX: A New Labelling Scheme for Dynamically Updating XML Data. In *Proceedings of the 16th Australasian Database Conference (ADC'05)*, volume 39, pages 185–193, Newcastle, Australia, 2005. Australian Computer Society.

[16] Maggie Duong and Yanchun Zhang. Dynamic Labelling Scheme for XML Data Processing. In *Proceedings of the On The Move to Meaningful Internet Systems Part II (OTM'08)*, volume 5332 of *LNCS*, pages 1183–1199, Monterrey, Mexico, 09-14 November 2008. Springer-Verlag.

[17] Peter Elias. Universal Codeword Sets and Representations of the Integers. *IEEE Transactions on Information Theory*, 21(2):194 – 203, March 1975.

[18] Ghislain Fourny, Daniela Florescu, Donald Kossmann, and Markos Zacharioudakis. A Time Machine for XML. *Technical Report, ETH Zurich*, (734):10, 2012.

[19] Aviezri S. Fraenkel and Shmuel T. Kleinb. Robust Universal Complete Codes for Transmission and Compression. *Elsevier Journal of Discrete Applied Mathematics*, 64(1):31–55, 04 January 1996.

[20] Charles F. Goldfarb. *The SGML handbook*. Oxford: Clarendon Press, New York, USA, 1990.

[21] Torsten Grust. Accelerating XPath Location Steps. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, pages 109–120, Wisconsin, USA, 03-06 June 2002. ACM.

[22] Theo Härder, Michael Peter Haustein, Christian Mathis, and Markus Wagner. Node Labeling Schemes for Dynamic XML Documents Reconsidered. *Elsevier Journal of Data & Knowledge Engineering*, 60(1):126–149, January 2007.

[23] Theo Härder, Christian Mathis, Sebastian Bächle, Karsten Schmidt, and Andreas M. Weiner. Essential Performance Drivers in Native XML DBMSs. In *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'10)*, volume 5901 of *LNCS*, pages 29–46, Czech Republic, 23-29 January 2010. Springer-Verlag.

[24] Michael Peter Haustein and Theo Härder. Optimizing Lock Protocols for Native XML Processing. *Elsevier Journal of Data & Knowledge Engineering*, 65(1):147–173, April 2008.

[25] Michael Peter Haustein, Theo Härder, Christian Mathis, and Markus Wagner. DeweyIDs - The Key to Fine-Grained Management of XML Documents. *Journal of Information and Data Management*, 1(1):147–160, February 2010.

[26] Brian Hayes. Third Base. *American Scientist*, 89(6):490–494, November-December 2001.

[27] Michael Kay. *XPath 2.0 Programmer's Reference.* Wiley, 2004.

[28] Michael Kay. Ten Reasons Why Saxon XQuery is Fast. *IEEE Data Engineering Bulletin*, 31(4):65–74, December 2008.

[29] Dao Dinh Kha, Masatoshi Yoshikawa, and Shunsuke Uemura. An XML Indexing Structure with Relative Region Coordinate. In *Proceedings of the 17th International Conference on Data Engineering (ICDE'01)*, pages 313–320, Heidelberg, Germany, 02-06 April 2001. IEEE Computer Society.

[30] Aye Aye Khaing and Ni Lar Thein. A Persistent Labeling Scheme for Dynamic Ordered XML Trees. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence (WI'06)*, pages 498–501, Hong Kong, China, 18-22 December 2006. IEEE Computer Society.

[31] Hye-Kyeong Ko and SangKeun Lee. An Efficient Scheme to Completely Avoid Re-labeling in XML Updates. In *Proceedings of the 7th International Conference on Web Information Systems Engineering (WISE'06)*, volume 4255 of *LNCS*, pages 259–264, Wuhan, China, 23-26 October 2006. Springer-Verlag.

[32] Hye-Kyeong Ko and SangKeun Lee. A Binary String Approach for Updates in Dynamic Ordered XML Data. *IEEE Transactions on Knowledge and Data Engineering*, 22(4):602–607, April 2010.

[33] Bei Li, Katsuya Kawaguchi, Tatsuo Tsuji, and Ken Higuchi. A Labeling Scheme for Dynamic XML Trees Based on History-offset Encoding. In *Information Processing Society of Japan (IPSJ) Online Transactions*, volume 3, pages 71–87, 31 March 2010.

[34] Changqing Li and Tok Wang Ling. An Improved Prefix Labeling Scheme: A Binary String Approach for Dynamic Ordered XML. In *Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA'05)*, volume 3453 of *LNCS*, pages 125–137, Beijing, China, 17-20 April 2005. Springer-Verlag.

[35] Changqing Li and Tok Wang Ling. QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM'05)*, pages 501–508, Bremen, Germany, 31 October - 05 November 2005. ACM.

[36] Changqing Li, Tok Wang Ling, and Min Hu. Efficient Processing of Updates in Dynamic XML Data. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, pages 01–13, Georgia, USA, 03-07 April 2006. IEEE Computer Society.

[37] Changqing Li, Tok Wang Ling, and Min Hu. Reuse or Never Reuse the Deleted Labels in XML Query Processing Based on Labeling Schemes. In *Proceedings of the 11th International Conference on Database Systems for Advanced Appli-*

cations (DASFAA'06), volume 3882 of *LNCS*, pages 659–673, Singapore, 12-15 April 2006. Springer-Verlag.

[38] Changqing Li, Tok Wang Ling, and Min Hu. Efficient Updates in Dynamic XML Data: from Binary String to Quaternary String. *The VLDB Journal*, 17(3):573–601, 01 May 2008.

[39] Changqing Li, Tok Wang Ling, Jiaheng Lu, and Tian Yu. On Reducing Redundancy and Improving Efficiency of XML Labeling Schemes. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM'05)*, pages 225–226, Bremen, Germany, 31 October - 05 November 2005. ACM.

[40] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 361–370, Rome, Italy, 11-14 September 2001. Morgan Kaufmann.

[41] Jian Liu, Z.M. Ma, and Li Yan. Efficient Labeling Scheme for Dynamic XML Trees. *Information Sciences*, 221(0):338–354, 01 February 2013.

[42] Jun-Ki Min, Jihyun Lee, and Chin-Wan Chung. An Efficient Encoding and Labeling for Dynamic XML Data. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07)*, volume 4443 of *LNCS*, pages 715–726, Bangkok, Thailand, 09-12 April 2007. Springer-Verlag.

[43] Jun-Ki Min, Jihyun Lee, and Chin-Wan Chung. An Efficient XML Encoding and Labeling Method for Query Processing and Updating on Dynamic XML Data. *Journal of Systems and Software*, 82(3):503–515, March 2009.

[44] Meghdad Mirabi, Hamidah Ibrahim, Nur Izura Udzir, and Ali Mamat. Label Size Increment of Bit String Based Labeling Scheme in Dynamic XML Updating. In *Proceedings of the International Conference on Digital Enterprise and*

Information Systems (DEIS'11), volume 194 of *CCIS*, pages 466–477, London, UK, 20-22 July 2011. Springer-Verlag.

[45] Martin F. O'Connor. Level-based Indexing for Optimising XML Queries. Master's thesis, School of Computing, Dublin City University, Ireland, June 2005.

[46] Martin F. O'Connor and Mark Roantree. Desirable Properties for XML Update Mechanisms. In *Proceedings of the 2010 EDBT/ICDT Workshops (EDBT'10)*, number 23, Lausanne, Switzerland, 22 March 2010. ACM.

[47] Martin F. O'Connor and Mark Roantree. EBSL: Supporting Deleted Node Label Reuse in XML. In *Proceedings of the 7th International XML Database Symposium (XSym'10)*, volume 6309 of *LNCS*, pages 73–87, Singapore, 17 September 2010. Springer-Verlag.

[48] Martin F. O'Connor and Mark Roantree. SCOOTER: A Compact and Scalable Dynamic Labeling Scheme for XML Updates. In *Proceedings of the 23rd International Conference on Database and Expert Systems Applications (DEXA'12) Part I*, volume 7446 of *LNCS*, pages 26–40, Vienna, Austria, 03-06 September 2012. Springer-Verlag.

[49] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 903–908, Paris, France, 13-18 June 2004. ACM.

[50] Benoît Rittaud. On the Average Growth of Random Fibonacci Sequences. *Journal of Integer Sequences*, 10(2):32, 2007.

[51] Virginie Sans and Dominique Laurent. Prefix based Numbering Schemes for XML: Techniques, Applications and Performances. *Proceedings of the VLDB Endowment (PVLDB'08)*, 1(2):1564–1573, 2008.

[52] Haw Su-Cheng and Lee Chien-Sing. Node Labeling Schemes in XML Query Optimization: A Survey and Trends. *IETE Technical Review*, 26(2):88–100, 2009.

[53] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'01)*, pages 413–424, California, USA, 21-24 May 2001. ACM.

[54] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and Querying Ordered XML using a Relational Database System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, pages 204–215, Wisconsin, USA, 02-06 June 2002. ACM.

[55] Risi Thonangi. A Concise Labeling Scheme for XML Data. In *Proceedings of the 13th International Conference on Management of Data (COMAD'06)*, pages 04–14, Delhi, India, 14-16 December 2006. Computer Society of India.

[56] Wolfram||Alpha. *Fibonacci Numbers*, Wolfram Alpha LLC edition, December 2012. Available from: `http://mathworld.wolfram.com/FibonacciNumber.html` [Accessed 06 January 2013].

[57] WolframAlpha. *Zeckendorf Representation*, Wolfram Alpha LLC edition, December 2012. Available from: `http://mathworld.wolfram.com/ZeckendorfRepresentation.html` [Accessed 06 January 2013].

[58] World Wide Web Consortium. *Document Type Declaration in Extensible Markup Language (XML) 1.0 (Fifth Edition)*, W3C Recommendation edition, November 2008. Available from: `http://www.w3.org/TR/2008/REC-xml-20081126/#dt-doctype` [Accessed 06 January 2013].

[59] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, W3C Recommendation edition, November 2008. Available from: `http://www.w3.org/TR/2008/REC-xml-20081126/` [Accessed 06 January 2013].

[60] World Wide Web Consortium. *XML Path Language (XPath) 2.0 (Second Edition)*, W3C Recommendation edition, December 2010. Available from:

http://www.w3.org/TR/2010/REC-xpath20-20101214/ [Accessed 06 January 2013].

[61] World Wide Web Consortium. *XQuery 1.0: An XML Query Language (Second Edition)*, W3C Recommendation edition, December 2010. Available from: http://www.w3.org/TR/2010/REC-xquery-20101214/ [Accessed 06 January 2013].

[62] World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)*, W3C Recommendation edition, December 2010. Available from: http://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/ [Accessed 06 January 2013].

[63] World Wide Web Consortium. *XQuery Update Facility 1.0*, W3C Recommendation edition, March 2011. Available from: http://www.w3.org/TR/2011/REC-xquery-update-10-20110317/ [Accessed 06 January 2013].

[64] World Wide Web Consortium. *XML Schema Definition Language 1.1(XSD)*, W3C Recommendation edition, April 2012. Available from: http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/ [Accessed 06 January 2013].

[65] Xiaodong Wu, Mong-Li Lee, and Wynne Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*, pages 66–78, Massachusetts, USA, 30 March - 02 April 2004. IEEE Computer Society.

[66] Guangming Xing and Bill Tseng. Extendible Range-Based Numbering Scheme for XML Document. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) (Volume 2)*, pages 140–141, Nevada, USA, 05-07 April 2004. IEEE Computer Society.

[67] Liang Xu, Zhifeng Bao, and Tok Wang Ling. A Dynamic Labeling Scheme Using Vectors. In *Proceedings of the 18th International Conference on Database and Expert Systems Applications (DEXA'07)*, volume 4653 of *LNCS*, pages 130–140, Regensburg, Germany, 03-07 September 2007. Springer-Verlag.

[68] Liang Xu, Tok Wang Ling, Huayu Wu, and Zhifeng Bao. DDE: From Dewey to a Fully Dynamic XML Labeling Scheme. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*, pages 719–730, Rhode Island, USA, 29 June - 02 July 2009. ACM.

[69] F. Yergeau. *UTF-8, A Transformation Format of ISO 10646*, Request for Comments (RFC) 3629 edition, November 2003. Available from: `http://tools.ietf.org/html/rfc3629` [Accessed 06 January 2013].

[70] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel: A Path-based Approach to Storage and Retrieval of XML Documents using Relational Databases. *ACM Transactions on Internet Technology (TOIT)*, 1(1):110–141, August 2001.

[71] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'01)*, pages 425–436, California, USA, 21-24 May 2001. ACM.