Scalable XML change detection using relational databases

Erwin Leonardi

2007

Erwin Leonardi. (2007). Scalable XML change detection using relational databases. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/35727>

<https://doi.org/10.32657/10356/35727>

159
7529617
4X
D/Blue

SCALABLE XML CHANGE DETECTION USING RELATIONAL DATABASES

Erwin Leonardi

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Doctor of Philosophy

2007

© Copyright 2007
by
Erwin Leonardi
All Right Reserved

Ad Maiorem Dei Gloriam

*Dedicated to my God,
and
my mother, Liong Lie Tju.*

Abstract

The eXtensible Markup Language (XML) has recently emerged as a new standard for data representation and exchange on the Web. As a new standard Web document for data representation and exchange, XML document will also be changed when the information on the Web is changed. The fast and unpredictable changes on this information have created a new challenging problem of detecting and representing these changes. A number of approaches have been proposed to address this challenging problem. However, these previous work in change detection to XML documents is not suitable for detecting the changes to large XML documents as it requires a lot of memory to keep the two versions of XML documents in the memory. In this dissertation, we take a more conservative yet novel approach of using traditional relational database engines for detecting the changes to large XML documents. We store two versions of an XML document in the relational database and detect the changes by issuing SQL queries (whenever appropriate). We propose XANDY and HELIOS that use the *schema-oblivious* approach, namely, SUCXENT and the *schema-conscious* approach (in particular, Shared-Inlining approach), respectively, for storing XML documents before the changes are detected. We also proposed a novel approach for detecting the changes to DTDs called DTD-DIFF. Our experimental results show the followings. First, our relational-based approaches have better scalability compared to the published memory-based approaches. Second, our approaches for detecting the changes to *unordered* XML documents performs better for large XML documents (up to 100 times faster) compared to X-Diff, a published algorithm that addresses the problem of detecting the changes to unordered XML documents. The performance of our approaches for detecting the changes to *ordered* XML documents is comparable to XyDiff, an approach for detecting the changes

to ordered XML documents. Third, our approaches have superior result quality compared to X-Diff and XyDiff.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor, Assoc Prof Sourav Bhowmick for his support. He has introduced me to interesting topics in the XML and database fields. As an advisor, he demonstrates knowledge and patience that I need as a research student. He spends time to discuss several research issues, and make suggestions in my research directions. Thank also to Assoc Prof Sanjay Madria from University of Missouri-Rolla (USA) for his help.

My mother gave me incredible support throughout my candidature period. I could not have come this far without her constant love, support and sacrifices. I would also like to thank to my sisters, Veradia and Yunita, who have encouraged and prayed for me.

Besides, I would also like to thank Mr. Lai Chee Keong, Ms. Tan Lay Choo, and Mr. Chua Chiew Song who are the administrators in Centre for Advanced Information Systems (CAIS) for their assistance in providing administrative and technical support. Thanks also to Zhao Qiankun, Chen Ling, and Sandeep Prakash for the enlightening discussions I had with them. I would also like to extend my gratitude to all my friends in Catholic Students' Apostolate (NTU CSA), especially to Oka, Hendri, Iwan, Maggy, Ria, Esther, Vanny, Marcel, Adrianto, Anne, Yudi, and Shirvy for their encouragements and prayers.

My appreciation is for all those people who helped out along the way in one way or another. Finally, I would like to thank the School of Computer Engineering of Nanyang Technological University, Singapore for the generous resources and financial support provided for the duration of the course.

Thank you all.

Contents

Abstract	iii
Acknowledgements	v
List of Figures	x
1 Introduction	1
1.1 XML Data Model	2
1.2 Types of XML Documents	3
1.3 XDeltas	3
1.4 Motivations	13
1.5 Contributions of the Dissertation	16
1.6 Dissertation Organization	17
2 Literature Survey	19
2.1 Change Management of Tree Structured Data	19
2.2 Change Management of HTML Documents	21
2.3 Change Management of XML Documents	25
2.4 Storing XML in RDMS	32
2.5 Summary	39
3 Background	41
3.1 Background	41
3.2 Overview	42
3.3 Terminology	43
3.4 Schema for Storing XML Documents	52

4	Detecting the Changes to Ordered XML	59
4.1	The Relational Schema Extensions	60
4.2	Finding Best Matching Subtrees Phase in XANDY-O	63
4.3	Finding Best Matching Subtrees Phase in HELIOS-O	79
4.4	Detecting the Changes Phase in XANDY-O	88
4.5	Detecting the Changes Phase in HELIOS-O	104
4.6	Summary	117
5	Performance Study of Xandy-O and Helios-O	119
5.1	Experimental Settings and Data Sets	120
5.2	Different RDBMS	122
5.3	Numbers of Nodes	125
5.4	Phase 1 vs Phase 2	131
5.5	File Size vs Execution Time	133
5.6	Percentage of Changes	134
5.7	Execution Time Proportion of Different Types of Changes	137
5.8	Different Types of Changes	141
5.9	Depths of XML Documents	143
5.10	Similarity Score Threshold	145
5.11	Result Quality	148
5.12	Summary	150
6	Detecting the Changes to Unordered XML	153
6.1	The Relational Schema Extensions	153
6.2	Finding Best Matching Subtrees Phase in XANDY-U	155
6.3	Finding Best Matching Subtrees Phase in HELIOS-U	161
6.4	Detecting the Changes Phase in XANDY-U	165
6.5	Detecting the Changes Phase in HELIOS-U	168
6.6	Summary	174

7	Performance Study of Xandy-U and Helios-U	175
7.1	Experimental Settings and Data Sets	176
7.2	Numbers of Nodes	176
7.3	Phase 1 vs Phase 2	180
7.4	File Size vs Execution Time	181
7.5	Percentage of Changes	183
7.6	Execution Time Proportion of Different Types of Changes	184
7.7	Different Types of Changes	186
7.8	Depths of XML Documents	187
7.9	Similarity Score Threshold	188
7.10	Result Quality	190
7.11	Summary	192
8	Detecting the Changes to Hybrid XML	195
8.1	Detecting the Changes to Hybrid XML	195
8.2	Detecting the Changes to DTDs	208
8.3	Summary	232
9	Conclusions and Future Work	233
9.1	Contributions	233
9.2	Conclusions	234
9.3	Future Works	236
	References	237
	Appendices	250
A	List of Publications	250

List of Figures

1.1	Samples of XML document and DOM Tree	2
1.2	<i>Ordered</i> XML Documents	4
1.3	Representation Trees of <i>Ordered</i> XML Documents	5
1.4	Example of Interpretation of Deletions.	7
1.5	Example Semantically Incorrect for Name Updates.	8
1.6	XDelta Examples.	8
1.7	<i>Unordered</i> XML Documents	9
1.8	Representation Trees of <i>Unordered</i> XML Documents	10
1.9	<i>Hybrid</i> XML Documents	11
1.10	Representation Trees of <i>Hybrid</i> XML Documents	12
2.1	MH-DIFF: Examples.	20
2.2	MH-DIFF: Induced Graph.	21
2.3	MH-DIFF: Annotated Pruned Induced Graph.	22
2.4	Comparisons of Approaches for Detecting the Changes to HTML Documents.	24
2.5	Comparisons of Approaches for Detecting the Changes to XML Documents.	25
2.6	DeltaXML: The Presence of DTD.	30
2.7	DIFFXML: XML in Relational Database.	31
2.8	DIFFXML: Move Ratio Threshold.	31
2.9	DIFFXML: XDelta.	33
2.10	Comparisons of Approaches for Storing XML Documents in RDBMS.	34
2.11	The SUCXENT Schema.	38
2.12	The SUCXENT++ Schema.	38

3.1 Two Examples of XDeltas. 42

3.2 SUCXENT: Example. 54

3.3 Shared-Inlining: DTD Transformation. 55

3.4 Shared-Inlining: DTD Graph. 56

3.5 The Basic Inlining Schema. 56

3.6 The Shared and Hybrid Inlining Schema. 57

4.1 Modified SUCXENT Schema for XANDY-O. 60

4.2 Top-down Approach: XML in RDBMS. 60

4.3 Bottom-up Approach: XML in RDBMS. 61

4.4 DTD Tree, DTD, and Relational Schema. 62

4.5 HELIOS-O: XML in RDBMS. 63

4.6 Top-down Approach: The *rootMatching* Algorithm. 64

4.7 Top-down Approach: The *findBestMatchingSubtrees* Algorithm. 65

4.8 Top-down Approach: SQL Queries (1). 67

4.9 Intermediate Result of SQL Query in Figure 4.8(b) 68

4.10 Top-down Approach: SQL Query for Mapping Identical Subtrees. 69

4.11 Bottom-up Approach: The *findBestMatchingSubtrees* Algorithm. 70

4.12 Bottom-up Approach: The *findMatchingSiblingOrder* Algorithm. 71

4.13 Bottom-up Approach: SQL Queries for Finding Matching Sibling Orders. . . 71

4.14 Intermediate Result of SQL Query in Figure 4.13(b). 73

4.15 Bottom-up Approach: Example of Finding Matching Sibling Orders. 74

4.16 Table Descriptions. 74

4.17 The TempS0 table and SQL Query for Finding Matching Internal Nodes. . . . 75

4.18 Intermediate Results of SQL Query Depicted in Figure 4.17. 76

4.19 The TempMatching table. 77

4.20 Example: Maximizing Similarity Score. 78

4.21 Bottom-up Approach: The MATCHING Table. 78

4.22 HELIOS-O: The *findBestMatchingSubtrees* Algorithm. 79

4.23 HELIOS-O: The *findMatchingLeafNodesGroups* Algorithm. 79

4.24 HELIOS-O: SQL Queries for Finding Matching Leaf Nodes. 80

4.25 Intermediate Result of SQL Query in Figure 4.24(c). 82

4.26 HELIOS-O: Temporary Matching Tables and the Matching Table. 83

4.27 HELIOS-O: Temporary and Delta Table Descriptions. 83

4.28 HELIOS-O: SQL Queries for Finding Matching Internal Nodes. 84

4.29 Intermediate Result of SQL Query in Figure 4.28(a). 85

4.30 HELIOS-O: Example. 86

4.31 Intermediate Result of SQL Query in Figure 4.28(b). 87

4.32 HELIOS-O: SQL Queries for Collecting Best Matching Internal Nodes. 88

4.33 XANDY-O: The *changeDetector* Algorithm. 89

4.34 SQL Queries: Insertion of Internal and Leaf Nodes. 90

4.35 Delta Tables Attributes and Attribute Descriptions. 91

4.36 Delta Tables: Insertion, Deletion, and Updates. 91

4.37 Intermediate Result of SQL query in Figure 4.34(c). 92

4.38 SQL Queries: Updated Leaf Nodes. 94

4.39 Intermediate Result of SQL query in Figure 4.38(a). 95

4.40 Examples of Relative Updates Misdetection. 96

4.41 The *updateCorrector* Algorithm and SQL Queries. 97

4.42 Intermediate Results of SQL Query in Figure 4.41(c). 98

4.43 Move Among Siblings: Example, Algorithm, and SQL Queries. 99

4.44 Intermediate Result of SQL Query in Figure 4.43(c). 100

4.45 The `moveList` Table. 101

4.46 Effects of Insertion/Deletion to Local Order. 102

4.47 Example: Move To Different Parent Nodes. 102

4.48 SQL Queries: Move To Different Parent Nodes. 103

4.49 HELIOS-O: The *changeDetector* Algorithm. 105

4.50 HELIOS-O: SQL Queries for Detecting Inserted Internal Nodes. 107

4.51 HELIOS-O: Delta Tables. 108

4.52 HELIOS-O: Table Descriptions. 109

4.53 HELIOS-O: SQL Queries for Detecting Inserted Leaf Nodes. 110

4.54 HELIOS-O: SQL Queries for Detecting Updated Inlined Leaf Nodes. 113

4.55 HELIOS-O: Intermediate Result of SQL Query in Figure 4.54(a). 114

4.56 HELIOS-O: SQL Queries for Detecting Updated Non-Inlined Leaf Nodes. . . 115

4.57 HELIOS-O: SQL Queries for Detecting Moved Nodes. 116

5.1 Data Sets. 120

5.2 Experimental Results: $\mathcal{RDB}_A$ vs $\mathcal{RDB}_B$ (HELIOS-O). 121

5.3 Query Execution Plans for SQL Query in Figure 4.24(c). 122

5.4 Query Execution Plans for SQL Query in Figure 4.50(a). 123

5.5 Experimental Results: $\mathcal{RDB}_A$ vs $\mathcal{RDB}_B$ (XANDY-O). 124

5.6 Execution Time vs Number of Nodes (Log scale). 126

5.7 Execution Time vs Number of Nodes (Log scale, 1GB RAM). 127

5.8 Performance Gain. 129

5.9 TCSD: Execution Time vs Number of Nodes (Log scale). 130

5.10 Experimental Results: Phase 1 and Phase 2 Performance Comparison. . . . 131

5.11 Experimental Results: Query Execution Plan Comparison. 132

5.12 Execution Time vs File Size. 134

5.13 Experimental Results: Execution Time vs Percentage of Changes (SIGMOD-03). 135

5.14 Experimental Results: Execution Time vs Percentage of Changes (SIGMOD-05). 136

5.15 Query Execution Plan. 137

5.16 HELIOS-O: Execution Time Proportion for Different Types of Changes. . . . 138

5.17 XANDY-O: Execution Time Contribution of Different Types of Changes. . . . 140

5.18 HELIOS-O: Different Types of Changes. 141

5.19 XANDY-O: Different Types of Changes. 143

5.20 Experimental Results: Depths. 145

5.21 Similarity Score Threshold θ vs Execution Time. 146

5.22 Similarity Score Threshold θ vs Result Quality. 147

5.23 Experimental Results: Result Quality. 148

5.24 Result Quality: Examples (1). 148

5.25 Result Quality: Examples (2). 149

6.1 XANDY-U: Modified SUCXENT Schema. 154

6.2 XANDY-U: XML in RDBMS. 154

6.3 HELIOS-U: DTD and Modified Schema. 155

6.4 HELIOS-U: XML in RDBMS. 156

6.5 XANDY-U: The *findMatchingSiblingOrder* algorithm, An Example, and SQL
Queries. 157

6.6 XANDY-U: Table Descriptions. 157

6.7 XANDY-U: SQL Queries Used in The *refineMatchingLV* algorithm. 158

6.8 XANDY-U: Intermediate Result of SQL Query in Figure 6.7(b). 159

6.9 XANDY-U: The TempLV and TempS0 Tables. 160

6.10 XANDY-U: The TempMatching Tables. 161

6.11 XANDY-U: SQL Query and The MATCHING Table. 162

6.12 HELIOS-U: Finding Matching Leaf Nodes Groups. 162

6.13 HELIOS-U: The M_Author, M_Article, M_Issue, and MATCHING Tables. 164

6.14 XANDY-U: The *changeDetector* Algorithm. 165

6.15 XANDY-U: The Delta Tables. 166

6.16 HELIOS-U: The *changeDetector* Algorithm. 169

6.17 HELIOS-U: Delta Tables. 170

7.1 DCSD: Execution Time vs Number of Nodes (Log scale). 177

7.2 DCSD: Execution Time vs Number of Nodes (Log scale, 1GB RAM). 178

7.3 Performance Gain. 179

7.4 TCSD: Execution Time vs Number of Nodes (Log scale). 180

7.5 HELIOS-U and XANDY-U: Phase 1 vs Phase 2. 181

7.6 File Size vs Execution Time. 182

7.7 Percentage of Changes vs Execution Time. 183

7.8 HELIOS-U: Execution Time Proportion for Different Types of Changes. . . . 184

7.9 XANDY-U: Execution Time Proportion for Different Types of Changes. . . . 185

7.10 HELIOS-U: Different Types of Changes. 186

7.11 XANDY-U: Different Types of Changes. 187

7.12 Depth vs Execution Time. 188

7.13 Similarity Score Threshold θ vs Execution Time. 189

7.14 Result Quality vs Similarity Score Threshold. 190

7.15 Result Quality. 191

8.1 XANDY-H: The *findNodeType* Algorithm. 196

8.2 XANDY-H: The **nodeType** Table. 196

8.3 XANDY-H: The *findMatchingSiblingOrder* Algorithm and SQL Queries for
Finding Matching Leaf Nodes. 198

8.4 XANDY-H: The *detectTheChanges* Algorithm. 200

8.5 XANDY-H: The Delta Tables. 201

8.6 XANDY-H: SQL for Detecting Updated Leaf Nodes. 202

8.7 XANDY-H: Data Sets. 204

8.8 XANDY-H: Experimental Results (1). 206

8.9 XANDY-H: Experimental Results (2). 207

8.10 XANDY-H: Experimental Results (3). 208

8.11 Two versions of a DTD. 209

8.12 The Corresponding XSD Files of DTDs in Figure 8.11. 210

8.13 X-Diff and XyDiff results. 212

8.14 Content Trees and Examples. 215

8.15 The DTD-DIFF Algorithm. 222

8.16 The *CalculateHashValue* Algorithm. 222

8.17 The *Matching* Algorithm. 223

8.18 The <i>ComputeCost</i> Algorithm.	224
8.19 The <i>detectAttributeChanges</i> Algorithm.	225
8.20 Data Sets.	226
8.21 Experimental Results (1).	228
8.22 Experimental Results: Real Data Sets.	230
8.23 Experimental Results (2).	231
8.24 Result quality.	232

Chapter 1

Introduction

The eXtensible Markup Language (XML) [Conb] has recently emerged as a new standard for representing and exchanging data on the Internet. Many applications have adopted XML for exchanging information on the Web. XML is designed to simplify information exchange between applications. It is a subset of SGML (Standardized General Markup Language) [Conc] designed to provide structure to textual document and augments HTML (Hypertext Markup Language) by allowing data to carry its meaning and not just presentation details.

XML was designed to describe data and to focus on what the data are [W3S]. XML contains tags and data. Tags in XML are not predefined. XML allows users to make up any new tags for descriptive markups for their own applications. Such user-defined tags on data elements allow users to identify the semantics of data. This earns XML the name as portable data [Ser], write once use anywhere. The author only needs to mark up his data to XML once and the data could be used and re-used by many sources/applications. The interpretation of the XML data will depend on the receivers. Figures 1.1(a) and 1.1(b) depict two different ways to represent the same information in XML. Since XML data is self describing [W3S], it is considered as one of the most promising means to define semi-structured data, which is expected to be ubiquitous in large volumes from diverse data sources and applications on the Web.

As a new standard Web document for representing and exchanging data, XML documents will also be changed when the information on the Web is changed. These changes typically take two general forms. The first is existence. XML documents exhibit varied longevity patterns. The second is structure and content modifications. An XML document replaces its antecedents, usually leaving no trace of the previous XML documents. The fast and unpredictable changes on this information have created a new challenging problem of

Figure 1.1: Samples of XML document and DOM Tree

detecting and representing these changes (hereafter called *XML deltas* or *XDeltas*). This is a challenging problem because the information sources in the Web are autonomous and typical database approaches to detect changes based on triggering mechanisms are not usable [BMN03]. Such a change detection tool is important to incremental query evaluation, trigger condition evaluation, search engine, data mining applications, and mobile applications [CAM02, WDyC03]. In this dissertation, we present novel techniques to detect the changes to XML documents. We take a more conservative yet novel approach of using traditional relational database engines for detecting XML deltas.

We begin by presenting the data model of XML that is used in this dissertation in Section 1.1. Based on this XML data model, XML documents can be classified into different types. We shall elaborate on different types of XML documents in Section 1.2. Each type of XML documents has its own types of changes. We discuss different types of changes for different types of XML documents in Section 1.3. In Section 1.4, we present the motivation of our research. In Section 1.5, we summarize the contributions of this dissertation. Finally, Section 1.6 outlines the organization of the rest of this dissertation.

1.1 XML Data Model

Even though the underlying challenge is how to detect and represent the changes to large volume of data, the novel context of the XML forces us to significantly extend traditional techniques. XML data is commonly modeled as a tree structure (hereafter called *XML tree*) based on Document Object Model (DOM) [Cona]. In XML tree, there are three types of nodes as following.

- **Element node** - Element nodes is a non-leaf node with one *name*.
- **Text node** - Text node is a leaf node which has one *value*.

- **Attribute node** - Attribute node is a leaf node which has a *name* and a *value*.

For example, we have an XML document X_3 as shown in Figure 1.1(c) and its DOM tree representation T_{X_3} which is depicted in Figure 1.1(d). Element nodes are represented as boxes, such as “Catalog” and “CD”. The text nodes are represented as ellipses, such as “Fallen” and “John Mayer”. The attribute nodes are represented as dotted rectangles, such as “SKU” with value “001”. Note that we use these notations throughout the dissertation.

1.2 Types of XML Documents

According to the DOM specification, element nodes and text nodes are *ordered*, while attribute nodes are *unordered*. Hence, we classify XML documents into three types as following.

- **Ordered XML.** An ordered XML document is the one in which both ancestor-descendent relationships and left-to-right orders among siblings are significant.
- **Unordered XML.** An unordered XML is the one in which the ancestor-descendent relationship is significant, while the left-to-right order among siblings is not important.
- **Hybrid XML.** Our analysis of different real life XML documents has revealed that XML documents may not always be *purely ordered* or *purely unordered*. It is indeed possible to have both ordered and unordered nodes in the *same* XML document. We call such documents as *hybrid XML* [LBB05].

We observe that each type of these XML documents has its own characteristics. When an XML document is changed, the changes follow the characteristics of XML type that it belongs to. In other words, the types of changes of different types of XML documents are different. In the next section, we shall present the types of changes for each type of XML documents.

1.3 XDeltas

The types of changes in XML can be classified into *content changes* and *structural changes*. The structural changes are ones that occur in the internal elements and modify the structure, but do not change textual data content. The content changes are ones that occur in the leaf

Figure 1.2: Ordered XML Documents

elements and modify the textual data content. In this section, we shall discuss the types of changes for each type of XML documents. We also present the motivating examples to facilitate the discussion.

1.3.1 Ordered XML

Suppose we have two versions of an XML document, D_1 and D_2 , as depicted in Figures 1.2(a) and 1.2(b), respectively. The tree representations of D_1 (denoted as T_1) and D_2 (denoted as T_2) are depicted in Figures 1.3(a) and 1.3(b), respectively. These XML documents store the information on staff in a university. The positions of the staff in the documents are sorted according to their ranks. These XML documents can be classified as *ordered XML*.

In our example, T_1 has evolved to T_2 . We notice that “Smith” is not a faculty member of the university anymore. Hence, subtree rooted at node 2 (t_2) in T_1 is deleted. “Chan” does not focus his research in data mining area anymore. Consequently, node 12 in T_1 is deleted. “Mark” gets promoted to a higher rank. Thus, the value of node 16 has been updated from “Assoc Prof” to “Prof”. He also extends his research interests to information retrieval area.

Figure 1.3: Representation Trees of *Ordered* XML Documents

Hence, node 107 in T_2 is inserted. We notice that subtree rooted at node 113 (t_{113}) in T_2 that stores the information about “Steve” is inserted. This means that the university has a new faculty member whose name is Steve. Recall that the position of staff in the XML document is sorted according to their ranks. We observe that the subtree t_{14} that stores information about “Mark” who just gets promoted is moved among its siblings. From this example, we are able to summarize the types of changes in an ordered XML as following. The types of changes for internal nodes are as follows.

- **Insertion of Internal Node.** A new internal node i whose node name is “name” is inserted into the XML tree as the k -th child of parent node p . Observe that we need to specify the position of an inserted internal node among its siblings as the left-to-right position among siblings is important in ordered XML. An insertion of a new internal node i can be seen as an insertion of a new subtree. This is because an internal node must have at least one child node. That is, an insertion of an internal node i must be followed by insertions other nodes in the subtree rooted at node i . For example,

consider subtree t_{113} in T_2 as depicted in Figure 1.3(b). The insertion of internal node 113 is followed by insertions of nodes in subtree t_{113} .

- **Deletion of Internal Node.** An internal node i is deleted from an XML tree. Similar to the insertion of internal nodes, a deletion of an internal node i must also be followed by deletions of other nodes in the subtree rooted at node i . For example, consider subtree t_2 in T_1 as depicted in Figure 1.3(a). The deletion of internal node 2 is followed by deletions of nodes in subtree t_2 .
- **Move to Different Parent Node.** An internal node i whose parent node in the old version is p is moved to be the child node of node q in the new version, where $p \neq q$.
- **Move among Siblings.** An internal node i that is the k -th child of node p in the old version is moved to be the j -th child of node p in the new version, where $k \neq j$. For example, consider node 14 in T_1 and node 102 in T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. Observe that nodes 14 and 102 are *matching node*. Node 14 that is the third child of node 1 is moved to be the first child of node 101.

There are two interpretations on the edit operations, that is, Selkow's model [Sel77] and Kuo-Chung Tai's model [Tai79]. In Tai's model [Tai79], deletion of a node means making its child nodes become the child nodes of its parent node. For instance, deletion of node "research" in the XML document depicted in Figure 1.4(a) yields the XML document as shown in Figure 1.4(b). This model may not be appropriate for XML documents, since deleting a node changes its depth in the tree and may also invalidate the document structure according to its DTD (or XML Schema) [Cob03]. Thus, for XML data, we use Selkow's model [Sel77] in which operations are only applied to leaves or subtrees. In particular, when a node is deleted, the entire subtree rooted at the node is deleted (Figure 1.4(c)). This captures the XML semantic better, for instance removing the information of a staff from a university by deleting the corresponding subtree [Cob03].

The types of changes for leaf nodes are as follows.

- **Insertion of Leaf Node.** A leaf node ℓ whose node name and node value are "*name*" and "*value*", respectively, is inserted into the XML tree as the k -th child of parent node p . For example, leaf node 107 in T_2 whose node name and node value are "*interest*" and "*Information Retrieval*", respectively, is an inserted leaf node as the second child node of node 105.

CHAPTER 1. INTRODUCTION

Figure 1.4: Example of Interpretation of Deletions.

- **Deletion of Leaf Node.** A leaf node ℓ is deleted from the XML tree. For example, leaf node 12 in T_1 whose node name and node value are “*interest*” and “*Data Mining*”, respectively, is deleted from T_1 .
- **Content Update of Leaf Node.** The node value of a leaf node ℓ is updated from “*old value*” to “*new value*”. Consider node 16 in T_1 . The node value of node 16 is updated from “*Assoc Prof*” to “*Prof*” (node 104 in T_2).
- **Move to Different Parent Node.** A leaf node ℓ whose parent node in the old version is p is moved to be the child node of node q in the new version, where $p \neq q$.
- **Move among Siblings.** A leaf node ℓ that is the k -th child of node p in the old version is moved to be the j -th child of node p in the new version, where $k \neq j$.

The types of changes for attribute nodes are as follows.

- **Insertion of Attribute Node.** An attribute node a whose attribute name and attribute value are “*name*” and “*value*”, respectively, is inserted as the attribute node of node n . Note that the attribute nodes are unordered. Hence, we do not need to specify the left-to-right order among its siblings.
- **Deletion of Attribute Node.** An attribute node a is deleted from being the attribute node of node n .
- **Content Update of Attribute Node.** Similar to the leaf nodes, the value of an attribute a can be changed/updated from “*old value*” to “*new value*”.

Note that we do not consider the updates of node name because it may lead to *semantically incorrect deltas*. Let us elaborate further. Consider two versions of an XML document fragment as depicted in Figure 1.5. If we consider the updates of node name, then there will

Figure 1.5: Example Semantically Incorrect for Name Updates.

Figure 1.6: XDelta Examples.

be two sets of XDeltas from X_1 and X_2 as depicted in Figure 1.6. If we consider $X\Delta_1$, then we notice that the node name of node 2 is updated from “lastname” to “telp”. It will lead us to have a delta that is *semantically* incorrect. Therefore, we shall detect the $X\Delta_2$ as depicted in Figure 1.6(b). Note that the $X\Delta_2$ is a *semantically* correct one. On the other hand, suppose the node name of node 2 is updated to “surname”. We shall detect this update as a pair of deletion and insertion of nodes as we do not have information of semantic relationships between “lastname” and “surname”. Observe that the detected XDelta is still correct even though the result quality may be reduced (in some cases). The prior works [CAM02, Fon01, Fon02, CMB04, WDyC03] that address the problem of detecting the changes to XML documents also do not consider the node name updates.

1.3.2 Unordered XML

Consider two versions of an XML document, D_1 and D_2 , as depicted in Figures 1.7(a) and 1.7(b), respectively. The tree representations of D_1 and D_2 are depicted in Figures 1.8(a) and 1.8(b), respectively. Assume that these XML documents are unordered XML. We observe that T_1 has been evolved to T_2 . Similar to ordered XDeltas, we also classify the type of changes for unordered XML to *types of changes for internal nodes*, *types of changes for leaf nodes*, and *types of changes for attribute nodes*. The types of changes for internal nodes are as follows.

- **Insertion of Internal Node.** A new internal node i whose node name is “name” is inserted into an XML tree as the child node of parent node p . Observe that we do

Figure 1.7: *Unordered XML Documents*

not need to specify the position of the inserted internal node among its siblings as the left-to-right position among siblings is not important in unordered XML. An insertion of a new internal node i can be seen as an insertion of a new subtree rooted at node i because of the same reason as discussed in Section 1.3.1. For instance, subtree rooted at node 126 (t_{126}) is inserted into T_2 .

- **Deletion of Internal Node.** An internal node i is deleted from an XML tree. Similar to the insertion of internal nodes, a deletion of an internal node i must also be followed by deletions of other nodes in the subtree rooted at node i . For example,

CHAPTER 1. INTRODUCTION

Figure 1.8: Representation Trees of *Unordered* XML Documents

consider subtree t_{14} in T_1 . The deletion of internal node 14 is followed by deletions of nodes in subtree t_{14} .

The types of changes for leaf nodes are as follows.

- **Insertion of Leaf Node.** A leaf node ℓ whose node name and node value are “*name*” and “*value*”, respectively, is inserted into an XML tree as the child node of parent node p . We do not need to specify the position of an inserted leaf node among its siblings as we deal with *unordered* XML. For example, leaf node 124 in T_2 whose node name and node value are “*author*” and “*Peter*”, respectively, is an inserted leaf node and the child node of node 121.
- **Deletion of Leaf Node.** A leaf node ℓ is deleted from an XML tree. For example, leaf node 12 in T_1 whose node name and node value are “*author*” and “*Marcella*”, respectively, is deleted from T_1 .
- **Content Update of Leaf Node.** The node value of a leaf node ℓ is updated from “*old value*” to “*new value*”. Consider node 7 in T_1 . The node value of node 9 is updated from “*Web Mining*” to “*Web Usage Mining*” (node 107 in T_2).

Figure 1.9: Hybrid XML Documents

The types of changes for attribute nodes are similar to the ones discuss in Section 1.3.1. We also do not consider the updates to node name for the same reason as discussed in Section 1.3.1. The movement of nodes to different parent nodes will be detected as pairs of deletions and insertions of nodes.

1.3.3 Hybrid XML

Consider two versions of an XML document, D_1 and D_2 , as depicted in Figures 1.9(a) and 1.9(b), respectively. The tree representations of D_1 and D_2 are depicted in Figures 1.10(a) and 1.10(b), respectively. These XML documents can be classified as hybrid XML for the following reason. The child nodes of node “*chapters*” must be ordered. This reflects the condition in the real world in which the chapters in a book are ordered. We observe that the child nodes of node “*authors*” may be ordered as the order of authors of a book may be important. The child nodes of nodes “*books*” and “*book*” are not necessarily ordered as books in a book collection are not necessarily in sequence. Therefore, we can assume that these XML documents are *hybrid XML*. We assume that only nodes “*authors*” and “*chapters*” are ordered nodes. Then, the types of changes for hybrid XML are also classified into *types of changes for internal nodes*, *types of changes for leaf nodes*, and *types of changes for attribute nodes*. The types of changes for internal nodes are as follows.

Figure 1.10: Representation Trees of Hybrid XML Documents

- **Insertion of Internal Node.** If the parent node of a newly inserted internal node is an *ordered* node, then this type of changes is similar to the one discussed in Section 1.3.1. That is, we need to specify the position among siblings of the inserted internal nodes. Otherwise, it is similar to the one discussed in Section 1.3.2. For instance, subtree rooted at node 102 (t_{102}) is inserted into T_2 .
- **Deletion of Internal Node.** This type of changes is similar to the one discussed in Section 1.3.1. For example, consider subtree t_{11} in T_1 . The deletion of internal node 11 is followed by deletions of nodes in subtree t_{11} .
- **Move among Siblings.** This type of changes is similar to the one discussed in Section 1.3.1. The difference is that, in hybrid XML, only the moves among siblings of the child nodes of the *ordered* nodes is considered as changes. For instance, the positions among siblings of nodes 4 and 7 in T_1 are changed (nodes 113 and 118 in T_2). The changes to their positions among siblings are not considered as changes as their parent node (node “book”) is an unordered node.

The types of changes for leaf nodes are as follows.

- **Insertion of Leaf Node.** If the parent node of a newly inserted leaf node is an *ordered* node, then this type of changes is similar to the one discussed in Section 1.3.1.

That is, we need to specify the position among siblings of the inserted leaf nodes. Otherwise, it is similar to the one discussed in Section 1.3.2. For example, leaf node 116 in T_2 whose node name and node value are “chapter” and “Next day”, respectively, is inserted as the third child node of node 113 as its parent node (node “chapters”) is an *ordered node*.

- **Deletion of Leaf Node.** A leaf node ℓ is deleted from the XML tree.
- **Content Update of Leaf Node.** The node value of a leaf node ℓ is updated from “old value” to “new value”. Consider node 9 in T_1 . The node value of node 9 is updated from “Unexpected evening” to “Unexpected night” (node 115 in T_2).
- **Move among Siblings.** This type of changes is similar to the one of internal nodes. For instance, the positions among siblings of nodes 5 and 6 in T_1 are changed (nodes 119 and 120 in T_2). The changes to their positions among siblings are considered as changes as their parent node (node “authors”) is an *ordered node*.

The types of changes for attribute nodes are similar to the ones discuss in Section 1.3.1. We also do not consider the updates to node name for the same reason as discussed in Section 1.3.1. The movement of nodes to different parent nodes will be detected as pairs of deletions and insertions of nodes.

1.4 Motivations

In this section, we discuss the motivations behind the building of novel relational-based approaches for detecting the changes to XML documents.

1.4.1 Shortcomings of Memory-based Approaches

Recently, a number of techniques for detecting the changes to XML data have been proposed. XyDiff [CAM02] is a main-memory algorithm for detecting the changes to *ordered* XML documents. In an ordered XML, both the parent-child relationship and the left-to-right order among siblings are important. X-Diff [WDyC03] is proposed for computing the changes to *unordered* XML documents. In an unordered XML, the parent-child relationship is significant, while the left-to-right order among siblings is not important. In this section, we identify the shortcomings faced by the memory-based approaches [CAM02, WDyC03].

Scalability Problem. The memory-based approaches [CAM02, WDyC03] work as follow. First, two versions of an XML document are parsed into tree structures and stored them in the main memory. Next, the approaches apply algorithms for finding the *best matching* subtrees from two XML trees. Having determined the *best matching* subtrees, they detect the changes and generate an edit script. We observe that the memory-based approaches require the entire trees (i.e., DOM trees) of two XML documents to be memory resident. This problem is exacerbated by the fact that these trees are typically much larger (about 5-10 times) than their XML documents [TLW01, NJ03]. Therefore, the memory-based approaches are not suitable for detecting the changes to *large* XML documents and this motivates the needs to build more scalable change detection system. Observe that the information on the Web keep growing in significant rate. That is, when the information is increasing, the size of XML documents storing the information will also grow.

Multiple Times Parsing Problem. The multiple times parsing problem occurs because the memory-based approaches parse an XML document multiple times whenever we want to compare it with more than one document at different times. For example, suppose at time t_1 we have two versions of an XML document, that is, X_1 and X_2 . The memory-based approaches parse X_1 and X_2 , and detect the differences between X_1 and X_2 . Suppose at time t_2 the new version of this XML document, X_3 , is available. If we want to detect the changes between X_2 and X_3 , then we need to parse X_2 and X_3 before detecting the changes. We observe that, in our example, X_2 will be parsed more than once. That is, there will be *unnecessary* cost because of the parsing process. Note that XML parsing has been pointed out as the main bottleneck in many XML applications [NJ03].

1.4.2 Relational Databases for Storing and Querying XML Documents

Recently, a number of approaches for storing and querying XML documents using relational databases have been proposed [FK99, JLWY02a, STZ⁺99, YASU01, RFHR03, BFRS02, PBM04a, PBM04b]. These approaches can be classified into two categories, namely, the *schema-conscious* approach [STZ⁺99, RFHR03, BFRS02] and *schema-oblivious* approach [FK99, JLWY02a, YASU01, PBM04a, PBM04b]. In *schema-conscious* approach, a relational schema is created based on the DTD/schema of the XML documents. In the *schema-oblivious* approach, a fixed schema used to store XML documents is maintained. The basic idea is to capture the tree structure of an XML document. This approach does not require

existence of an XML schema/DTD. Also, number of tables is fixed in the relational schema and does not depend on the structural heterogeneity of XML documents. We shall elaborate on these approaches further in Chapter 2.

A relational database can be used in two ways to address the change detection problem. Let us elaborate on this further. Suppose source A sends an XML document D_1 (version 1) at time t_1 to source B . Source B stores D_1 in its local relational database. At time t_2 , A modifies D_1 to D_2 (version 2) and sends it to B . B can now detect the changes to the XML document in the following two ways.

- (i) B extracts D_1 from the relational database and compares it to D_2 (before inserting D_2 into the database) by using any one of the above main memory-based change detection approaches.
- (ii) B first stores D_2 in the relational database and then detects the changes to the documents by executing a set of SQL queries whenever appropriate.

In the first approach, the costs incurred are the extraction time of D_1 and the change detection time of the main memory-based algorithms. However, as mentioned earlier, these algorithms are not scalable. Furthermore, the extraction cost is incurred every time we wish to compare D_1 . The costs incurred by the second approach are the time taken to insert D_2 into the database and the change detection time in the database. In particular, by storing XML documents as tables, we can filter out tuples and attributes that are not needed. Second, the system using this approach is more scalable as it can handle very large XML documents that may not fit into the main memory. Third, by storing XML in RDBMS, we only need to parse the XML documents once and then we can find the differences by issuing SQL queries against the database. Fourth, the relational-based approaches can be built side-by-side with the existing XML data management systems that are using relational database for storing and querying XML documents. Finally, implementing a change detection algorithm in SQL makes the programming task easier. Also, as SQL is an industry standard and available on all major RDBMS's, the implementation of the change detection technique is portable. We believe that the second approach is an attractive option.

As there are two different approaches for storing XML documents in relational database, in this dissertation, we will investigate *whether we can use both schema-conscious and schema-oblivious approaches as the underlying schema for storing XML documents in the*

relational-based approaches. Note that the characteristics of *schema-conscious* approach raise certain challenges. For instance, in this approach no special relational schema needs to be designed as it can be generated on the fly based on the DTD of the XML document(s). That is, unlike *schema-oblivious* approaches, the underlying relational schema is DTD-dependent. Consequently, the challenge is to create a general framework for change detection so that the framework is independent of the structural heterogeneity of various XML documents. In other words, given a specific *schema-conscious* approach, our framework should be able to detect all changes accurately independent of the changes to the underlying relational schema due to different structure of XML documents.

The usage of relational databases for addressing the problem of detecting the changes raises some issues that must be investigated in order to make the relational-based approach make sense as following.

- How scalable are the relational-based approaches compared to the existing memory-based approaches?
- Can the relational-based approaches be comparable or even faster than the state-of-the-arts memory-based approaches?
- Can the relational-based approaches detect the changes to *ordered*, *unordered*, and *hybrid* XML documents accurately?
- Can the relational-based approaches produce comparable or even better result compared to the ones produced by the existing memory-based approaches?

As parts of this dissertation, we will investigate on the above issues.

1.5 Contributions of the Dissertation

The major contributions of this dissertation are summarized as follows.

- We show that it is indeed possible to detect the changes accurately by utilizing relational databases. It is also possible to use the *schema-oblivious* approach and the *schema-conscious* approach as the underlying relational schema for storing XML documents before the changes are detected.

- We propose XANDY (**X**ml **enA**bled **chaN**ge **D**etection **sY**stem), a relational-based approach that uses SUCXENT schema [PBM04b] as its underlying relational schema. SUCXENT is classified as a *schema-oblivious* approach. There are three variants of XANDY, namely, XANDY-U, XANDY-O, and XANDY-H that address the problem of detecting the changes to *unordered* XML, *ordered* XML, and *hybrid* XML, respectively.
- We present HELIOS (**scHE**ma-conscious **x**ml-**enabL**ed **change detectIO**n **S**ystem) that employs the schema-conscious approach, namely, Shared-Inlining [STZ⁺99], as its underlying relational schema. We describe two variants of HELIOS, namely, HELIOS-U and HELIOS-O that solve the problem of detecting the changes to *unordered* XML and *ordered* XML, respectively.
- We propose a new type of XML documents, namely, *hybrid* XML based on our analysis of different real world XML documents that they may not always be *purely ordered* or *purely unordered*. Recall that the problem of detecting the changes to *hybrid* XML documents is addressed by XANDY-H. In addition, we present DTD-DIFF, a novel algorithm for detecting the changes to DTDs. To the best of our knowledge, DTD-DIFF is the first approach that addresses the problem of detecting the changes to DTDs. Note that DTD can be represented as a set of *hybrid trees*. This is because DTDs often contain sequence (denoted by “,”) and choice (denoted by “|”) groups. The order of elements in a sequence group is important, while the order of elements in a choice group is not significant.
- An extensive performance study was conducted on our approaches. The experimental results show that the relational-based approach is more scalable than the memory-based approaches. For larger XML document, our approaches are faster compared to X-Diff [WDyC03] and comparable to XyDiff [CAM02] (C version). In addition, we also show that the result quality of our approaches is comparable to (or even, in some cases, better than) one detected by X-Diff. Observe that change detection on unordered is NP-Complete in general cases [ZSS92]. We also study that our approaches are able to produce better result quality compared to XyDiff [CAM02].

1.6 Dissertation Organization

This dissertation is organized as follows. In Chapter 2, we shall briefly present the prior works in topics related to our research. Chapter 3 discusses the overview of our approaches

CHAPTER 1. INTRODUCTION

and the terminology that we shall use to facilitate exposition in this dissertation. We shall elaborate on our approaches, namely, HELIOS-O and XANDY-O, for detecting the changes to *ordered* XML documents in Chapter 4. The performance study of HELIOS-O and XANDY-O is presented in Chapter 5. In Chapter 6, we discuss HELIOS-U and XANDY-U that are used to address the problem of detecting the changes to *unordered* XML documents. Chapter 7 presents the performance study of HELIOS-U and XANDY-U compared to X-Diff [WDyC03]. In Chapter 8, we elaborate on XANDY-H that solves the problem of detecting the changes to *hybrid* XML documents. We also present DTD-DIFF, an approach for detecting the changes to DTDs. Finally, in Chapter 9, we conclude this dissertation by summarizing the contributions of this dissertation and highlighting promising directions for future work.

Chapter 2

Literature Survey

In this chapter, we shall discuss prior works in topics related to our research. Our research is related to change management of web data, that is, HTML and XML change management system. We shall also present the state-of-the-art approaches for detecting the changes to tree structured data. This is because XML can be represented in a tree structure. As our approaches store XML documents in relational databases before detecting the changes, we shall also briefly discuss the previous approaches for storing XML documents in relational databases.

The discussion in this chapter is organized as follows. In Section 2.1, we present the previous approaches that address the problem of detecting changes to tree structured data. Section 2.2 discusses the state-of-the-art approaches for detecting the changes to HTML documents. We elaborate on prior works that address the problem of detecting changes to XML documents in Section 2.3. In Section 2.4, we briefly discuss the state-of-the-art approaches for storing XML documents in relational databases. Finally, we summarize discussion in Section 2.5.

2.1 Change Management of Tree Structured Data

In this section, we shall briefly discuss the prior works related to change management of tree structured data.

2.1.1 MH-DIFF

MH-DIFF [CRGMW96, CGM97] is an efficient algorithm for meaningful change detection between two hierarchically structured data snapshots, or trees. The structured data is modeled as rooted, labeled *unordered* trees. Each node has a label and one parent node.

Figure 2.1: MH-DIFF: Examples.

Detecting the changes means finding a set of edit operations which transforms the tree representation of one snapshot to that of the other. MH-DIFF introduces some basic edit operations of structured information changes: *insert*, *delete*, and *update* operations. There are also three additional edit operations that can decrease edit cost. They are *move*, *copy*, and *glue* operations. These operations become more important if the moved, copied or glued subtree is a large subtree.

Given two versions of a tree, namely, T_1 and T_2 , as depicted in Figure 2.1, MH-DIFF algorithm works as follows. First, MH-DIFF creates an *induced graph* that is a bipartite graph consisting of nodes of T_1 on one side and nodes of T_2 on another side, plus special nodes $\oplus$ on T_1 's side and node $\ominus$ on T_2 's side (Figure 2.2(a)). The next step is to prune the edges in the induced graph that are not needed by a *minimum-cost edit script* by applying pruning rules [CGM97] (Figure 2.2(b)). Then, the algorithm determines the *minimum edge cover*. The edges in the *pruned induced graph* that are not needed in forming the minimal edge cover are pruned (Figure 2.2(c)). Having found the minimum edge cover, MH-DIFF annotates the edges in the *pruned induced graph* (Figure 2.3). Finally, it generates minimal-cost edit script by using the *annotated pruned induced graph*. The algorithm runs in time $O(ne + e^2)$, where n is the number of leaf nodes, and e is “weighted edit distance” (typically, $e \ll n$). More precisely, e is the sum of the number of deleted and inserted subtrees, and the total size of subtrees that moved for the shortest edit script. The assumption used in the algorithm is that the documents do not contain duplicate or similar objects. However,

Figure 2.2: MH-DIFF: Induced Graph.

this assumption does not hold for XML documents. In XML documents, there can be many similar or duplicate objects. For example, consider T_1 in Figure 1.3(a). We notice that there are three “rank” nodes whose values are “Assoc Prof”. Therefore, the algorithm does not guarantee to generate the optimal results if we use it for detecting the changes to XML documents.

2.2 Change Management of HTML Documents

Most of the web pages, either dynamic or static web pages, on the web use HTML to display the data from human perspective. An HTML document consists of markups or tags and data. The hierarchies of tags and data in HTML documents are often meaningless since the tags, which are used to manipulate the representation of data, do not actually describe the data in HTML documents. Both tags and data in HTML documents can be changed at any time. The changes on the tags will only effect the representation of data.

In HTML documents, the changes can take place in two different levels, that is, *sub page level* and *page level*. In sub page level, the changes only occur on the content of HTML

Figure 2.3: MH-DIFF: Annotated Pruned Induced Graph.

documents. In page level, the changes take place on the content and hyperlinks in the HTML documents. The changes of hyperlinks in the HTML document change the structure of a Web site. The state-of-the-art approaches for detecting the changes to HTML documents are as following.

- The AT&T Internet Difference Engine (AIDE) [DBCK98] is a system that finds and displays changes to pages on the Web. A tool, called *HtmlDiff*, automatically compares two HTML pages and creates a “merged” page to show the differences with special HTML markups. AIDE also supports recursive tracking and differencing of a page and its descendants. When recursion is specified, the changes to the child pages are reported separately by default. A user may specify a number of operations in AIDE which includes registering a URL including the degree of recursion through links to other pages, view textual differences between a pair of versions, and view a graph showing the structure of a page, etc. With AIDE we can detect the changes of insert and delete types.
- TopBlend [CDHV00] is a HTML differencing tool implemented in Java. TopBlend uses the fast Jacobson-Vo algorithm [JV92], which solves the Heaviest Common Subsequence problem, for page comparison. Users are able to run TopBlend either on the server or client sides. The changes can be represented in either a merged HTML view

or a more convenient side-by-side view for web pages with complex graphics designs. TopBlend has been integrated with the AT&T Internet Difference Engine (AIDE) and can also be employed easily by other website tracking services.

- WebCiao system [CK97] is a system for visualizing and tracking the structure of Web sites. It can detect the structural differences of Web sites. The system consists of three major components: *HTML Information Abstractor*, *Database Differencing Tool*, and *WebCiao Operators*. WebCiao enables users to combine a set of query and analysis operators on a virtual database pipeline. The deltas can be represented in graphical view, database view, and HTML page. The graphical representation shows the relationships between pages using colored nodes to indicate which pages have been modified.
- WebGUIDE (Web Graphical User Interface to a Difference Engine) [DBCK96] is another tool that supports recursive document comparison: users may explore the differences between the pages with respect to two dates. Differences between pages are computed automatically and summarized in a new HTML page, and the differences in link structure are shown in graphical representations. WebGUIDE is a combination of two tools, that is, Ciao [CFKW95] and the AIDE. With Ciao, the high-level structural differences are displayed as graphs that show the relationships between pages using colored nodes to indicate which pages have been modified. Using AIDE, the low-level textual differences are illustrated by marking changes between versions and modifying anchors to cause documents reached from that page to be annotated. WebGUIDE allows a user to issue queries for specific types of deltas.
- WebCQ system [LPT00] is a prototype system for web information monitoring and delivery. It provides a personalized notification of what and how web pages of interest have been changed and personalized summarization of web page changes. Users' update monitoring requests are modeled as continual queries [LPT99] on the Web. With WebCQ, we can detect changes of insert, delete and update types.
- Flesca et al. [FFM01] proposed a new technique for measuring the similarity of two Web documents. This can be efficiently used to discover changes in selected portions of the original Web document. The authors were interested in changes that add, delete, or update information contained in specific portions of a Web page. A Web document is

CHAPTER 2. LITERATURE SURVEY

	Sub page level	Page level	Insert	Delete	Update	Delta Presentation
AIDE	●	●	●	●	● ¹	merged-page
TopBlend	●	●	●	●	● ¹	merged-page, side-by-side
WebGUIDE	●	●	●	●	● ¹	graph view, merged-page
WebCiao		●	●	●	●	graph view
WebCQ	●	●	●	●	●	merged-page, side-by-side, only difference
Flesca et al.	●		●	●	●	only change
SCD	●		●	●	●	semantic hierarchies view
Bhowmick et al.		●	● ²	● ²	● ²	delta web table

1 : Updates detected as deletions alongside insertions

2 : Operations are for both nodes and links

Figure 2.4: Comparisons of Approaches for Detecting the Changes to HTML Documents.

modeled as unordered labeled tree. Each node of the tree corresponds to a structuring HTML tag in the document. A document tree consists of a set of nodes, the parent function associating each node with its parent, the root node, labeling function from leaf node, typing function from nodes to their element types, and attribute function from nodes to their attribute names. The nodes in a document tree are characterized by their markup type and the associated set of attribute-value pairs. Leaf nodes have associated the actual textual content of the document. The authors used the complete weighted bipartite graph to compare two document subtrees. This weighted bipartite graph is used to establish association between elements in the old and new versions of a Web document. The changes are detected by measuring the similarities between subtrees. The authors proposed a language to specify the trigger which was called *Web update query* (web trigger) [FFM01].

- SCD algorithm [LN01] is a sub page level change detection algorithm which detects semantic changes of hierarchical structured data contents in any two HTML documents. SCD generates *semantic trees* for each version of a document and computes the weighted differences between two branches of two semantic trees. Applying SCD algorithm, two types of changes could be discovered in HTML documents, they are insertion and deletion.
- Bhowmick et al. [BMN03] presented a mechanism for detecting and representing changes given the old and new versions of a set of interlinked Web documents in the context of the Web warehousing system called WHOWEDA (Warehouse of Web Data) [Bho01, BMNL98] by using *web algebraic operators* such as the *web join* and the *outer*

	Memory-based	Relational-based	Ordered XML	Unordered XML	Hybrid XML	Content changes	Structural changes	Result Quality	Schema
XyDiff	●		●			●	●	Low	-
X-Diff	●			●		●	●	High	-
DeltaXML	●		●			●	●	High	-
DiffXML		●	●			●		High	Simplistic
Leonardi et al.		●	●			●		High	SUCXENT
XANDY		●	●	●	●	●	●	High	SUCXENT
HELIOS		●	●	●		●	●	High	Shared-Inlining

Figure 2.5: Comparisons of Approaches for Detecting the Changes to XML Documents.

web join. Web documents are stored in WHOWEDA in the form of *web tuples* which are represented as directed graph to detect the deltas. The proposed approach can be used for detecting and representing changes on HTML and XML documents.

Figure 2.4 depicts the comparisons of features of the approaches that are used for detecting the changes to HTML documents.

2.3 Change Management of XML Documents

In this section, we shall elaborate on the state-of-the-art approaches for detecting the changes to XML documents. In Chapter 1, we have discussed that the approaches for detecting the changes to XML documents can be classified into the *memory-based* approaches and the *relational-based* approaches. We shall organize the discussion in this section according to the above classification. Figure 2.5 summarizes the comparisons of features of the approaches that are used for detecting the changes to XML documents (including the approaches proposed in this dissertation).

2.3.1 The Memory-Based Approach

The memory-based approaches [CAM02, Fon01, Fon02, WDyC03, YB04, CE99, Cob03] mainly work as follows. First, two versions of an XML document are parsed into two trees that are stored in the main memory. By using algorithms/heuristics, the approaches determine *minimum-cost matching subtrees* (also called as *best matching subtrees*). Having found the best matching subtrees, the algorithms start to detect the changes. Finally, the delta files are produced. In this section, we shall discuss some state-of-the-arts memory-based approaches.

X-Diff

In [WDyC03], the authors introduced a new algorithm for detecting changes to *unordered* XML documents. Since X-Diff is focused on unordered trees, so X-Diff considers two trees are equivalent if they are *isomorphic*, they are identical except for the orders among siblings. X-Diff supports three types of changes: *insert*, *delete*, and *update* operations. Insert operation is used to add a new node or subtree. Delete operation is the inverse of insert operation. We use delete operation if we want to remove a node or subtree from the tree. Update operation is used to change the value of a node. X-Diff does not detect the movement of a node among its siblings as X-Diff focuses on *unordered* XML documents. The movement of a node/subtree to be the child node of other parent node is detected as a pair of deletion and insertion of nodes/subtrees.

We now elaborate on how X-Diff detects the changes to unordered XML documents by using an example. Given two versions of an XML document, namely, D_1 and D_2 as depicted in Figure 1.7, X-Diff parses these XML documents into XTrees T_1 and T_2 . While X-Diff parses D_1 and D_2 , it also computes the hash values and signatures of nodes in T_1 and T_2 . The signature is calculated as follows. If node x is a node, then $Signature(x) = /Name(x_1)/Name(x_2)/.../Name(x_n)/Name(x)/Type(x)$, where x_1 is the root node of T , $(x_1, x_2, x_3, ..., x_n, x)$ if the path from root node to x , and $Name(x)$ and $Type(x)$ are the node name and node type of node x , respectively. For example, the signature of node 6 in T_1 is $Signature(n_6) = /SigmodRecord/issue/articles/article/internal$. If node x is a text node, then $Signature(x) = /Name(x_1)/Name(x_2)/.../Name(x_n)/Type(x)$. For instance, the signature of the value node of node 3 in T_1 is $/SigmodRecord/issue/volume/text$.

The hash value is computed as follows. If node x is an attribute/text node, then $\mathbf{XHash}(x) = \text{hash}(\text{Type}(x) \bullet \text{Name}(x) \bullet \text{Value}(x))$. For example, the hash value of the value node of node 7 in T_1 is $\text{hash}(\text{text} \bullet \text{title} \bullet \text{Web Mining})$. If node x is an element node that which has a list of child nodes, $x_1, x_2, ..., x_n$, then $\mathbf{XHash}(x)$ is obtained as follows.

- Compute $\mathbf{XHash}(x_1), \mathbf{XHash}(x_2), ..., \mathbf{XHash}(x_n)$.
- Sort these XHash values, so that $\mathbf{XHash}(x_1') \leq \mathbf{XHash}(x_2') \leq ... \leq \mathbf{XHash}(x_n')$, where $x_1', x_2', ..., x_n'$ are the nodes after sorting.
- Concatenate $Type(x)$ and $Name(x)$ with the sorted XHash values, and compute the XHash value of x : $\mathbf{XHash}(x) = \text{hash}(Type(x) \bullet Name(x) \bullet \mathbf{XHash}(x_1') \bullet \mathbf{XHash}(x_2') \bullet ... \bullet \mathbf{XHash}(x_n'))$.

For example, the hash value of node 5 in T_1 is $\mathbf{XHash}(n_5) = \text{hash}(\text{internal} \bullet \text{articles} \bullet \mathbf{XHash}(n_6) \bullet \mathbf{XHash}(n_{14}))$, where $\mathbf{XHash}(n_6) \leq \mathbf{XHash}(n_{14})$.

Next, X-Diff starts to determine *minimum-cost matchings*. First, it compares the hash values of the root nodes of T_1 and T_2 . If $\text{hash}(\text{root}(T_1)) = \text{hash}(\text{root}(T_2))$, then this means that T_1 is isomorphic to T_2 . Otherwise, X-Diff finds the minimum-cost matching, $M_{\min}(T_1, T_2)$. To reduce the search space, X-Diff filters out all nodes that have the same hash values. The matching process is in the bottom-up fashion, that is, it starts by matching the leaf nodes followed by their parent nodes. Given node x in T_1 and node y in T_2 , first, X-Diff compares the signatures of nodes x and y . If they are similar, then the algorithm computes $\text{Dist}(x, y)$ and stores $\text{Dist}(x, y)$ in a distance table DT . Otherwise, the algorithm shall process the following leaf nodes. Having determined the edit distances between leaf nodes that have the same signature, the algorithm continues computing the edit distances of the parent nodes. The matching process will stop after the algorithm computes $\text{Dist}(\text{root}(T_1), \text{root}(T_2))$. For example, X-Diff first compares node 3 in T_1 to node 103 in T_2 . As $\text{Signature}(n_3) = \text{Signature}(n_{103})$, it computes $\text{Dist}(n_3, n_{103})$ and stores $\text{Dist}(n_3, n_{103})$ in a distance table DT . Next, it compares node 3 in T_1 to node 104 in T_2 . X-Diff does not compute $\text{Dist}(n_3, n_{104})$ as $\text{Signature}(n_3) \neq \text{Signature}(n_{104})$. The comparisons continue until node 32 in T_1 and node 131 in T_2 are compared. The next step is to compare the parent nodes of these leaf nodes. First, X-Diff compares node 2 in T_1 to node 102 in T_2 . As $\text{Signature}(n_2) = \text{Signature}(n_{102})$, it computes $\text{Dist}(n_2, n_{102})$ and stores $\text{Dist}(n_2, n_{102})$ in a distance table DT . The matching process is stopped after X-Diff computes $\text{Dist}(n_1, n_{101})$.

The next step is to determine a minimum-cost matching, $M_{\min}(T_1, T_2)$, from the distance table DT . The process of determining the minimum-cost matching is in top-down fashion. Finally, X-Diff generates minimum edit script E for $(T_1 \rightarrow T_2)$ based on $M_{\min}(T_1, T_2)$.

The complexity of X-Diff is bounded by the complexity of finding minimum-cost matchings, that is, $O(|T_1| \times |T_2| \times \max\{\deg(T_1), \deg(T_2)\} \times \log_2(\max\{\deg(T_1), \deg(T_2)\}))$, where $\deg(T_1)$ and $\deg(T_2)$ are the maximum out-degree of T_1 and T_2 , respectively. One of the main contributions of X-Diff is that X-Diff generates more accurate results compared to XyDiff [CAM02]. Hence, X-Diff is suitable for the situations that users want to get more accurate results. One of the drawbacks is that the min-cost max-flow algorithm for computing the bipartite mapping between two XML trees is the most time consuming part in X-Diff. In addition, X-Diff also suffers from the scalability and multiple times parsing

problems as discussed in Chapter 1. In this dissertation, we shall show that our approaches are faster (for larger data sets) and more scalable than X-Diff. We also shall show that our approaches are able to produce XDeltas that have comparable result quality compared to the one generated by X-Diff.

XyDiff

In [CAM02], the authors presented a diff algorithm for XML data called XyDiff, which is a part of Xyleme project [Xyl]. Xyleme is a dynamic World Wide XML warehouse which has capability of storing massive volume of XML data. In Xyleme system, XML documents are considered as *ordered trees*. All nodes have values, data for text nodes and labels for element nodes. XyDiff supports four types of changes: *insert*, *delete*, *update*, and *move* operations. Insert operation is used to add a new node or subtree. We use delete operation if we want to remove a node or subtree from the tree. Update operation is used to change the value of a node. Move operation moves a subtree rooted at a given node to another specified position in the tree. Note that move operation can be replaced a pair of delete and insert operations.

We now elaborate on how XyDiff detects the changes to ordered XML documents by using an example. Given two versions of an XML document, namely, D_1 and D_2 , as depicted in Figure 1.2, XyDiff parses D_1 and D_2 into T_1 and T_2 , respectively. While XyDiff parses them, it calculates a hash value for every node in T_1 and T_2 . It also computes the weight of each node. A weight is the size of the content for text nodes and the sum of the weights of children for element nodes. For example, consider node 6 in T_1 (Figure 1.3(a)). The weight of node 6 is equal to the size of the content, that is, 10. Now, consider node 5 in T_1 . The weight of node 5 is the sum of weights of nodes 6 and 7. The weight of node 5 is equal to 27. Next, XyDiff constructs a priority queue designed to contain subtrees from the new document. The priority is given by the weights.

Having constructed a priority queue, XyDiff is ready to find matchings starting from heaviest nodes. First, it removes the heaviest node n in T_2 from the queue and constructs a list of candidates $\mathcal{C} = \{c_1, c_2, \dots, c_x\}$. From list $\mathcal{C}$, XyDiff selects the best candidate node c_b and matches c_b and n . If there are many candidates, the best candidate is one whose parent node matches the parent node of n , if any. Then, XyDiff propagates the matchings to the ancestors and descendants to get more matchings. For example, consider T_1 and T_2 that are depicted in Figure 1.3. XyDiff does not find an identical subtree for the tree rooted at nodes 101, but it finds out that node 103 matches node 15. Then, the matching is

propagated to the parent nodes. We notice that node 102 can be matched to node 8 as they have the same node name. XyDiff also checks the siblings of node 101. It finds out that nodes 104 and 10 may be eventually be matched even though they have different node value. It also matches nodes 105 and 11. The matchings are also propagated to other subtrees in T_1 and T_2 until XyDiff finds all matchings.

Finally, XyDiff computes the delta after finding all matchings. First, it determines the insertions and deletions followed by updates. Next, it detects the move operations (for both moves to other subtrees and moves among siblings). To find a minimum number of moves among siblings, XyDiff finds a *largest order preserving subsequence*. Having detected the changes, XyDiff generates the delta. The delta only consists of the edit operations that transform T_1 to T_2 . Observe that the delta is in the specific format in the context of Xyleme system.

The complexity of XyDiff is $O(n \times \log(n))$. Even though XyDiff has $O(n \times \log(n))$ complexity in execution time and is able to generate fairly good result in many cases, XyDiff cannot guarantee any form of optimal/near-optimal results because of the greedy rules used in the algorithm. Compared to X-Diff, XyDiff has better performance; however, the result quality of XyDiff is worse than the one of X-Diff. As XyDiff is categorized as an memory-based approach, it also faces the scalability and multiple times problems as discussed in Chapter 1. In this dissertation, we shall show that our approaches are able to produce higher result quality than XyDiff.

DeltaXML

DeltaXML [Del, Fon01, Fon02] commercially provides a plug-in solution for detecting and displaying changes to content between two versions of an *ordered* XML document. It represents the changes in XML with minimal additional attributes and elements. In other words, its XDeltas are well-formed XML documents. The attributes are used to describe types of changes. The changes are detected by using a technique based on longest common subsequence computations.

When the DTD of the XML documents is available, DeltaXML is able to validate the changes to the DTD. Hence, it produces more meaningful and correct XDeltas [Fon01]. For example, consider two versions of an XML document as depicted in Figures 2.6(a) and 2.6(b). If the DTD of these XML documents is not available, then DeltaXML will assume that these XML documents conform to DTD as depicted in Figure 2.6(c). Hence, it produces

Figure 2.6: DeltaXML: The Presence of DTD.

the XDelta as depicted in Figure 2.6(e). If the DTD is available (Figure 2.6(d)), then it produces the XDelta as depicted in Figure 2.6(f). Observe that the XDelta in Figure 2.6(f) is more accurate than the one in Figure 2.6(e).

2.3.2 The Relational-based Approach

The relational-based approaches [CMB04, LBM05, LBDM04, LBB05, LB05a, LB, LB05b, LB07] mainly work as follows. They parse two versions of an XML document into two trees and store both trees in a relational database. By issuing SQL queries (whenever appropriate), the approaches determine *minimum-cost matching subtrees* (also called as *best matching subtrees*) and detect the changes. In this section, we shall discuss some state-of-the-arts relational-based approaches. Note that the prior works for storing XML documents in relational databases will be discussed in subsequent section.

DIFFXML

Chen et al. proposed a relational-based approach called DIFFXML [CMB04] for detecting the changes to *ordered* XML documents. DIFFXML uses a modified method to store XML documents in relational database based on Rudolf Bayer's slide [Bay01]. This schema is as follows.

XML_IND (ID, DID, Path, SurrogatePattern, Value)

ID	DID	Path	SurrogatePattern	Value
1	1	univStaff/staff/name	1[1]1[1]1[1]	Smith
2	1	univStaff/staff/rank	1[1]1[1]2[1]	Assoc Prof
3	1	univStaff/staff/interest/research	1[1]1[1]3[1]1[1]	Web Mining
4	1	univStaff/staff/interest/research	1[1]1[1]3[1]1[2]	Multimedia Mining
5	1	univStaff/staff/name	1[1]1[2]1[1]	Chen
6	1	univStaff/staff/rank	1[1]1[2]2[1]	Assoc Prof
7	1	univStaff/staff/interest/research	1[1]1[2]3[1]1[1]	Data Mining
8	1	univStaff/staff/interest/research	1[1]1[2]3[1]1[2]	Web Data Management
9	1	univStaff/staff/name	1[1]1[3]1[1]	Mark
10	1	univStaff/staff/rank	1[1]1[3]2[1]	Assoc Prof
11	1	univStaff/staff/interest/research	1[1]1[3]3[1]1[1]	Digital Libraries
12	1	univStaff/staff/name	1[1]1[4]1[1]	Don
13	1	univStaff/staff/rank	1[1]1[4]2[1]	Asst Prof
14	1	univStaff/staff/interest/research	1[1]1[4]3[1]1[1]	Indexing
15	1	univStaff/staff/interest/research	1[1]1[4]3[1]1[2]	Security

Figure 2.7: DIFFXML: XML in Relational Database.

Figure 2.8: DIFFXML: Move Ratio Threshold.

The *ID* attribute is used as a unique index generated by database. The *DID* attribute is used to specify the document id to which particular tuple belongs to. The *Path* attribute specifies the path from root node to the particular leaf node in an XML tree. The *Value* attribute is used to keep the values of leaf nodes. The *SurrogatePattern* attribute is used to determine the ancestor-descendant relationship between nodes. Suppose we have an XML document as depicted in Figure 1.2(a). Figure 2.7 depicts the XML document shredded in relational database. DIFFXML is only able to detect the *content changes*. The delta detected by DIFFXML consists of insertions of leaf nodes, deletion of leaf nodes, content updates, and moves of leaf nodes.

Suppose we have two versions of an XML document, D_1 and D_2 as depicted in Figures 1.2(a) and 1.2(b), respectively. The tree representations of D_1 and D_2 are depicted in Figures 1.3(a) and 1.3(b), respectively. After T_1 and T_2 are stored in relational database, DIFFXML detects the changes as following. First, it detects the move operations. It first

finds all possible root nodes of moved subtrees. Then, these candidate tuples are tested to find out which tuples are actually moved and which are not. For each tuple, we collect all its decedents and test the portion of decedents that kept unchanged during moves. If the percentage of unchanged decedents exceeds a certain value, called the *move ratio threshold* (MRT), then that tuple will be determined as a moved tuple and will be marked with a MOVE flag. For example, consider T_1 and T_2 as depicted in Figures 2.8(a) and 2.8(b), respectively. Suppose we set the value of MRT to 0.5. Node 2 in T_1 does not move to be node 15 in T_2 . This is because the move ratio of nodes 2 and 15 is 0.333 that is less than MRT. Let us now consider T_1 and T_3 as depicted in Figures 2.8(a) and 2.8(c), respectively. We can consider that node 2 in T_1 moves to be node 25 in T_3 . This is because the move ratio of nodes 2 and 25 is 0.666 that is greater than MRT.

The second step is to detect the update operations. There are two types of update operations: the *absolute update* and *relative update*. An absolute updated node is one whose position in the DOM tree is not changed, but its value is updated. A node is considered as a relative updated node if it is in a moved subtree, its value is changed, and its relative position in the DOM tree is not changed. The last steps are to detect the insertions and deletions. The detected delta is depicted in Figure 2.9.

DIFFXML is similar to our preliminary work in [LBDM04]. Both of them only detect the content changes to *ordered* XML documents. There are some major differences as follows. First, in our preliminary work [LBDM04], we use SUCXENT schema [PBM04b] as the underlying schema. On the other hand, the underlying relational schema of DIFFXML is simplistic and not efficient for path expressions query processing without support of proper indexing scheme. Ideally, a change detection system built on top of a relational database should also support efficient insertion and extraction of XML documents and efficient execution of path expression queries. Our approach in [LBDM04] uses SUCXENT schema that enables us to insert, extract, and query XML data efficiently.

2.4 Storing XML in RDMS

XML document is usually stored in text file, while data on the Web are currently stored in the relational database [KCR02]. Currently, there are several alternatives for storing XML data: in semi-structured databases [GMW99], in LDAP directories [eB2, ML01], in object-oriented databases systems [BBB⁺88, BLL⁺02], object-relational databases systems [CKS⁺00, RP02,


```

// Move(id, parent_id, new_pos)
Move(15, 102, 1) // Leaf node "Mark"
Move(18, 105, 1) // Leaf node "Digital Libraries"
Move(13, 111, 1) // Leaf node "Web Data Management"

// Update(id, old_value, new_value)
Update(16, "Assoc Prof", "Prof")
Update(7, "Multimedia Mining", "Information Retrieval")
Update(16, "Assoc Prof", "Asst Prof")

// Insert(id, parent_id, pos, value)
Insert(114, 113, 1, "Steve")
Insert(117, 116, 1, "Semantic Web")

// Delete(id)
Delete(3)
Delete(6)
Delete(12)

```

Figure 2.9: DIFFXML: XDelta.

KC02], in relational databases systems [FK99, JLWY02a, JLWY02b, TVB⁺02, STZ⁺99, YASU01, RFHR03, BFRS02, PBM04a, PBM04b, BWLM03], and in native XML databases systems [Sch01, WNL03, Mei02, PAKC⁺03]. Among these approaches, the relational storage approach has attracted considerable interest with a view to leveraging their powerful and reliable data management services. In order to store an XML document in a relational database, the tree structure of the XML document must first be mapped into an equivalent, flat, relational schema. XML documents are then shredded and loaded into the mapped tables.

Recall that the relational storage approaches can be classified into two categories, namely, the *schema-conscious* approach [STZ⁺99, RFHR03, BFRS02] and *schema-oblivious* approach [FK99, JLWY02a, JLWY02b, TVB⁺02, YASU01, PBM04a, PBM04b, BWLM03]. In this section, we shall briefly discuss some of the relational storage approaches for storing XML documents. Figure 2.10 depicts the comparisons of features of the approaches that are used for storing XML documents in relational databases.

2.4.1 The Schema-conscious Approach

In [STZ⁺99], the authors proposed three techniques for storing XML in RDBMS: *the Basic-Inlining technique*, *the Shared-Inlining technique*, and *the Hybrid-Inlining technique*. The proposed techniques need the existence of DTD of the XML documents. The first step in these approaches is to simplify complex DTDs. Next, the DTD elements are grouped and converted into relations. The inlining techniques inline as many descendants of an element as

	Schema-oblivious	Schema-conscious	Order Preserving
Edge	●		●
XRel	●		●
XParent	●		●
SUCXENT	●		
SUCXENT++	●		
Inlining		●	
LegoDB		●	

Figure 2.10: Comparisons of Approaches for Storing XML Documents in RDBMS.

possible into their ancestors to reduce number of relations. As a result, the fragmentation of data is minimized and the cost of joining relations during query processing is reduced. The recursive relationship is handled by using foreign keys. We shall elaborate on the inlining techniques further in Chapter 4. Note that we use the Shared-Inlining techniques for storing XML documents in our approaches [LB05a, LB05b, LB07].

LegoDB [RFHR03, BFRS02] is a cost-based XML-to-relational mapping engine. It explores a space of possible mappings and selects the best mapping for a given application defined by XML Schema, XML data statistics, and an XML query workload. Compared to Shared Inlining [STZ⁺99], LegoDB system exploits a richer set of mapping primitives. In addition to parent-child relationships, LegoDB also takes into account additional schema constructs such as choice and repetition, and it allows multiple mapping functions for a given construct. The steps of generating relational schema in LegoDB are as following. Given an XML schema and statistics extracted from an example XML database, LegoDB generates the initial *physical schema*. Then, LegoDB translates the *physical schema* into a corresponding relational catalog (relational schema and statistics) and XQuery workload into SQL queries. The *relational optimizer* in LegoDB computes the cost estimation for the relational catalog by using the transformed SQL queries. This process is repeatedly applied until LegoDB finds a *good* configuration.

2.4.2 The Schema-oblivious Approach

In this section, we shall present some of the state-of-the-arts approaches that are classified as schema-oblivious approaches.

Florescu et al. Approach

In [FK99], Florescu et al. proposed three approaches for storing XML in RDBMS: *Edge*, *Binary*, and *Universal* approaches.

The Edge approach stores XML data graphs (a directed graph) in a table called *Edge*. This approach does not need the existence of DTD. The **Edge** table consists of six attributes: *SourceID*, *Ordinal*, *TargetID*, *Label*, *Flag*, and *Value*. The Edge approach stores all the edges of XML data graph. The edges are specified by the *SourceID* and *TargetID* attributes. The *Label* attribute keeps the edge name. The order of the edge among its sibling is kept in the *Ordinal* attribute. The *Flag* attribute indicates whether the edge represents inter-object reference or point to a value.

In binary approach, we group all edges with the same label into one table. This approach corresponds to a horizontal partitioning of the **Edge** table used in the first approach, using name as the partitioning attribute. Thus, we create as many Binary tables as different sub element and attribute names occur in the XML document. Each Binary table has the following structure:

$$B_{name}(source, ordinal, flag, target, value)$$

The key of such a Binary table is $\{source, ordinal\}$, and all the fields have the same meaning as in the Edge table.

A single Universal table can store all the edges. The **Universal** table corresponds to the result of a full outer join of all Binary tables. The structure of the Universal table is as follows, if $n_1, \dots, n_k$ are the label names.

$$Universal(source, ordinal_{n_1}, flag_{n_1}, target_{n_1}, ordinal_{n_2}, flag_{n_2}, target_{n_2}, \dots, \\ ordinal_{n_k}, flag_{n_k}, target_{n_k})$$

The Universal table has many fields which are set to null, and it also has a great deal of redundancy. In other words, the Universal table is denormalized, with all the known advantages and disadvantages of such a denormalization.

Igor Tatarinov et al. Approach

Tatarinov et al. [TVB⁺02] proposed an approach for storing and querying *ordered* XML documents. In order to store and query shredded ordered XML documents using a relational database system, we need some mechanism to capture document order in the relational data model. This is accomplished by encoding each nodes position in an XML document as a

data value. The authors presented three “lossless” order encoding methods that span the spectrum of query and update performance. The first of these encoding methods, *Global Order*, performs the best on queries. At the other end of the spectrum is *Local Order*, which performs best on updates. Finally, *Dewey Order* is a hybrid of the preceding two methods, which performs reasonably well on both queries and updates. The *Dewey Order* uses the concept of Dewey Decimal Classification [Cen].

The ordered XML documents can be stored in relational schema by modifying some previous approaches, such as the Edge Approach [FK99]. If the global encoding is used, the Edge approach relational schema will be modified as following.

$$\text{Edge}(id, parent_id, end_dec_id, path, value)$$

The *end_dec_id* attribute is used to store the node ID of the last descendent node. If the local encoding is used, the Edge approach relational schema will be modified as following.

$$\text{Edge}(id, parent_id, sIndex, path, value)$$

The *sIndex* attribute stores the left-to-right position among sibling nodes. If the dewey encoding is used, the Edge approach relational schema will be modified as following.

$$\text{Edge}(dewey, path, value)$$

The *dewey* attribute that is the primary key is used to store the dewey order of each node. The Dewey numbering scheme that is used by Tatarinov et al. approach may need extra space to store the order numbers of all nodes. The cost of updating the dewey values of nodes in case of the document is changed is still expensive.

XRel

The XRel approach [YASU01] stores XML data graph in four tables: *Path*, *Element*, *Text*, and *Attribute* tables as shown below.

$$\text{Path}(PathID, PathExp)$$

$$\text{Element}(DocID, PathID, Start, End, Ordinal)$$

$$\text{Text}(DocID, PathID, Start, End, Value)$$

$$\text{Attribute}(DocID, PathID, Start, End, Value)$$

The “*docID*”, “*pathID*”, “*start*”, “*end*”, and “*value*” attributes represent document identifier, *simple path expression identifier*, start position of a region, end position of a region,

and string-value, respectively. The *region* of an element or text node is a pair of numbers that represent, respectively, the start and end positions of the node in an XML document. The *region* of an attribute node is a pair of two identical numbers equal to the start position of the parent element node plus one.

XRel requires the use of $\theta(< \text{ or } >)$ joins to resolve ancestor-descendant relationships. These have been shown to be quite expensive due to the manner in which an RDBMS processes joins [ZND⁺01]. In fact, special algorithms such as the multi-predicate merge sort join algorithm [ZND⁺01] have been proposed to optimize these operations.

XParent

XParent [JLWY02a, JLWY02b] is a four-table database schema: **LabelPath**, **DataPath**, **Element**, and **Data** tables.

LabelPath(*ID*, *Len*, *Path*)

DataPath(*Pid*, *Cid*)

Element(*PathID*, *Did*, *Ordinal*)

Data(*PathID*, *Did*, *Ordinal*, *Value*)

The **LabelPath** table stores the label of paths. Each label-path is stored in the *Path* attribute. The *Len* attribute stores the number of edges of the label-path. Because data-paths usually vary in length, and can be very long, we store pairs of node identifiers in the **DataPath** table. Here, *Pid* and *Cid* attributes stores parent-node id and child-node id of an edge in the data-path, respectively. The *Did* attribute for data-path id is a node identifier which also serves as unique data-path identifier for data-path ended at the node itself. The **DataPath** table keeps the parent-child relationship. Therefore, it needs joins in order to check edge connections. To address this problem, the authors propose a new table called the **Ancestor** table which maintains the ancestor-descendant relationship.

Ancestor(*Did*, *Ancestor*, *Level*)

The *Did* attribute maintains a data-path identifier that identifies a data path. The *Ancestor* and *Level* attributes maintain ancestor-descendent relationships and .

XParent replaces θ -joins with *equi*-joins over this set of ancestors. This solves the problem of θ -joins as in XRel [YASU01]. One of the drawbacks of XParent approach is XParent stores every node in the document and in addition every ancestor of a node is also stored in the **Ancestor** table. This leads to a substantially larger database size and at the same time leads to a considerable increase in insertion and extraction times.

Document (<u>Doc_ID</u> , Doc_Name)
Path (<u>Path_ID</u> , Path_Exp)
LeafValue (<u>Doc_ID</u> , <u>LeafOrder</u> , <u>Path_ID</u> , LeftSibIxLevel, SiblingOrder, LeafValue)
AncestorInfo (<u>Doc_ID</u> , <u>MinSibOrder</u> , <u>MaxSibOrder</u> , <u>NodeLevel</u> , NodeName)

Figure 2.11: The SUCXENT Schema.

Document (<u>DocId</u> , Name)
Path (<u>PathId</u> , PathExp, <u>CPathId</u>)
TextContent (<u>DocId</u> , <u>PathId</u> , <u>LeafOrder</u> , <u>CPathId</u> , <u>BranchOrder</u> , <u>BranchOrderSum</u> , Text)
PathValue (<u>DocId</u> , <u>PathId</u> , <u>LeafOrder</u> , <u>CPathId</u> , <u>BranchOrder</u> , <u>BranchOrderSum</u> , LeafValue)
DocumentRValue (<u>DocId</u> , <u>Level</u> , RValue)

Figure 2.12: The SUCXENT++ Schema.

SUCXENT

In [PBM04b], Prakash et al. proposed a schema-oblivious approach for storing XML documents called SUCXENT (Schema Unconscious XML Enabled System - pronounced “succinct”). SUCXENT efficiently stores and queries XML data in relational database by storing only the leaf nodes and their ancestors. SUCXENT allows effective evaluation of queries without involving θ -joins. Figure 2.11 depicts the relational schema of SUCXENT. We shall elaborate on SUCXENT further in Chapter 4.

SUCXENT++

Recently, Prakash et al. proposed another schema-oblivious approach for storing XML documents called SUCXENT++ [PBM06, PBM04a, SWB⁺07]. SUCXENT++ outperforms existing schema-oblivious techniques, such as XParent, by up to 15 times and Shared Inlining - a schema-conscious approach - by up to 8 times for certain types recursive queries [PBM06, PBM04a]. In addition, SUCXENT++ can reconstruct shredded documents up to 2 times faster than Shared-Inlining.

Figure 2.12 depicts the schema of SUCXENT++. The **Document** table is used for storing the names of the documents in the database. Whenever a new document is inserted in

SUCXENT++ its file name or URL is stored in the *Name* attribute and a unique identifier is stored in the *DocId* attribute. This identifier is used as a reference to this document in the rest of the schema. The *Path* table records every unique root-to-leaf path encountered in the XML documents stored in the database. This table maintains path identifiers and relative path expressions recorded as instances of *PathId* and *PathExp* respectively. The *CPathId* attribute is used for optimization.

The *PathValue* table stores the leaf nodes of the XML documents stored in the database. The *DocId* attribute in a tuple refers to the document the leaf node in this tuple belongs to. The *PathId* attribute refers to the root-to-leaf path corresponding to this leaf node as stored in the *Path* table. The *CPathId* attribute is used for optimization purpose only. The *LeafValue* attribute stores the textual content of the leaf node. The *LeafOrder* attribute is used to record the node order of the leaf nodes in an XML tree. The *BranchOrder* attribute records the intersection level (nearest common ancestor) of a leaf node with another leaf node that immediately precedes it. The *BranchOrder* attribute is used to enable efficient query processing. The *TextContent* table is used to store large text data (such as DNA sequences). This table has similar structure to the *PathValue* table. The *DocumentRValue* table is used to enable SUCXENT++ to identify the intersection level or level of the nearest common ancestor of these two nodes efficiently. The details on how to compute the values of *RValue*, *BranchOrderSum*, and *CPathId* attributes can be found in [PBM06, PBM04a].

2.5 Summary

In this chapter, we have described the prior works related to our research. First, we presented the previous approaches for detecting the changes to tree structured data. We also briefly discussed the state-of-the-art approaches for detecting the changes to HTML documents. We elaborated on the approaches for detecting the changes to XML documents. Finally, the approaches for storing XML documents in relational databases were also presented.

Chapter 3

Background

In this chapter, we shall present the overview of our proposed approaches (Section 3.2). Section 3.3 shall present the terminology that we shall use subsequently to facilitate exposition in this dissertation. Finally, we shall elaborate on the relational schemas used by our approaches for storing XML documents before the changes are detected in Section 3.4.

3.1 Background

In this dissertation, we propose two relational-based approaches for detecting the changes to XML documents as follows. The first approach is XANDY (**X**ml **enA**bled **chaN**ge **D**etection **sY**stem) that uses the *schema-oblivious* approach for storing the XML documents, that is, SUCXENT [PBM04b] (**S**chema **U**nconscious **X**ML **E**nabled **S**ystem - pronounced “*succinct*”). We chose SUCXENT because Prakash et al. has shown in [PBM04b] that SUCXENT outperforms significantly the current state-of-the-art *schema-oblivious* approaches like XParent [JLWY02a] as far as storage size (up to 20%), insertion time (up to six times), extraction time, and path expression queries (up to 25%) are concerned. Note that Jiang et al. has shown in [JLWY02a] that XParent outperforms existing *schema-oblivious* approaches such as Edge approach [FK99], and XRel [YASU01].

The second approach we shall discuss is HELIOS (*scHEma-conscious xml-enabL*ed *change detectIO*n **S**ystem) that uses a *schema-conscious* approach for storing the XML documents (Shared-Inlining [STZ⁺99] in our case). Observe that the characteristics of schema-conscious approach raise certain challenges. For instance, in this approach no special relational schema needs to be designed as it can be generated on the fly based on the DTD of the XML document(s). That is, unlike schema-oblivious approaches, the underlying relational schema is DTD-dependent. Consequently, the challenge is to create a general framework for change

Figure 3.1: Two Examples of XDeltas.

detection so that the framework is independent of the structural heterogeneity of various XML documents. In other words, given a specific schema-conscious approach, our framework should be able to detect all changes accurately independent of the changes to the underlying relational schema due to different structures of XML documents. Note that the framework discussed in this dissertation is only for XML documents whose schemas do not contain recursive elements.

The objective of XANDY and HELIOS is to detect the *minimum XDelta* between two versions of an XML document. The minimum XDelta is formally defined as follows.

Definition 3.1 [Minimum XDelta] Let T_1 and T_2 be the first and second versions of an XML document, respectively. Let XDelta $\delta = \langle o_1, o_2, \dots, o_n \rangle$ be a sequence of edit operations that transforms T_1 to T_2 . Let $|\delta|$ be the number of edit operations in XDelta δ . Let $\Delta = \{\delta_1, \delta_2, \dots, \delta_n\}$ be a set of XDeltas such that $\forall \delta_i \in \Delta$ transforms T_1 to T_2 , where $1 \leq i \leq |\Delta|$. Then, δ_x is the **minimum XDelta** of T_1 and T_2 iff $(|\delta_x| < |\delta_y|) \forall 1 \leq y \leq |\Delta|$ and $x \neq y$, where $\delta_x \in \Delta$ and $\delta_y \in \Delta$. $\square$

Example 3.1 [Minimum XDelta] Consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. Suppose we have XDelta δ_1 that consists of seven update operations and one deletion as depicted in Figure 3.1(a). Figure 3.1(b) depicts XDelta δ_2 that has six edit operations. XDelta δ_2 is a candidate of minimum XDelta. XDelta δ_2 shall be a minimum XDelta if there are no other XDeltas that have number of edit operations lesser than six. $\square$

3.2 Overview

In XANDY and HELIOS, there are two phases that are used to facilitate us to detect the minimum XDelta. Let us elaborate on these phases further. Suppose we have two versions

of an XML document, namely, T_1 and T_2 . First, T_1 and T_2 are stored in relational database. Then, XANDY/HELIOS can be used to detect the changes to T_1 and T_2 . The two phases of XANDY/HELIOS in detecting the changes to XML documents are as follows.

(i) **Phase 1: Finding Best Matching Subtrees.**

The objective of Phase 1 is to find the most similar subtrees in T_1 and T_2 . In order to find the most similar subtrees, we need to match subtrees in T_1 to ones in T_2 . Observe that a subtree in T_1 can be matched to more than one subtree in T_2 , and vice versa. In addition, we need to measure the similarity of each matching by calculating the *similarity score*. The most similar matching subtrees are called *best matching subtrees*. Note that we issue SQL queries (whenever appropriate) to find the best matching subtrees. Consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. We observe that subtree rooted at node 14 (t_{14}) can be matched to t_{102} and t_{108} . This is because t_{14} and t_{102} have some similar components. Similarly, t_{14} and t_{108} also have some similar components. Intuitively, t_{14} is more similar to t_{102} than to t_{108} . Therefore, we can determine that t_{14} and t_{102} are best matchings subtrees in T_1 and T_2 . Phase 1 is important as it facilitates us to find *minimum XDeltas*. For instance, if we match t_2 and t_{102} , t_8 and t_{108} , t_{14} and t_{113} , and t_{19} and t_{118} , then we shall have XDelta as depicted in Figure 3.1(a). The XDelta depicted in Figure 3.1(b) can be detected if we match t_8 and t_{108} , t_{14} and t_{102} , and t_{19} and t_{118} .

(ii) **Phase 2: Change Detection.**

Having determined the best matching subtrees between T_1 and T_2 , XANDY/HELIOS issues SQL queries (whenever appropriate) to detect different types of changes as discussed in Chapter 1. The detected changes are stored in several relations. Consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. Phase 1 will result three pairs of best matching subtrees, that is, t_{14} and t_{102} , t_8 and t_{108} , and t_{19} and t_{118} . If we use this information, then we detect an XDelta as depicted in Figure 3.1(b).

3.3 Terminology

We now define some symbols that we shall use subsequently to facilitate exposition. Let T be a tree representation of an XML document D . The root node of T is denoted by $root(T)$. Let $\mathcal{L}(T) = \{\ell_1, \ell_2, \dots, \ell_n\}$ be a set of leaf nodes in XML tree T . The textual content of a

leaf node ℓ is denoted by $value(\ell)$. A set of internal nodes in T is denoted by $\mathcal{I}(T)$, and i denotes an internal node, where $i \in \mathcal{I}$. The name and level of node n are denoted by $name(n)$ and $level(n)$, respectively. Then, $path(n)$ denotes the path from $root(T)$ to node n . The parent node, child node, and ancestor node of node n are denoted as $parent(n)$, $child(n)$, and $ancestor(n)$, respectively. In ordered XML, the left-to-right position of a node among its siblings is significant. Hence, $pos(n)$ denotes the left-to-right position of node n among its siblings if D is an *ordered XML*.

3.3.1 Best Matching Subtrees

Suppose we have two XML trees, namely, T_1 and T_2 . Recall that the objective of Phase 1 is to find the best matching subtrees in T_1 and T_2 by measuring the *similarity score*. The first step in finding the best matching subtrees is to determine the matching leaf nodes, from which the matchings are propagated in order to find matching internal nodes. From a set of matching internal nodes, then, we determine the best matching subtrees. Formally,

Definition 3.2 [Matching Leaf Nodes] Let $\ell_1 \in \mathcal{L}(T_1)$ and $\ell_2 \in \mathcal{L}(T_2)$ be two leaf nodes from the tree representations of the first and second versions of an XML document, respectively. Then, ℓ_1 and ℓ_2 are **matching leaf nodes** (denoted as $\ell_1 \leftrightarrow \ell_2$) if $name(\ell_1) = name(\ell_2)$, $level(\ell_1) = level(\ell_2)$, $path(\ell_1) = path(\ell_2)$, and $value(\ell_1) = value(\ell_2)$, where $\ell_1 \in \mathcal{L}(T_1)$ and $\ell_2 \in \mathcal{L}(T_2)$. $\square$

Example 3.2 [Matching Leaf Nodes] Consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. Leaf nodes 13 and 112 are matching leaf nodes ($\ell_{13} \leftrightarrow \ell_{112}$) because they have the same node name (“*interest*”), the same node level (“4”), the same node value (“*Web Data Management*”), and the same path (“*/univStaff/staff/research/interest*”). Note that a leaf node in T_1 can be matched to more than one leaf node in T_2 , and vice versa. Leaf node 110 in T_2 can be matched to leaf nodes 4, 10, and 16 in T_1 as they satisfy the conditions in Definition 3.2. $\square$

In ordered XML, we classify the matching leaf nodes into two types, namely, *fixed matching leaf nodes* and *shifted matching leaf nodes*. As the left-to-right position among siblings of a node in ordered XML is important, matching leaf nodes that have the same position among siblings are more meaningful than the ones that have different position among siblings. If the left-to-right position among siblings of a node is changed, then it is possible that this node is moved among its siblings. Formally,

Definition 3.3 [Fixed/Shifted Matching Leaf Nodes] Let $\ell_1 \leftrightarrow \ell_2$ be matching leaf nodes. Then,

- ℓ_1 and ℓ_2 are **fixed matching leaf nodes** iff $\text{pos}(\ell_1) = \text{pos}(\ell_2)$;
- ℓ_1 and ℓ_2 are **shifted matching leaf nodes** iff $\text{pos}(\ell_1) \neq \text{pos}(\ell_2)$. □

Example 3.3 [Fixed/Shifted Matching Leaf Nodes] Consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. Leaf nodes 18 and 106 are fixed matching leaf node as they are matching leaf nodes and $\text{pos}(\ell_{18}) = \text{pos}(\ell_{106})$. Leaf nodes 13 and 112 are shifted matching leaf node as they are matching leaf nodes and $\text{pos}(\ell_{13}) \neq \text{pos}(\ell_{112})$. □

In unordered XML, we do not classify the matching leaf nodes as the left-to-right position among siblings of a node in unordered XML is not significant. Note that if ℓ_1 and ℓ_2 are not matching leaf nodes, then they are denoted by $\ell_1 \nleftrightarrow \ell_2$. Next, we define the notion of *matching leaf node groups*. Formally,

Definition 3.4 [Matching Leaf Node Groups] Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two sets of leaf nodes whose parent nodes are i_1 and i_2 , respectively, where $i_1 \in \mathcal{I}(T_1)$ and $i_2 \in \mathcal{I}(T_2)$. Then, $\mathcal{G}_1$ and $\mathcal{G}_2$ are **matching leaf node groups** (denoted as $\mathcal{G}_1 \Leftrightarrow \mathcal{G}_2$) iff $\exists \ell_x \exists \ell_y$ such that $\ell_x \leftrightarrow \ell_y$, where $\ell_x \in \mathcal{G}_1$ and $\ell_y \in \mathcal{G}_2$. □

Example 3.4 [Matching Leaf Node Groups in Ordered XML] Consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. Suppose $\mathcal{G}_{17} = \{\ell_{18}\}$ and $\mathcal{G}_{105} = \{\ell_{106}, \ell_{107}\}$ are two sets of leaf nodes in T_1 and T_2 whose parent nodes are nodes 17 and 105, respectively. We observe that $\mathcal{G}_{17} \Leftrightarrow \mathcal{G}_{105}$ as $\ell_{18} \leftrightarrow \ell_{106}$, $\ell_{18} \in \mathcal{G}_{17}$, and $\ell_{106} \in \mathcal{G}_{105}$. □

Example 3.5 [Matching Leaf Node Groups in Unordered XML] Let us now consider T_1 and T_2 as depicted in Figures 1.8(a) and 1.8(b), respectively. We notice that $\mathcal{G}_{10} \Leftrightarrow \mathcal{G}_{110}$ as $\ell_{13} \leftrightarrow \ell_{111}$ and $\ell_{11} \leftrightarrow \ell_{112}$. Note that $\mathcal{G}_{10} = \{\ell_{11}, \ell_{12}, \ell_{13}\}$ and $\mathcal{G}_{110} = \{\ell_{111}, \ell_{112}\}$ are two sets of leaf nodes in T_1 and T_2 whose parent nodes are nodes 10 and 110, respectively. □

Next, we define *matching subtrees*. The root nodes of two matching subtrees are called *matching internal nodes*. Observe that a subtree in T_1 can be matched to more than one subtree in T_2 and vice versa. From a set of matching subtrees, we determine the most similar subtrees to be *best matching subtrees*. Note that we only match two subtrees at the same level. This is because the matching subtrees at different level is an expensive process [WDyC03]. As a result, X-Diff [WDyC03] and XyDiff [CAM02] also match two subtrees at the same level. Formally, the matching subtrees are defined as follows.

Definition 3.5 [Matching Subtrees] Let t_1 and t_2 be two subtrees rooted at nodes $i_1 \in \mathcal{I}(T_1)$ and $i_2 \in \mathcal{I}(T_2)$, respectively. Then, t_1 and t_2 are **matching subtrees** (denoted by $t_1 \simeq t_2$) if $\text{name}(i_1) = \text{name}(i_2)$, $\text{level}(i_1) = \text{level}(i_2)$, $\text{path}(i_1) = \text{path}(i_2)$, and $\exists p \exists q$ such that $p \leftrightarrow q$, where $i_1 = \text{ancestor}(p)$, $i_2 = \text{ancestor}(q)$, $p \in \mathcal{L}(T_1)$, and $q \in \mathcal{L}(T_2)$. $\square$

Example 3.6 [Matching Subtrees in Ordered XML] Consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. The subtrees rooted at node 8 in T_1 and node 108 in T_2 are matching subtrees ($t_8 \simeq t_{108}$) as they have three matching leaf nodes ($\ell_9 \leftrightarrow \ell_{109}$, $\ell_{10} \leftrightarrow \ell_{110}$, and $\ell_{13} \leftrightarrow \ell_{112}$). The subtrees rooted at node 14 in T_1 and node 108 in T_2 are also matching subtrees ($t_{14} \simeq t_{108}$) as they have one matching leaf node ($\ell_{16} \leftrightarrow \ell_{110}$). $\square$

Example 3.7 [Matching Subtrees in Unordered XML] Consider T_1 and T_2 as depicted in Figures 1.8(a) and 1.8(b), respectively. The subtrees rooted at node 6 in T_1 and node 106 in T_2 are matching subtrees ($t_6 \simeq t_{106}$) as they have four matching leaf nodes ($\ell_8 \leftrightarrow \ell_{108}$, $\ell_9 \leftrightarrow \ell_{109}$, $\ell_{11} \leftrightarrow \ell_{112}$, and $\ell_{13} \leftrightarrow \ell_{111}$). $\square$

Note that if t_1 and t_2 are not matching subtrees, then they are denoted by $t_1 \not\simeq t_2$. We use the terms of *matching subtrees* and *matching internal nodes* interchangeably.

Having found a set of matching subtrees, we need to measure the degree of similarity between two matching subtrees. We use the proportion of matching leaf nodes and total number of leaf nodes in the matching subtrees. Observe that matching leaf nodes have the same path. Therefore, the ancestor nodes of two matching leaf nodes must have the same node names and levels. If they have different paths, then it is more likely that they are in different level, or one of the ancestors' names is updated. Note that we do not consider node name updates. X-Diff [WDyC03] and XyDiff [CAM02] also do not consider node name updates. We only consider the types of changes as presented in Chapter 1. We now define a metric called *similarity score* to measure how similar two subtrees are. Formally,

Definition 3.6 [Similarity Score] The similarity score $\mathfrak{R}$ of two subtrees t_1 and t_2 that are in T_1 and T_2 , respectively, is as follows:

- if T_1 and T_2 are **ordered XMLs**, then $\mathfrak{R}(t_1, t_2) = \frac{2|A|+|B|}{|t_1|+|t_2|}$ where $|t_1|$ and $|t_2|$ are the total numbers of leaf nodes in t_1 and t_2 , respectively, $|A|$ and $|B|$ are numbers of nodes of fixed and shifted matching leaf nodes in t_1 and t_2 , respectively and $A \cap B = \emptyset$.

- if T_1 and T_2 are **unordered XMLs**, then $\mathcal{R}(t_1, t_2) = \frac{2|t_1 \cap t_2|}{|t_1| + |t_2|}$ where $|t_1 \cap t_2|$ is the number of matching leaf nodes in t_1 and t_2 . $\square$

Example 3.8 [Similarity Score in Ordered XML] Consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. The similarity score of t_8 in T_1 and t_{108} in T_2 is $\mathcal{R}(t_8, t_{108}) = 0.714$. The value of $|A|$ is 2 as $\ell_9 \leftrightarrow \ell_{109}$ and $\ell_{10} \leftrightarrow \ell_{110}$ are fixed matching leaf nodes. The value of $|B|$ is 1 as $\ell_{13} \leftrightarrow \ell_{112}$ is the only shifted matching leaf nodes. Then, $|t_1| + |t_2|$ is equal to 7. $\square$

Example 3.9 [Similarity Score in Unordered XML] Consider T_1 and T_2 as depicted in Figures 1.8(a) and 1.8(b), respectively. The similarity score of t_6 in T_1 and t_{106} in T_2 is $\mathcal{R}(t_6, t_{106}) = 0.667$. This is because the values of $|t_1 \cap t_2|$ and $|t_1| + |t_2|$ are equal to 4 and 12, respectively. $\square$

The value of similarity score is between 0 and 1. Two subtrees are more similar if the similarity score is higher. Based on the similarity score, we classify the subtrees into two types as follows.

- **Matching Subtrees** ($0 < \mathcal{R}(t_1, t_2) \leq 1$). The *matching subtrees* are ones in which they have at least one matching leaf node. Therefore, the similarity score must be greater than zero. If two matching subtrees have more matching leaf nodes, then the similarity score will be close to “1”. In some cases, the similarity score of two matching subtrees can be equal to “1”. In ordered XML, we called them as *identical subtrees* as they are identical. For example, consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. Then, t_{19} and t_{118} are identical subtrees. In unordered XML, we call them as *isomorphic subtrees* as they are identical except for the orders among siblings.
- **Unmatching Subtrees** ($\mathcal{R}(t_1, t_2) = 0$). The *unmatching subtrees* are ones in which they do not have matching leaf nodes. Hence, the similarity score of unmatching subtrees will be equal to “0”. For instance, consider T_1 and T_2 as depicted in Figures 1.8(a) and 1.8(b), respectively. Then, t_{14} and t_{126} are *unmatching subtrees*.

In order to minimize number of subtree comparisons, we define a *similarity score threshold* θ whose value is between 0 and 1. If $\mathcal{R}(t_1, t_2) < \theta$, then we assume that t_1 and t_2 are *unmatching*

CHAPTER 3. BACKGROUND

subtrees. Observe that by setting the value of θ , we are able to reduce the search space of finding matching subtrees. However, we may only get *non-optimal changes*. In Chapter 5, we shall see the effects of similarity score threshold θ to the performance and result quality of our approaches.

Next, based on the above concepts, the *best matching subtrees* are formally defined as follows.

Definition 3.7 [Best Matching Subtrees] Let T_1 and T_2 be two sets of subtrees that are in T_1 and T_2 , respectively. Let $t \in T_1$ be a subtree and $P \subseteq T_2$ be a set of subtrees. Also t and $t_i \in P$ are matching subtrees $\forall 0 < i \leq |P|$. Then, t and t_i are **best matching subtrees** (denoted by $t \approx t_i$) iff $(\mathcal{R}(t, t_i) > \mathcal{R}(t, t_j)) \forall 0 < j \leq |P|$ and $i \neq j$. $\square$

Example 3.10 [Best Matching Subtrees in Ordered XML] Consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. Subtree t_{14} can be matched to subtrees t_{102} and t_{108} . Observe that $\mathcal{R}(t_{14}, t_{102}) = 0.571$ and $\mathcal{R}(t_{14}, t_{108}) = 0.333$. According to Definition 3.7, subtrees t_{14} and t_{102} are best matching subtrees ($t_{14} \approx t_{102}$). $\square$

Example 3.11 [Best Matching Subtrees in Unordered XML] Now, we consider T_1 and T_2 as depicted in Figures 1.8(a) and 1.8(b), respectively. Subtree t_{25} can be matched to subtrees t_{117} and t_{126} . Then, subtrees t_{25} and t_{117} are best matching subtrees ($t_{25} \approx t_{117}$). This is because $\mathcal{R}(t_{25}, t_{117}) = 0.923$ and $\mathcal{R}(t_{25}, t_{126}) = 0.200$. $\square$

Note that if t_1 and t_2 are not best matching subtrees, then they are denoted by $t_1 \not\approx t_2$.

3.3.2 Types of Changes

In *unordered XML*, we have five types of changes, that is, *insertion of internal nodes*, *deletion of internal nodes*, *insertion of leaf nodes*, *deletion of leaf nodes*, and *content update*. We have six types of changes for *ordered XML* documents, namely, *insertion of internal nodes*, *deletion of internal nodes*, *insertion of leaf nodes*, *deletion of leaf nodes*, *content update*, and *move operations*. Furthermore, the content updates can be classified into two categories, that is, *absolute update* and *relative update*. The move operations are also classified into two types, namely, *move to different parent nodes* and *move among siblings*. In this section, we shall formally define these types of changes.

Insertion and Deletion of Internal Nodes

Intuitively, inserted internal nodes are ones that are in the new version (T_2), but not in the

CHAPTER 3. BACKGROUND

old version (T_1). An inserted internal node is the root node of an inserted subtree. Similarly, deleted internal nodes are ones that are in the old version (T_1), but not in the new version (T_2). A deleted internal node is the root node of a deleted subtree. Observe the inserted and deleted internal nodes must not be the root nodes of best matching subtrees as these nodes are in both versions. Formally,

Definition 3.8 [Inserted/Deleted Internal Node] Let n and m_x be the root nodes of subtrees t_n and t_{m_x} , respectively. Then,

- n is an *inserted internal node* iff $n \in \mathcal{I}(T_2)$ and $\forall m_x \in \mathcal{I}(T_1)$ such that $t_{m_x} \not\approx t_n$;
- n is a *deleted internal node* iff $n \in \mathcal{I}(T_1)$ and $\forall m_x \in \mathcal{I}(T_2)$ such that $t_{m_x} \not\approx t_n$. $\square$

Example 3.12 [Inserted/Deleted Internal Nodes in Ordered XML] Consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. We have three best matching subtrees at level 2 ($t_8 \approx t_{108}$, $t_{14} \approx t_{102}$ and $t_{19} \approx t_{118}$). Node 113 that is the root node of subtree t_{113} in T_2 is an *inserted internal node*. This is because $t_2 \not\approx t_{113}$, $t_8 \not\approx t_{113}$, $t_{14} \not\approx t_{113}$, and $t_{19} \not\approx t_{113}$. Node 2 that is the root node of subtree t_2 in T_1 is a *deleted internal node*. Observe that $t_2 \not\approx t_{102}$, $t_2 \not\approx t_{108}$, $t_2 \not\approx t_{113}$, and $t_2 \not\approx t_{118}$. $\square$

Example 3.13 [Inserted/Deleted Internal Nodes in Unordered XML] Suppose we have two versions of an unordered XML document, namely, T_1 and T_2 that are depicted in Figures 1.8(a) and 1.8(b), respectively. We observe that, at level 4, $t_6 \approx t_{106}$ and $t_{25} \approx t_{117}$. Node 126 that is the root node of subtree t_{126} in T_2 is an *inserted internal node*. This is because $t_6 \not\approx t_{126}$, $t_{14} \not\approx t_{126}$, and $t_{25} \not\approx t_{126}$. We also notice that node 14 that is the root node of subtree t_{14} in T_1 is a *deleted internal node* as $t_{14} \not\approx t_{106}$, $t_{14} \not\approx t_{117}$, and $t_{14} \not\approx t_{126}$. $\square$

Insertion and Deletion of Leaf Nodes

Intuitively, inserted leaf nodes are only available in the new version of an XML document. The inserted leaf nodes can be classified into two types as follows.

- **Inserted leaf nodes in the newly inserted subtrees.** These inserted leaf nodes must be the child nodes of inserted internal nodes. Note that inserted internal nodes are the root nodes of inserted subtrees. For example, consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. The leaf nodes with identifiers 114, 115, and 117 belong to the *inserted subtree* rooted at node 113.

- **Inserted leaf nodes in the best matching subtrees.** The parent nodes of these inserted leaf nodes are the root nodes of best matching subtrees. Note that best matching subtrees are available in the old and new versions of XML documents. For example, consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. The leaf node with identifier 107 is also inserted in the new version. The parent node of node 107 is node 105 which is matched to node 17 ($t_{17} \approx t_{105}$).

Similarly, deleted leaf nodes are only available in the old version of an XML document. The deleted leaf nodes can also be classified into two categories as follows.

- **Deleted leaf nodes in the deleted subtrees.** The parent nodes of these deleted leaf nodes are deleted internal nodes which are the root nodes of deleted subtrees. For example, consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. The leaf nodes with identifiers 3, 4, 6, and 7 belong to the *deleted subtree* rooted at node 2.
- **Deleted leaf nodes in the best matching subtrees.** The parent nodes of these deleted leaf nodes are the root nodes of best matching subtrees. For example, consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b), respectively. The leaf node with identifier 12 is also deleted. This leaf node is in the best matching subtrees $t_{11} \approx t_{111}$.

Formally,

Definition 3.9 [Inserted/Deleted Leaf Node] Let ℓ be a leaf node. Then,

- ℓ is an **inserted leaf node** iff $\ell \in \mathcal{L}(T_2)$ and the following conditions are satisfied:
 - if ℓ is in a **newly inserted subtree**, then $\text{parent}(\ell) = n_k$, where $n_k \in \mathcal{N}$ and $\mathcal{N}$ is a set of inserted internal nodes,
 - if ℓ is in a **best matching subtree**, then $\text{parent}(\ell) = i_2$, and $\forall a_x \in \mathcal{L}(T_1)$ such that $(a_x \not\sim \ell)$, where $\text{parent}(a_x) = i_1$ and $t_{i_1} \approx t_{i_2}$.
- ℓ is a **deleted leaf node** iff $\ell \in \mathcal{L}(T_1)$ and the following conditions are satisfied:
 - if ℓ is in a **deleted subtree**, then $\text{parent}(\ell) = d_k$, where $d_k \in \mathcal{D}$ and $\mathcal{D}$ is a set of deleted internal nodes,
 - if ℓ is in a **best matching subtree**, then $\text{parent}(\ell) = i_1$, and $\forall a_x \in \mathcal{L}(T_2)$ such that $(\ell \not\sim a_x)$, where $\text{parent}(a_x) = i_2$ and $t_{i_1} \approx t_{i_2}$. □

CHAPTER 3. BACKGROUND

Observe that an updated leaf node can be decomposed into a pair of deleted and inserted leaf nodes. By using Definition 3.9, the updated leaf nodes are detected as pairs of deleted and inserted leaf nodes.

Update Operation

Intuitively, an updated node is available in the old and new versions of an XML document, but its value is updated. Recall that updated leaf nodes are detected as pairs of deleted and inserted leaf nodes by using Definition 3.9. Hence, we are able to find the updated leaf nodes from two sets of leaf nodes: *inserted leaf nodes* and *deleted leaf nodes*.

Recall that, in ordered XML, we classify the updated leaf nodes into two types, that is, *absolute update operation* and *relative update operation*. Informally, an updated leaf node is classified as an *absolute updated leaf node* if its left-to-right position among its siblings is not changed. On the other hand, if the left-to-right position of an updated leaf node among its sibling is changed, then it is classified as a *relative updated leaf node*. In *unordered XML*, we do not need to classify the update leaf nodes further as the left-to-right position of a node among its siblings is not significant. Formally, an updated leaf node is defined as follows.

Definition 3.10 [Updated Leaf Node] Let $\mathcal{Y}$ and $\mathcal{E}$ be two sets of inserted and deleted leaf nodes, respectively. Let u be an **updated leaf node** decomposed into $e \in \mathcal{E}$ and $y \in \mathcal{Y}$. Node u is an updated leaf node iff $\text{name}(e) = \text{name}(y)$, $\text{path}(e) = \text{path}(y)$, $\text{level}(e) = \text{level}(y)$, $\text{value}(e) \neq \text{value}(y)$, and $\text{parent}(e) \approx \text{parent}(y)$.

Definition 3.11 [Absolute/Relative Updated Leaf Node] Let T_1 and T_2 are ordered XML. Let u be an updated leaf node decomposed into $e \in \mathcal{E}$ and $y \in \mathcal{Y}$, where $\mathcal{E} \subset \mathcal{L}(T_1)$ and $\mathcal{Y} \subset \mathcal{L}(T_2)$. Then,

- node u is an **absolute updated leaf node** iff $\text{pos}(e) = \text{pos}(y)$;
- node u is a **relative updated leaf node** iff $\text{pos}(e) \neq \text{pos}(y)$. □

Example 3.14 [Updates in Ordered XML] Suppose we have two versions of an ordered XML document, that is, T_1 and T_2 that are depicted in Figures 1.3(a) and 1.3(b), respectively. Subtrees t_{14} and t_{102} are best matching subtrees. Node 16 is updated from “Assoc Prof” to “Prof”. Node 16 is classified as an *absolute updated leaf node*. Note that in our running example, we do not have relative updated leaf nodes. □

Example 3.15 [Updates in Unordered XML] Consider T_1 and T_2 as depicted in Figures 1.8(a) and 1.8(b), respectively. Subtrees t_6 and t_{106} are best matching subtrees. Node 7 is updated from “*Web Mining*” to “*Web Usage Mining*”. $\square$

Move Operation

Move operation is a type of changes that only available in ordered XML. It is classified into two categories. First, a moved node changes its position among its siblings in the XML tree. That is, before and after the move operation, it has *same* parent but different position among its siblings. Second, the node (subtree) is moved to be the child of a different parent. That is, before and after the move operation, it has *different* parents, and may have different position among its siblings. These two move operations are formally defined as follows.

Definition 3.12 [Moved Internal Node] Let t_1 and t_2 be two subtrees rooted at node $i_1 \in \mathcal{D}$ and node $i_2 \in \mathcal{N}$, respectively. Subtree t_1 is moved if $t_1 \approx t_2$ and satisfies:

- if t_1 is moved among its siblings, then $\text{pos}(i_1) \neq \text{pos}(i_2)$, and $\text{parent}(i_1) \approx \text{parent}(i_2)$;
- if t_1 is moved to different parent node, then $\text{parent}(i_1) \neq \text{parent}(i_2)$. $\square$

The same intuitions as in Definition 3.12 is used to define move operations of leaf nodes.

Definition 3.13 [Moved Leaf Node] Let ℓ_1 and ℓ_2 be two leaf nodes where $\ell_1 \in \mathcal{E}$ and $\ell_2 \in \mathcal{Y}$, respectively. ℓ_1 is moved if $\ell_1 \leftrightarrow \ell_2$ and satisfies:

- if ℓ_1 is moved among its siblings, then $\text{pos}(\ell_1) \neq \text{pos}(\ell_2)$, and $\text{parent}(\ell_1) \approx \text{parent}(\ell_2)$;
- if ℓ_1 is moved to different parent node, then $\text{parent}(\ell_1) \neq \text{parent}(\ell_2)$. $\square$

3.4 Schema for Storing XML Documents

In this section, we shall elaborate on the SUCXENT schema [PBM04b] and the *Shared-Inlining* approach [STZ⁺99].

3.4.1 SUCXENT

In [PBM04b], Prakash et al. proposed a schema-oblivious approach for storing XML documents called SUCXENT. SUCXENT efficiently stores and queries XML data in relational database by storing only the leaf nodes and their ancestors. SUCXENT allows effective evaluation of queries without involving θ -joins. Figure 2.11 depicts the relational schema of SUCXENT. Basically, SUCXENT schema consists of four tables: the **Document** table, the **Path** table, the **LeafValue** table, and the **AncestorInfo** table. The **Document** table is used to enable SUCXENT to store multiple versions of an XML document. The **Path** table stores the path expression of nodes.

The **LeafValue** table is used for storing the information on leaf nodes. The *Doc.ID* attribute indicates which XML document a particular leaf node belongs to. The *Path.ID* attribute maintains the id of the path of a particular leaf node stored in the **Path** table. The *LeafOrder* attribute is used to record the node order of the leaf nodes in an XML tree. Consider an XML document depicted in Figure 1.2(a). When we parse the XML document, we find out that the leaf node “*name*” with value “*Smith*” is the first leaf node in the document. Hence, we assign the *LeafOrder* equal to “1” for this leaf node. The next leaf node is node “*rank*” with value “*Assoc Prof*”. Therefore, the *LeafOrder* of this node is equal to “2”. Two leaf nodes have the same *SiblingOrder* if they share the same parent. For example, the leaf nodes with *LeafOrder* equal to “1” and “2” shall have the same *SiblingOrder* (equal to “1”) since they share the same parent node (node “*staff*” with node id 2). The *LeftSibInLevel* (Left Sibling Intersection Level) is a level of the nearest common ancestor node between the leaf nodes belonging to a particular sibling order s and the ones belonging to the sibling order $s - 1$. For example, consider the leaf nodes with *SiblingOrder* equal to “1” and “2” in the XML tree. The nearest common ancestor node of these nodes is node “*staff*” (id=2) which is at level 2. The *LeafValue* stores the textual content of the leaf nodes. Note that the *LeftSibInLevel* attribute in this table is only useful for constructing the XML documents from the relational database.

The **AncestorInfo** table is used for storing the ancestor information for each leaf node. The *Doc.ID* attribute indicates to which XML document a particular ancestor node belongs to. We record the names and the level of ancestor nodes in the *NodeName* and *NodeLevel* attributes, respectively. The *MinSibOrder* and *MaxSibOrder* store the minimum and maximum sibling orders of the leaf nodes under a particular ancestor node, respectively. For

CHAPTER 3. BACKGROUND

AncestorInfo					Path	
Doc_ID	Node Level	MinSib Order	MaxSib Order	NodeName	Path_ID	Path_Exp
1	1	1	8	univStaff	1	./univStaff./staff./name
1	2	1	2	staff	2	./univStaff./staff./rank
1	3	2	2	research	3	./univStaff./staff./research./interest
1	2	3	4	staff		
1	3	4	4	research		
...	...	...	...	...		
1	3	8	8	research		

Document	
Doc_ID	Doc_Name
1	Staff-01.xml

LeafValue						
Doc_ID	Leaf Order	Path_ID	Level	Sibling Order	LeafValue	LeftSib lxnLevel
1	1	1	3	1	Smith	-1
1	2	2	3	1	Assoc Prof	-1
1	3	3	5	2	Web Mining	2
1	4	3	5	2	Multimedia Mining	2
...	...	...	...	...	...	...

Figure 3.2: SUCXENT: Example.

example, node “staff” (id=2) in Figure 1.2(a) has *MinSibOrder* and *MaxSibOrder* equal to “1” and “2”, respectively. Node “univStaff” (id=1) has *MinSibOrder* and *MaxSibOrder* equal to “1” and “8”, respectively.

Suppose we have an XML document as shown in Figure 1.2(a). Figure 3.2 depicts the XML document shredded in relational database by using SUCXENT schema (partial view only). We now formally define an ancestor node (w.r.t the SUCXENT schema) as follows.

Definition 3.14 [Ancestor Node] Let $minsiborder(i)$ and $maxsiborder(i)$ be the minimum sibling order and maximum sibling order of an internal node i respectively. Node i is an **ancestor node** of a node n (denoted by $ancestor(n)$) iff $level(n) > level(i)$, and satisfies:

- if n is a **leaf node**, then $minsiborder(i) \leq siborder(n) \leq maxsiborder(i)$.
- if n is an **internal node**, then $minsiborder(i) \leq minsiborder(n)$ and $maxsiborder(i) \geq maxsiborder(n)$. □

And a parent node (w.r.t the SUCXENT schema) is formally defined as follows.

Definition 3.15 [Parent Node] Node i is a **parent node** of a node n (denoted by $parent(n)$) iff $i = ancestor(n)$ and $level(n) = level(i) + 1$. □

Figure 3.3: Shared-Inlining: DTD Transformation.

3.4.2 The Inlining Approach

In [STZ⁺99], the authors proposed three techniques for storing XML in RDBMS: *the Basic Inlining technique*, *the Shared Inlining technique*, and *the Hybrid Inlining technique*. The proposed techniques need the existence of DTD of the XML documents. The first step of the techniques is simplifying DTD. The transformations of DTD are of three types as follows: *flattening transformation*, *simplification transformation*, and *grouping transformation*. Figure 3.3 depicts these transformations. Next, the DTD elements are grouped and converted into relations. The inlining techniques inlines as many descendants of an element as possible into their ancestors to reduce number of relations. As a result, the fragmentation of data is minimized and the cost of joining relations during query processing is reduced.

In the Basic Inlining technique, the relations are created for every element. This is based on an assumption that an XML document can be rooted at any element in DTD. In this technique, a DTD is simplified by discarding the information on the order of occurrence among elements. Thus, the simplified DTD preserves only the semantics of child elements concerned with whether the element (a) can occur only once or more times and (b) is mandatory or not. The graph based on the simplified information is called a *DTD graph*. Given an element graph, relations are created as follows. A relation is created for the root element of the graph. All the elements descendants are inlined into that relation with the following two exceptions: (a) children directly below a “*” node are made into separate relations this corresponds to creating a new relation for a set-valued child; and (b) each node having a backpointer edge pointing to it is made into a separate relation this corresponds to creating a new relation to handle recursion.

The idea of the Shared Inlining technique is to identify the element nodes that are represented in multiple relations in the Basic Inlining technique and to share them by creating separate relation for there elements. In the Shared Inlining technique, relations are created for all elements in the DTD graph and the nodes of which have an in-degree

Figure 3.4: Shared-Inlining: DTD Graph.

Cinema (CinemaID:integer, Cinema.Name:string, Cinema.Address:string)
Name (NameID:integer, Name:string)
Address (AddressID:integer, Address:string)
Theater (TheaterID:integer, Theater.Movie:string)
Cinema.Theater (Cinema.TheaterID:integer, Cinema.Theater.parentID:integer, Cinema.Theater.TID:string, Cinema.Theater.Capacity:string, Cinema.Theater.Movie:string)
Movie (MovieID:integer, Movie:string)
ShowTimes (ShowTimesID:integer, ShowTimes.parentID:integer)
ShowTimes.ShowTime (ShowTimes.ShowTimeID:integer, ShowTimes.ShowTime.parentID:integer, ShowTimes.ShowTime.ShowTime:string)
ShowTime (ShowTimeID:integer, ShowTime.STID:string, ShowTime:string)

Figure 3.5: The Basic Inlining Schema.

greater than one. Nodes with an in-degree of one are inlined in the parent nodes relation. For each element node with an in-degree of zero, a separate relation is created because they are not reachable from any other node. In the DTD graph, edges marked with “*” indicate that the element of a destination node can occur more than once. For each such element, a separate relation is created because relational databases cannot store set values as they are. Finally, element nodes, which appear along with the directed paths from the element in the DTD graph that creates the relational schema (R), are also inlined as an attribute in the relational schema R . However, the directed paths must not contain “*”.

In the Hybrid Inlining technique, elements with in-degrees greater than 2 are also inlined if they are reachable without passing “*”. Incidentally, order information among elements that is discarded in the first step can be represented by adding positional information in the relational schema. The schema of Hybrid Inlining technique may be the same as one of Shared Inlining technique.

CHAPTER 3. BACKGROUND

Figure 3.6: The Shared and Hybrid Inlining Schema.

Suppose we have a DTD as depicted in Figure 3.4(a). This DTD will be transformed into the DTD as shown in Figure 3.4(b). If we use the Basic Inlining technique, then we shall have a relational schema as depicted in Figure 3.5. The Shared and Hybrid Inlining schemas are depicted in Figure 3.6.

Chapter 4

Detecting the Changes to Ordered XML

In the previous chapter, we have presented the overview of our approaches and the terminology that will be used in this dissertation. We also have briefly discussed the relational schemas that are used as the underlying storage structure for XANDY and HELIOS.

The variants of XANDY and HELIOS for detecting changes to ordered XML documents are called XANDY-O [LB] and HELIOS-O [LB05b], respectively. Furthermore, XANDY-O has two different approaches based on how the best matching subtrees are determined. They are the *top-down* approach and the *bottom-up* approach. In the top-down approach, the best matching subtrees are determined in top-down fashion. That is, we start from the root nodes and try to match the subtrees in the deeper level. Note that the top-down approach uses the greedy approximation. In the bottom-up approach, we try to match the subtrees in bottom-up fashion. We start from subtrees at the deepest level, and propagate the matchings up to the root nodes. In this chapter, we shall elaborate further on our approaches, namely, XANDY-O and HELIOS-O. Note that the preliminary works of these approaches can be found in [LB, LB05b, LBDM04].

This chapter is organized as follows. In Section 4.1, we shall present how we extend SUCXENT and *Shared-Inlining* schemas. Section 4.2 shall discuss how XANDY-O determines the best matching subtrees (Phase 1). We shall elaborate on how HELIOS-O finds the best matching subtrees (Phase 1) in Section 4.3. In Section 4.4, we shall discuss how XANDY-O detects the changes to *ordered* XML documents (Phase 2) after determining the best matching subtrees. Section 4.5 shall present the second phase of HELIOS-O (the *change detection* phase). Finally, we shall summarize the discussion in this chapter in Section 5.12. Note that the performance study of XANDY-O and HELIOS-O will be presented in Chapter 5.

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Document (<u>Doc_ID</u> , Doc_Name)	Document (<u>Doc_ID</u> , Doc_Name)
Path (<u>Path_ID</u> , Path_Exp)	Path (<u>Path_ID</u> , Path_Exp)
LeafValue (<u>Doc_ID</u> , <u>LeafOrder</u> , <u>Path_ID</u> , LeftSibIxLevel, SiblingOrder, LeafValue, LocalOrder , Level , Dewey)	LeafValue (<u>Doc_ID</u> , <u>LeafOrder</u> , <u>Path_ID</u> , LeftSibIxLevel, SiblingOrder, LeafValue, LocalOrder , Level)
AncestorInfo (<u>Doc_ID</u> , <u>MinSibOrder</u> , <u>MaxSibOrder</u> , <u>NodeLevel</u> , <u>NodeName</u> , LocalOrder , Dewey)	AncestorInfo (<u>Doc_ID</u> , <u>MinSibOrder</u> , <u>MaxSibOrder</u> , <u>NodeLevel</u> , <u>NodeName</u> , LocalOrder)

(a) Top-down Approach

(b) Bottom-up Approach

Figure 4.1: Modified SUCXENT Schema for XANDY-O.

AncestorInfo							LeafValue								
Doc_ID	Node Level	MinSib Order	MaxSib Order	NodeName	Local Order	Dewey	Doc_ID	Leaf Order	Path_ID	Level	Sibling Order	LeafValue	LeftSib IxnLevel	Local Order	Dewey
1	1	1	8	univStaff	1	1	1	1	1	3	1	Smith	-1	1	1.1.1
1	2	1	2	staff	1	1.1	1	2	2	3	1	Assoc Prof	-1	2	1.1.2
1	3	2	2	research	3	1.1.3	1	3	3	5	2	Web Mining	2	1	1.1.3.1
1	2	3	4	staff	2	1.2	1	4	3	5	2	Multimedia Mining	2	2	1.1.3.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1	3	8	8	research	3	1.4.3	2	1	1	3	1	Mark	-1	1	1.1.1
2	1	1	8	univStaff	1	1	2	2	2	3	1	Prof	-1	2	1.1.2
2	2	1	2	staff	1	1.1	2	3	3	5	2	Digital Libraries	2	1	1.1.3.1
2	3	2	2	research	3	1.1.3	2	4	3	5	2	Information Retrieval	2	2	1.1.3.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2	3	8	8	research	3	1.4.3	2	14	3	5	8	Security	2	2	1.4.3.2

Document

Doc_ID	Doc_Name
1	Staff-01.xml
2	Staff-02.xml

Path

Path_ID	Path_Exp
1	/univStaff/staff/name
2	/univStaff/staff/rank
3	/univStaff/staff/research/interest

Figure 4.2: Top-down Approach: XML in RDBMS.

4.1 The Relational Schema Extensions

In this section, we shall present the extensions of SUCXENT and *Shared-Inlining* schemas. Note that XANDY-O and HELIOS-O use SUCXENT and *Shared-Inlining* schemas as their underlying relational schemas, respectively. The extensions are necessary as the properties of existing relational schemas are not enough to be used for detecting the changes to ordered XML documents.

4.1.1 XANDY-O

In this section, we shall present the relational schema extensions for both approaches in XANDY-O.

The Top-down Approach

For the top-down approach, SUCXENT schema is modified as follows. The *LocalOrder* and *Dewey* attributes are added in the *LeafValue* and *AncestorInfo* tables to store the position of a node among its siblings and ancestors' local orders of each node, respectively. This

LeafValue								AncestorInfo					
Doc_ID	Leaf Order	Path_ID	Level	Sibling Order	LeafValue	LeftSib	Local Order	Doc_ID	Node Level	MinSib Order	MaxSib Order	NodeName	Local Order
1	1	1	3	1	Smith	-1	1	1	1	1	8	univStaff	1
1	2	2	3	1	Assoc Prof	-1	2	1	2	1	2	staff	1
1	3	3	5	2	Web Mining	2	1	1	3	2	2	research	3
1	4	3	5	2	Multimedia Mining	2	2	1	2	3	4	staff	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
2	1	1	3	1	Mark	-1	1	1	3	8	8	research	3
2	2	2	3	1	Prof	-1	2	2	1	1	8	univStaff	1
2	3	3	5	2	Digital Libraries	2	1	2	2	1	2	staff	1
2	4	3	5	2	Information Retrieval	2	2	2	3	2	2	research	3
...	...	...	...	...	...	...	...	...	...	...	...	...	...
2	14	3	5	8	Security	2	2	2	3	8	8	research	3

Path	
Path_ID	Path_Exp
1	/univStaff/staff/name
2	/univStaff/staff/rank
3	/univStaff/staff/research/interest

Document	
Doc_ID	Doc_Name
1	Staff-01.xml
2	Staff-02.xml

Figure 4.3: Bottom-up Approach: XML in RDBMS.

Dewey attribute is adopted from the Dewey Ordering Encoding [Cen]. The local order is assigned in an incremental manner among the siblings from left to right. We also add the *Level* attribute in the **LeafValue** table to store the level of leaf nodes. The extended SUCXENT schema for the top-down approach is shown in Figure 4.1(a).

In [TVB⁺02], the authors discussed that Dewey Ordering Encoding does not work properly in some cases. For example, when we compare “1.2” and “10.2”, “10.2” lexicographically occurs before “1.2”. In our case, we only compare whether the value of *Dewey* attribute of a tuple is equal to the one of other tuples. Hence, the top-down approach does not suffer from the above limitation.

Consider two versions of an ordered XML document as depicted in Figures 1.2(a) and 1.2(b). Figure 4.2 depicts these XML documents shredded in relational database (partial view only). Note that the *Doc.ID* attribute is used to distinguish different versions of XML documents.

The Bottom-up Approach

We extend SUCXENT schema for the bottom-up approach as follows. We add the *Level* and *LocalOrder* attributes in the **LeafValue** table to store the level and the position among siblings of leaf nodes, respectively. The **AncestorInfo** table is extended by adding the *LocalOrder* attribute that is used to store the positions among siblings of internal nodes. We do not use the *Dewey* attribute in this approach for the following reason. Our approach determines the best matching subtrees at level *level* + 1 before finding the best matching subtrees at level *level*. That is, the matching subtrees at level greater than *level* are already

Figure 4.4: DTD Tree, DTD, and Relational Schema.

determined. Hence, we just need to use the information on the best matching subtrees at level $level + 1$ in order to find the best matching subtrees at level $level$. The extended SUCXENT schema for the bottom-up approach is depicted in Figure 4.1(b).

Consider two versions of an ordered XML document as depicted in Figures 1.2(a) and 1.2(b). Figure 4.3 depicts these XML documents shredded in relational database (partial view only).

4.1.2 HELIOS-O

Assume that D_1 and D_2 depicted in Figures 1.2(a) and 1.2(b), respectively, are accompanied with a DTD as depicted in Figure 4.4(b). The DTD tree of this DTD is shown in Figure 4.4(a). Given a DTD depicted in Figure 4.4(b), Shared-Inlining approach generates a relational schema as depicted in Figure 4.4(c). In [STZ⁺99], the authors did not specifically specify the attribute for storing the local order of nodes that is important in ordered XML documents. Therefore, we need to extend the relational schema generated by Shared-Inlining approach.

Before we discuss the extensions, let us present some notations that will be used in later discussion. Given a DTD tree $\mathcal{H}_U$ that is tree representation of DTD U , the nodes in $\mathcal{H}_U$ are classified as follows.

- **Inlined Nodes.** An *inlined node* is one that is not below “*” or “+”. There are two types of inlined nodes, namely, *inlined leaf nodes* (denoted by $\mathbb{I}_\ell$) and *inlined internal nodes* (denoted by $\mathbb{I}_i$). For example, consider a DTD tree as depicted in Figure 4.4(a). Nodes *name* and *research* are an inlined leaf node and an inlined internal node, respectively. An inlined node will be stored as an attribute in the

Document

Doc_ID	Doc_Name
1	staff1.xml
2	staff2.xml

UnivStaff

Doc_ID	ID
1	1
2	2

Interest

Doc_ID	ID	LocalOrder	PID	Val
1	1	1	1	Web Mining
1	2	2	1	Multimedia Mining
1	3	1	2	Data Mining
1	4	2	2	Web Data Management
1	5	1	3	Digital Libraries
1	6	1	4	Indexing
1	7	2	4	Security
2	1	1	1	Digital Libraries
2	2	2	1	Information Retrieval
2	4	1	2	Web Data Management
2	5	1	3	Semantic Web
2	6	1	4	Indexing
2	7	2	4	Security

Staff

Doc_ID	ID	LocalOrder	PID	Name	Rank	Research
1	1	1	1	Smith	Assoc Prof	True
1	2	2	1	Chan	Assoc Prof	True
1	3	3	1	Mark	Assoc Prof	True
1	4	4	1	Don	Asst Prof	True
2	1	1	2	Mark	Prof	True
2	2	2	2	Chan	Assoc Prof	True
2	3	3	2	Steve	Asst Prof	True
2	4	4	2	Don	Asst Prof	True


```

Input:
  document id of the first version did1,
  document id of the second version did2,
  similarity score threshold  $\theta$ 
Output:
  the MATCHING table

1  root1 = getRootNode(did1)
2  root2 = getRootNode(did2)
3  IF (root1.name != root2.name) THEN RETURN
4  score = calculateScore(root1, root2)
5  IF (score == 1.0) THEN
6    Store the root node information in the MATCHING table
7    RETURN
8  ELSE IF (score <  $\theta$ ) THEN RETURN
9  Store the root node information in the MATCHING table
10 findBestMatchingSubtree(did1, did2,  $\theta$ , root1, root2);
11 extendIdenticalSubtrees(did1, did2);

```

Figure 4.6: Top-down Approach: The *rootMatching* Algorithm.

4.2.1 The Top-down Approach

In this section, we shall discuss how our top-down approach finds the best matching subtrees. First, we shall give an overview of the algorithm. Whenever the algorithm invokes SQL queries, we shall elaborate on the intuitions behind the SQL queries. Finally, we shall give an example on how the best matching subtrees are determined using our top-down approach.

Root Node Matching

After two versions of an XML document, D_1 and D_2 , are stored in relational database, we invoke the *rootMatching* algorithm as shown in Figure 4.6. There are three input parameters for the *rootMatching* algorithm, namely, the document ids of the first and second versions of an XML document stored in relational database (*did1* and *did2*, respectively), and the similarity score threshold (θ). Observe that the *rootMatching* algorithm will return a set of best matching subtrees stored in the MATCHING table.

The first step of finding best matching subtrees in the top-down approach is to retrieve the root nodes of T_1 and T_2 . The intuition of the SQL query for retrieving a root node of an XML document is to find a tuple in the **AncestorInfo** whose *NodeLevel* attribute is equal to “1”. The next step is to compare the root nodes of T_1 and T_2 . If T_1 has a different node name from T_2 , then we assume that T_1 and T_2 are different trees. Consequently, the delta shall consist of a deletion of T_1 and an insertion of T_2 . Otherwise, we calculate the similarity score of T_1 and T_2 ($\mathcal{R}(T_1, T_2)$) by using Definition 3.6. We shall discuss later how the similarity score is calculated. If $\mathcal{R}(T_1, T_2) = 1$, then it means that T_1 and T_2 are identical. Hence, we store the information about the root nodes of T_1 and T_2 in the


```

Input:
document id of the first version did1,
document id of the second version did2,
parent node in the first version r1,
parent node in the second version r2,
threshold theta

Output:
the MATCHING table

1  list1 = getInternalChildNode(did1, r1); /* SQL in Fig.4.8(a) */
2  list2 = getInternalChildNode(did2, r2); /* SQL in Fig.4.8(a) */
3  IF (list1 is empty or list2 is empty) THEN RETURN
   /* Matching the child internal nodes */
4  FOR (x=0; x<sizeof(list1); x++)
5      maxScore = 0.0; Z = -1;
6      FOR (y=0; y<sizeof(list2); y++)
7          IF ((list1[x].name != list2[y].name) OR
              (list2[y] is already matched)) THEN BREAK;
              /* Calculating similarity score */
8          score = calculateScore(list1[x], list2[y]);
9          IF (score == 1) THEN /* Identical subtrees */
10             maxScore = score; Z = y;
11             BREAK;
12         ELSE IF ((score >= theta) AND
                   (score > maxScore)) THEN
13             maxScore = score; Z = y;
14         END IF
15     END FOR
16     IF (maxScore != 0.0) AND (Z != -1) THEN
17         Store (list1[x], list2[Z]) in the MATCHING table;
18         Mark list2[Z] that it has been matched;
19         IF (maxScore < 1.00) THEN
20             /* Match the internal nodes at next level */
21             findBestMatchingSubtrees(did1, did2, theta, list1[x],
                                     list2[Z]);
22         END IF
23     END IF
24 END FOR

```

Figure 4.7: Top-down Approach: The *findBestMatchingSubtrees* Algorithm.

MATCHING table. If $\mathcal{R}(T_1, T_2) < \theta$, then we assume that T_1 and T_2 are different. Hence, the XDelta will be a deletion of T_1 and an insertion of T_2 . If $\theta \leq \mathcal{R}(T_1, T_2) < 1$, then T_1 and T_2 are matching subtrees. Hence, we propagate the matchings in the top-down fashion by invoking a recursive algorithm called the *findBestMatchingSubtrees* algorithm as depicted in Figure 4.7.

Finding Best Matching Subtrees

The *findBestMatchingSubtrees* algorithm works as follows. Suppose we have two root nodes of two subtrees, r_1 and r_2 , where $r_1 \in \mathcal{I}(T_1)$, $r_2 \in \mathcal{I}(T_2)$, $\text{name}(r_1) = \text{name}(r_2)$, and $\text{level}(r_1) = \text{level}(r_2)$. The first step is to retrieve two sets of internal nodes (lines 1–2, Figure 4.7), $I_1 \subset \mathcal{I}(T_1)$ and $I_2 \subset \mathcal{I}(T_2)$, where $\forall i_{1x} \in I_1$ such that $r_1 = \text{parent}(i_{1x})$ and $\forall i_{2y} \in I_2$ such that $r_2 = \text{parent}(i_{2y})$. The SQL query for retrieving the child internal nodes of a given internal node r is depicted in Figure 4.8(a). Observe that “*minso*”, “*maxso*”, and “*level*” are replaced by the minimum sibling order, maximum sibling order, and level

of node r , respectively. If $I_1 = \emptyset$ or $I_2 = \emptyset$, then the algorithm considers that subtree rooted at r_1 is not matched to the one rooted at r_2 (line 3, Figure 4.7). Otherwise, the algorithm starts to match $i_{1x} \in I_1$ and $i_{2y} \in I_2$. As the matching internal nodes must have the same node name, the algorithm checks whether $\text{name}(i_{1x}) = \text{name}(i_{2y})$ and i_{2y} has no matching internal node (line 7, Figure 4.7). If only $\text{name}(i_{1x}) = \text{name}(i_{2y})$ and i_{2y} has no matching internal nodes, then the algorithm computes $\mathcal{R}(i_{1x}, i_{2y})$ (line 8, Figure 4.7). We shall discuss later how the similarity score is calculated. There are two possibilities of $\mathcal{R}(i_{1x}, i_{2y})$ as follows.

- $\mathcal{R}(i_{1x}, i_{2y}) = 1$. In this case, subtree rooted at i_{1x} is identical to the one rooted at i_{2y} . The algorithm sets the values of maxScore and Pos (line 9, Figure 4.7) and stops comparing i_{1x} to other internal node in I_2 .
- $\theta \leq \mathcal{R}(i_{1x}, i_{2y}) < 1$. In this case, i_{1x} and i_{2y} are matching internal nodes. The next step is to check whether i_{1x} and i_{2y} are best matching internal nodes by comparing the value of $\mathcal{R}(i_{1x}, i_{2y})$ to maxScore . If $\mathcal{R}(i_{1x}, i_{2y}) > \text{maxScore}$, then the algorithm sets the values of maxScore and Z (line 13, Figure 4.7) and continues comparing i_{1x} to other internal node in I_2 .

Next, if i_{1x} has best matching internal node in I_2 , then the algorithm stores the information of i_{1x} and i_{1Z} into the **MATCHING** table. If only $\mathcal{R}(i_{1x}, i_{2y}) < 1$, then the algorithm propagates the matching to the next level by invoking the *findBestMatchingSubtrees* algorithm recursively (line 20, Figure 4.7). Note that at this point of time, $\theta \leq \mathcal{R}(i_{1x}, i_{2y}) \leq 1$.

Calculating Similarity Score

Given two internal nodes, r_1 and r_2 , $\mathcal{R}(r_1, r_2)$ is calculated as follows. First, we have to determine the numbers of *fixed* and *shifted* matching leaf nodes in the subtrees rooted at r_1 and r_2 . Recall that the *fixed* and *shifted* matching leaf nodes are defined in Definition 3.3. The SQL query for calculating the number of *fixed* matching leaf nodes is depicted in Figure 4.8(b). Suppose ℓ_1 and ℓ_2 are the descendant leaf nodes of r_1 and r_2 , respectively. Lines 5–7 are used to ensure that ℓ_1 and ℓ_2 have the same paths, same local orders, and same node value. Lines 8–11 are used to ensure that ℓ_1 and ℓ_2 are the descendant leaf nodes of r_1 and r_2 , respectively. Observe that the sibling order of a leaf node will be between the minimum and maximum sibling orders of its ancestor nodes. The prefix of dewey value of

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.8: Top-down Approach: SQL Queries (1).

a leaf node must be the dewey values of its ancestor nodes. For instance, consider node 6 in T_1 as depicted in Figure 1.3(a). The sibling order of node 6 is “2” (the LeafValue table, Figure 4.2). Node 6 is the descendent node of node 2 as sibling order node 2 is between the minimum and maximum sibling orders of node 2, that is, “1” and “2”, respectively. Similarly, the prefix of dewey value of node 6 (“1.1.3.2”) is the dewey value of node 2 (“1.1”). Line 12 is used to make sure that ℓ_1 and ℓ_2 have the same relative position in the first and second versions of an XML tree, respectively. For example, suppose we compare node 14 in T_1 and node 102 in T_2 . We find out that $\ell_{18} \leftrightarrow \ell_{106}$ that is one of the matching leaf nodes in subtrees rooted at nodes 14 and 102 are the *fixed* matching leaf nodes. Observe that ℓ_{18} and ℓ_{106} have the same relative positions to nodes 14 and 102, respectively. The dewey values of nodes 14 and 102 are “1.3” and “1.1”, respectively. The dewey values of ℓ_{18} and ℓ_{106} are “1.3.3.1” and “1.1.3.1”, respectively. The condition in line 12 (Figure 4.8(b)) will be true (“3.1” = “3.1”). Suppose we want to calculate the number of fixed matching leaf nodes in t_{14} and t_{102} . Figure 4.9 depicts the intermediate result of the SQL query in Figure 4.8(b).

The next step is to calculate the numbers of *shifted* matching leaf nodes in the subtree rooted at r_1 and r_2 . The intuition behind calculating the numbers of shifted matching leaf nodes is similar to the one behind the numbers of fixed matching leaf nodes. The SQL query

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.9: Intermediate Result of SQL Query in Figure 4.8(b) .

in Figure 4.8(c) is used to calculate number of the *shifted* matching leaf nodes. Line 11 is used to make sure that the relative positions of ℓ_1 and ℓ_2 have been changed. Lines 12-26 are used to filter out the matching leaf nodes that have been considered as *fixed* matching leaf nodes.

Next, we calculate numbers of leaf nodes in the subtrees rooted at r_1 and r_2 . We use the SQL query that is depicted in Figure 4.8(d) to calculate the numbers of leaf nodes in a subtree. The “*minso*” and “*maxso*” are replaced by the minimum and maximum sibling orders of an internal node, respectively. For example, suppose we calculate number of leaf nodes in the subtree rooted at node 8. Then, “*minso*” and “*maxso*” are replaced by “3” and “4”. Finally, we are able to calculate $\mathcal{R}(r_1, r_2)$ by using Definition 3.6. Note that $|A|$ and $|B|$ are calculated by using the SQL queries depicted in Figures 4.8(b) and 4.8(c), respectively. The values of $|t_1|$ and $|t_2|$ are computed by using the SQL query in Figure 4.8(d).

Mapping Identical Subtrees

After we have identified the matching subtrees in the top-down fashion, we need to determine the identical subtrees that are in the detected identical subtrees (line 11, Figure 4.6). This is because as following. When we match r_1 and r_2 and find out that r_1 and r_2 are the root

```

1  SELECT
2    M.DID1, M.DID2, A1.MINSIBORDER, A1.MAXSIBORDER,
3    A2.MINSIBORDER, A2.MAXSIBORDER, A1.NODELEVEL,
4    A1.LOCALORDER, A2.LOCALORDER,
5    1.000 AS SCORE
6  FROM MATCHING AS M, ANCESTORINFO AS A1,
7    ANCESTORINFO AS A2
8  WHERE
9    M.DID1 = did1 AND M.DID2 = did2 AND
10   M.SCORE = 1.000 AND
11   A1.DOC_ID = did1 AND A2.DOC_ID = did2 AND
12   A1.NODENAME = A2.NODENAME AND
13   A1.NODELEVEL = A2.NODELEVEL AND
14   A1.LOCALORDER = A2.LOCALORDER AND
15   A1.MINSIBORDER >= M.MINSO1 AND
16   A1.MAXSIBORDER <= M.MAXSO1 AND
17   A2.MINSIBORDER >= M.MINSO2 AND
18   A2.MAXSIBORDER <= M.MAXSO2

```

Figure 4.10: Top-down Approach: SQL Query for Mapping Identical Subtrees.

nodes of identical subtrees, we do not match the internal nodes that are the descendant nodes of r_1 and r_2 further. Hence, we need to map the identical subtrees that are rooted at the descendant nodes of r_1 and r_2 . Note that the subtrees in the identical trees must be identical subtrees. For example, subtrees rooted at nodes 19 and 118 are identical subtrees. The top-down approach only stores the information of $t_{19} \approx t_{118}$. Therefore, it is necessary to map all the identical subtrees in t_{19} and t_{118} (in our case $t_{22} \approx t_{121}$ and store in the **MATCHING** table. The SQL query for mapping all the subtrees in the identical subtrees is depicted in Figure 4.10. The intuition is to find the internal nodes that are the descendant nodes of the root nodes of identical subtrees. Lines 12–14 are used to make sure that the descendant internal nodes have the same node name and local order, and are at the same level. Lines 15–18 are used to capture ancestor-descendant relationships between the root nodes of identical subtrees and their descendant nodes.

The *findBestMatchingSubtrees* algorithm in the top-down approach (Figure 4.7) is a greedy approximation. Once the algorithm determines that subtree rooted at node p in the old version document is matched to subtree rooted at node q in the new version, it will not compare the subtree rooted at node p to subtrees in the old version document nor the subtree rooted at node q to subtrees in the new version document. This greedy approximation may lead the algorithm to result *non-optimal* delta in some cases. If we do not use the greedy approximation, then the algorithm does $|N| \times |M|$ subtrees comparisons (similarity score calculations) for each level, where $|N|$ and $|M|$ are the numbers of internal nodes that are the child nodes of matching subtrees in the old and new versions of an XML


```

Input:
  did1 : document id of first version of document
  did2 : document id of second version of document
  theta : similarity score threshold
Output:
  the MATCHING table

1 IF (!isRootNodeMatched(did1, did2)) THEN return;
2 findMatchingSiblingOrder(did1, did2);
3 maxLevel = getInternalNodeMaxLevel(did1, did2);
  // Bottom-up matching
4 FOR (curLevel=maxLevel; curLevel>=1; curLevel--)
5   findPMatchingIntNodes(did1,did2,curLevel);
6   maximizeSimilarityScore(did1,did2,curLevel);
7   deleteUnMatchingNodes(did1, did2, curLevel, theta);
8 END FOR
9 populatingBestMatchingSubtrees(did1,did2);

```

Figure 4.11: Bottom-up Approach: The *findBestMatchingSubtrees* Algorithm.

tree, respectively. This leads to the slower performance of the algorithm. Hence, we trade off the result quality for better performance. To address the limitations of the top-down approach, we propose the bottom-up approach.

4.2.2 The Bottom-up Approach

In this section, we shall discuss how the bottom-up approach of XANDY-O finds the best matching subtrees. The bottom-up approach works as follows.

Suppose we have two versions of an XML document, namely, D_1 and D_2 , stored in relational database. The tree representations of D_1 and D_2 are T_1 and T_2 , respectively. The first step in the bottom-up approach is to compare the root nodes of T_1 and T_2 . If their node names are different, then we assume that T_1 and T_2 are different. Consequently, the XDelta will consist of a deletion of T_1 and an insertion of T_2 . Otherwise, the bottom-up approach shall determine the *matching leaf nodes groups*. Note that, in XANDY, the matching leaf nodes groups are also known as *matching sibling orders*. This is because the leaf nodes in a matching leaf node group have the same parent node. In SUCXENT schema, leaf nodes that have the same parent node must have the same sibling order.

Having determined the matching sibling orders, the bottom-up approach starts propagating the matchings in bottom-up fashion. For each level, the bottom-up approach finds the matching subtrees. As a subtree in T_1 can be matched to more than one subtree in T_2 , the bottom-up approach needs to determine the best matching subtrees for a set of matching subtrees. Finally, the bottom-up approach populates the best matching subtrees.

Input: *did1* and *did2*
 Output: the TempSO table

```

1 findFixedMatchingLV(did1, did2); /* Fig.4.13(a) */
2 findShiftedMatchingLV(did1, did2); /* Fig.4.13(b) */
3 findMatchingSOFromFixedMatchingLV(did1, did2); /* Fig.4.13(c) */
4 findMatchingSOFromShiftedMatchingLV(did1, did2); /* Fig.4.13(d) */
5 findMatchingSO(did1, did2); /* Fig.4.13(e) and 4.13(f) */

```

Figure 4.12: Bottom-up Approach: The *findMatchingSiblingOrder* Algorithm.

Figure 4.13: Bottom-up Approach: SQL Queries for Finding Matching Sibling Orders.

Figure 4.11 depicts the *findBestMatchingSubtrees* algorithm in the bottom-up approach. We shall elaborate further on each step.

Finding Matching Sibling Orders

The matching sibling orders can be determined by using the *findMatchingSiblingOrder* algorithm as depicted in Figure 4.12. Let us elaborate further on each phase.

The first step is to find the *fixed* and *shifted* matching leaf nodes (lines 1-2, Figure 4.12). The SQL queries to find the *fixed* and *shifted* matching leaf nodes are depicted in Figures 4.13(a) and 4.13(b), respectively. The intuition of finding *fixed* matching leaf nodes is based on Definition 3.3. The fixed matching leaf nodes have the *same paths* (line 11, Figure 4.13(a)), the *same node values* (line 12, Figure 4.13(a)), and the *same local orders* (line 13, Figure 4.13(a)). Note that the node name and node level of a node are encapsulated in its path. For example, consider T_1 and T_2 as depicted in Figures 1.3(a) and 1.3(b).

Given the **LeafValue** table as depicted in Figure 4.3, the result of SQL query depicted in Figure 4.13(a) is shown in Figure 4.15(a).

Similarly, the intuition of finding *shifted* matching leaf nodes is based on Definition 3.3. The shifted matching leaf nodes have the *same paths* (line 8, Figure 4.13(b)) and the *same node values* (line 8, Figure 4.13(b)), but have *different local orders* (line 9, Figure 4.13(b)). Lines 10–12 in Figure 4.13(b) are used to guarantee that a shifted matching leaf node from the first version has not been assigned as a fixed matching leaf node. Similarly, lines 13–14 in Figure 4.13(b) are used to guarantee that a shifted matching leaf node from the second version has not been determined as a fixed matching leaf node. For instance, given the **LeafValue** table as depicted in Figure 4.3 and the **FIXEDLV** table as depicted in Figure 4.15(a), the result of SQL query depicted in Figure 4.13(b) is shown in Figure 4.15(b). The intermediate results of SQL Query in Figure 4.13(b) is shown in Figure 4.14.

The next step is to determine the matching sibling orders from two sets of *fixed* and *shifted* matching leaf nodes (lines 3-4, Figure 4.12). Figure 4.13(e) depicts the SQL query that is used to find matching sibling orders that have both *fixed* and *shifted* matching leaf nodes. Note that “*TMSO1*” and “*TMSO2*” can be replaced by SQL queries depicted in Figures 4.13(c) and 4.13(d), respectively. The intuition behind the SQL query depicted in Figure 4.13(c) is to find matching sibling orders from a set of *fixed* matching leaf nodes. It is possible by grouping the *fixed* matching leaf nodes (line 5) and counting how many fixed matching leaf nodes are in matching sibling orders (line 3). Note that we need to multiply the number of fixed leaf nodes are in matching sibling orders by two (Definition 3.6). For instance, given the **FIXEDLV** table as depicted in Figure 4.13(a), the result of SQL query in Figure 4.13(c) is shown in Figure 4.15(c). The SQL query in Figure 4.13(d) is used to find matching sibling orders from a set of *shifted* matching leaf nodes. The intuition behind the SQL query in Figure 4.13(d) is similar to the one behind the SQL query in Figure 4.13(c). For example, given the **SHIFTLV** table as depicted in Figure 4.13(b), the result of SQL query depicted in Figure 4.13(d) is shown in Figure 4.15(d).

In some cases, the matching sibling orders only have either *fixed* or *shifted* matching leaf nodes. Therefore, we also need to determine such matching sibling orders (line 5, Figure 4.12). Figure 4.13(f) depicts the SQL query that is used to find matching sibling orders that only have *fixed* matching leaf nodes. Lines 5–8 are used to make sure that matching sibling orders that have both *fixed* and *shifted* matching leaf nodes are discarded.

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.14: Intermediate Result of SQL Query in Figure 4.13(b).

To find matching sibling orders that only have *shifted* matching leaf nodes, we use SQL query in Figure 4.13(f) after slight modifications. It is modified as follows. We replace “TMS01” in line 4 with “TMS02”. The matching sibling orders are stored in the TempS0 table. The semantics of attributes of the TempS0 table are depicted in Figures 4.16(a) and 4.16(b). For instance, given the FIXEDLV (Figure 4.15(a)) and SHIFTLV (Figure 4.15(b)) tables, the result of SQL queries depicted in Figures 4.13(e) and 4.13(f) is the TempS0 table as shown in Figure 4.15(e). Observe that the value of *Total* attribute is still zero. Hence, we need to calculate the value of *Total* attribute by using SQL query depicted in Figure 4.13(g). Figure 4.17(a) depicts the TempS0 table after the value of *Total* attribute is calculated.

Bottom-up Matching

The next step is to propagate the matchings in bottom-up fashion. First, we determine

Figure 4.15: Bottom-up Approach: Example of Finding Matching Sibling Orders.

Figure 4.16: Table Descriptions.

the deepest level $maxLevel$ of internal nodes in the XML documents (line 3, Figure 4.11). The deepest level $maxLevel$ can be found by querying the **AncestorInfo** table. Having determined the deepest level $maxLevel$, the next step is to find the best matching subtrees starting from level $maxLevel$ to level 1 (the level of root node). For each $curLevel$, the algorithm starts finding the matching internal nodes (line 5, Figure 4.11). Then, the algorithm determines the best matching subtrees by finding the *best matching configurations* (line 6, Figure 4.11). Finally, we prune the *unmatching subtrees* whose similarity scores are less than similarity score threshold θ (line 7, Figure 4.11). We shall elaborate on each step in turn.

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.17: The TempSO table and SQL Query for Finding Matching Internal Nodes.

Finding Matching Internal Nodes. The SQL query in Figure 4.17(b) is used to retrieve the root nodes of matching subtrees at level *curLevel* based on Definition 3.5. The matching internal nodes must have the same node name (line 12, Figure 4.17(b)), the same node level (line 13, Figure 4.17(b)), and at least have one matching leaf node (lines 10–11, Figure 4.17(b)). Note that we use the matching sibling orders. This is because, in matching sibling orders, there is at least one matching leaf nodes. We join the *AncestorInfo* and *TempSO* tables in which the internal nodes and matching sibling orders are kept, respectively. Figure 4.18 depicts the intermediate result of SQL query depicted in Figure 4.17(b) (when *level* = 2). Then, we update the value of the *Total* attribute because the value of the *Total* attribute is still “0”. Finally, we update the value of the *Score* attribute. The value of the *Score* attribute is equal to $\frac{X}{Y}$, where *X* is the value of the *Counter* attribute and *Y* is the value of the *Total* attribute.

The matching internal nodes detected by using SQL query depicted in Figure 4.17 are kept in the *TempMatching* table. The semantics of the attributes of the *TempMatching* table are depicted in Figures 4.16(a) and 4.16(b). The *Flag* attribute of the *TempMatching* table is initially set to “0”. The usage of the *Flag* attribute shall be discussed later. Figures 4.19(a), 4.19(b), and 4.19(c) depict the *TempMatching* table when *curLevel* = 3, *curLevel* = 2, and *curLevel* = 1, respectively.

Maximizing Similarity Score. As a subtree in T_1 can be matched to more than one subtree in T_2 and vice versa, we need to determine the *best matching configuration* at level *curLevel* such that the similarity scores of subtrees at level (*curLevel* – 1) are maximized. Let us elaborate on this by using an example.

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Result

Level	MinSO1	MaxSO1	MinSO2	MaxSO2	LO1	LO2	Counter	Total	Score
2	1	2	3	4	1	2	2	4	0
2	3	4	3	4	2	2	5	7	0
2	5	6	1	2	3	1	4	7	0
2	5	6	3	4	3	2	2	4	0
2	7	8	5	6	4	3	2	4	0
2	7	8	7	8	4	4	8	8	0

Projection and Grouping

A1.Node Level	A1.MinSib Order	A1.MaxSib Order	A1.Node Name	A1.Local Order	A2.Node Level	A2.MinSib Order	A2.MaxSib Order	A2.Node Name	A2.Local Order	T.Level	T.SO1	T.SO2	T.Counter	T.Total
2	1	2	staff	1	2	3	4	staff	2	3	1	3	2	4
2	3	4	staff	2	2	3	4	staff	2	3	3	3	4	4
2	3	4	staff	2	2	3	4	staff	2	4	4	4	1	3
2	5	6	staff	3	2	1	2	staff	1	3	5	1	2	4
2	5	6	staff	3	2	1	2	staff	1	4	6	2	2	3
2	5	6	staff	3	2	3	4	staff	2	3	5	3	2	4
2	7	8	staff	4	2	5	6	staff	3	3	7	5	2	4
2	7	8	staff	4	2	7	8	staff	4	3	7	7	4	4
2	7	8	staff	4	2	7	8	staff	4	4	8	8	4	4

A1.Node Level	A1.MinSib Order	A1.MaxSib Order	A1.Node Name	A1.Local Order	T.Level	T.SO1	T.SO2	T.Counter	T.Total
2	1	2	staff	1	3	1	3	2	4
2	3	4	staff	2	3	3	3	4	4
2	3	4	staff	2	4	4	4	1	3
2	5	6	staff	3	3	5	1	2	4
2	5	6	staff	3	4	6	2	2	3
2	5	6	staff	3	3	5	3	2	4
2	7	8	staff	4	3	7	5	2	4
2	7	8	staff	4	3	7	7	4	4
2	7	8	staff	4	4	8	8	4	4

Join2

T.SO2 BETWEEN A2.MINSIBORDER AND
A2.MAXSIBORDER AND
A1.NODENAME = A2.NODENAME AND
A1.NODELEVEL = A2.NODELEVEL

Node Level	MinSib Order	MaxSib Order	NodeName	Local Order
1	1	8	univStaff	1
2	1	2	staff	1
3	2	2	research	3
2	3	4	staff	2
3	4	4	research	3
2	5	6	staff	3
3	6	6	research	3
2	7	8	staff	4
3	8	8	research	3

Join1

T.SO1 BETWEEN A1.MINSIBORDER AND A1.MAXSIBORDER

Node Level	MinSib Order	MaxSib Order	NodeName	Local Order
2	1	2	staff	1
2	3	4	staff	2
2	5	6	staff	3
2	7	8	staff	4

Level	SO1	SO2	Counter	Total
3	1	3	2	4
3	5	3	2	4
3	7	5	2	4
3	3	3	4	4
3	5	1	2	4
4	6	2	2	3
3	7	7	4	4
4	8	8	4	4
4	4	4	1	3

A1 (AncestorInfo table)

A1.DOC ID = did1 AND
A1.NODELEVEL = level

A2.DID2 = did2

A2 (AncestorInfo table)

Figure 4.18: Intermediate Results of SQL Query Depicted in Figure 4.17.

Suppose $curLevel = 2$ and the information on matching internal nodes at level 2 is depicted in Figure 4.19(b). The matching internal nodes information can be modeled as bipartite graph (Figure 4.20(a)). Note that the values in the circles are node ids of the nodes in T_1 (Figures 1.3(a)) and T_2 (Figures 1.3(b)). From the graph, we notice that $t_2 \simeq t_{108}$, $t_8 \simeq t_{108}$, $t_{14} \simeq t_{103}$, $t_{14} \simeq t_{108}$, $t_{19} \simeq t_{113}$, and $t_{19} \simeq t_{118}$. Furthermore, we observe that we can have six *matching configurations* as depicted in Figures 4.20(b)–4.20(g). Each matching configuration will affect the similarity score $\mathcal{R}(t_1, t_{101})$. Figure 4.20(h) depicts $\mathcal{R}(t_1, t_{101})$ for different matching configurations. We notice that the sixth matching configuration (Figure 4.20(h)) is able to maximize $\mathcal{R}(t_1, t_{101})$. That is, the sixth matching configuration is *best matching configuration*.

Level	MinSO1	MaxSO1	MinSO2	MaxSO2	LO1	LO2	Counter	Total	Score	Flag
3	4	4	4	4	3	3	1	3	0.3333	0
3	6	6	2	2	3	3	2	3	0.6667	0
3	8	8	8	8	3	3	4	4	1.0000	0

(a) Level 3

Level	MinSO1	MaxSO1	MinSO2	MaxSO2	LO1	LO2	Counter	Total	Score	Flag
3	4	4	4	4	3	3	1	3	0.3333	0
3	6	6	2	2	3	3	2	3	0.6667	0
3	8	8	8	8	3	3	4	4	1.0000	0
2	1	2	3	4	1	2	2	7	0.2857	0
2	3	4	3	4	2	2	5	7	0.7142	0
2	5	6	1	2	3	1	4	7	0.5714	0
2	5	6	3	4	3	2	2	6	0.3333	0
2	7	8	5	6	4	3	2	7	0.2857	0
2	7	8	7	8	4	4	8	8	1.0000	0

(b) Level 2

Level	MinSO1	MaxSO1	MinSO2	MaxSO2	LO1	LO2	Counter	Total	Score	Flag
3	4	4	4	4	3	3	1	3	0.3333	0
3	6	6	2	2	3	3	2	3	0.6667	0
3	8	8	8	8	3	3	4	4	1.0000	0
2	3	4	3	4	2	2	5	7	0.7142	0
2	5	6	1	2	3	1	4	7	0.5714	0
2	7	8	7	8	4	4	8	8	1.0000	0
1	1	8	1	8	1	1	17	29	0.5862	0

(c) Level 1

Figure 4.19: The TempMatching table.

The problem of finding best matching configuration is similar to the problem of finding *maximum weighted bipartite matching*. Hence, we are able to solve the problem of finding best matching configuration by using the algorithm for finding maximum weighted bipartite matching. In our implementation, we use the Hungarian method [PS82]. Having determined the *best matching configurations*, the algorithm annotates the root nodes of matching subtrees at level *curLevel* whose parent nodes are not used in the best matching configuration by setting the *Flag* attribute in the TempMatching table to “1”. The annotations mean that these subtrees may be moved to different parent nodes.

Deleting Unmatching Subtrees. The next step is to delete the matching internal nodes that have similarity scores less than similarity score threshold θ . Recall that such matching subtrees are considered as *unmatching subtrees*. The algorithm continues to find the best matching subtrees at level $(curLevel - 1)$ iteratively until $curLevel = 1$.

Populating Best Matching Subtrees

Having determined all best matching subtrees, the algorithm populates the best matching subtrees from the TempMatching table in top-down fashion. The best matching subtrees in T_1 and T_2 are stored in the MATCHING table. Note that the TempMatching table also stores

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.20: Example: Maximizing Similarity Score.

DID1	DID2	Level	Min SO1	Max SO1	Min SO2	Max SO2	LO1	LO2	Score
1	2	3	4	4	4	4	3	3	0.3333
1	2	3	6	6	2	2	3	3	0.6666
1	2	3	8	8	8	8	3	3	1.0000
1	2	2	3	4	3	4	2	2	0.7142
1	2	2	5	6	1	2	3	1	0.5714
1	2	2	7	8	7	8	4	4	1.0000
1	2	1	1	8	1	8	1	1	0.5862

Figure 4.21: Bottom-up Approach: The MATCHING Table.

the corresponding tuples of the root nodes of the subtrees that are suspected as moved subtrees. The semantics of the attributes of the `MATCHING` table are Figures 4.16(a) and 4.16(b). The `MATCHING` table storing the best matching subtrees of our example is depicted in Figure 4.21.

The *bottom-up approach* is able to find the best matching subtrees in T_1 and T_2 . Intuitively, a set S is maximized if all subsets s_i of S are maximized. That is, if subtrees at level $l + 1$ have maximum similarity score, then subtrees at level l will also have maximum similarity score. Having found the best matching subtrees, we are ready to detect the changes. We shall discuss how the changes are detected by XANDY-O in Section 4.4.

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

<p>Input U : DTD of the XML documents Two versions of an XML document stored in RDBMS</p> <p>Output the Matching table</p> <pre> /* --- STEP 1 --- */ 1 for all ℓ in $N_\ell(U)$ do 2 $tbName \leftarrow r_\ell$; $tempTb \leftarrow M_\ell$; 3 findMatchingLeafNodesGroups($tbName$, $tempTb$); 4 end for /* --- STEP 2 --- */ 5 $maxLevel$ = maximum level at which there is i in $\mathbb{I}_i(U)$ /* bottom-up matching */ 6 for $lev = maxLevel$ down to 1 do 7 for all $i \in \mathbb{I}_i(U)$ at level lev do 8 $childNode \leftarrow child(i)$; 9 $tempMChild \leftarrow M_{childNode}$; 10 $tbName \leftarrow r_i$; $tempTb \leftarrow M_i$; </pre>	<pre> /* --- STEP 2.1 --- */ 11 findMatchingInternalNodes($tbName$, $tempTb$, $tempMChild$); /* --- STEP 2.2 --- */ 12 maximizeScore(ω_i); 13 end for 14 end for /* --- STEP 3 --- */ 15 root is the root node of U 16 Queue $Q \leftarrow \{root\}$ 17 while (Q is not empty) do 18 $q = Q.get()$; 19 $Q \leftarrow$ the child internal nodes of q in U; 20 $nodeName \leftarrow name(q)$; $tempTb \leftarrow r_q$; 21 $parentNode \leftarrow parent(q)$; 22 $parentNodeName \leftarrow name(parentNode)$; 23 $attrName \leftarrow attribute(q)$; 24 retrieveMatching($nodeName$, $tempTb$, $parentNodeName$, $attrName$); 25 end do </pre>
---	---

Figure 4.22: HELIOS-O: The *findBestMatchingSubtrees* Algorithm.

<p>Input: $tbName$ and $tempTb$ Output: the matching leaf nodes</p> <pre> 1 findFixedMatchingLeafNodes ($tbName$); /* Fig.4.24(a) */ 2 findMatchingGroup-Fixed ($tbName$); /* Fig.4.24(b) */ 3 findMatchingGroup-Shifted ($tbName$); /* Fig.4.24(c) */ 4 findMatchingGroup ($tbName$, $tempTb$); /* Fig.4.24(d) */ 5 updateTotal ($tbName$, $tempTb$); /* Fig.4.24(e) */ 6 updateCounter ($tempTb$); /* Fig.4.24(f) */ 7 updateSimilarityScore ($tempTb$); /* Fig.4.24(g) */ </pre>

Figure 4.23: HELIOS-O: The *findMatchingLeafNodesGroups* Algorithm.

4.3 Finding Best Matching Subtrees Phase in HELIOS-O

In this section, we shall elaborate on how the best matching subtrees are determined in HELIOS-O.

The *findBestMatchingSubtrees* algorithm is depicted in Figure 4.22. Observed that it consists of three phases, namely, the *finding matching leaf nodes groups* phase, the *bottom-up matching* phase, and the *collecting best matching internal nodes* phase. The objective of this phase is to find the matching non-inlined leaf node groups as defined in Definition 3.4. Note that the inlined leaf nodes will be matched when we match internal nodes (Phase 2). This is because if we match inlined leaf nodes separately, then we shall query the relations that store the information of internal nodes twice, that is, once when we match inlined leaf nodes and another one when internal nodes are matched. In Phase 2, HELIOS-O propagates

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.24: HELIOS-O: SQL Queries for Finding Matching Leaf Nodes.

the matching in bottom-up fashion. Finally, HELIOS-O merges/collects the best matching internal nodes from different relations and to determine the best matching inlined internal nodes. We shall elaborate on each phase in turn. Note that “[param]” in the SQL queries used in the later discussion will be replaced the parameter *param* defined in the algorithm.

4.3.1 Finding Matching Leaf Nodes Groups Phase

The *findMatchingLeafNodesGroups* algorithm as depicted in Figure 4.23 is used to determine the *matching leaf nodes groups*. The algorithm works as follows.

First, the *findMatchingLeafNodesGroups* algorithm determines the *fixed matching leaf nodes* (line 1, Figure 4.23) by using SQL query in Figure 4.24(a). This SQL query is based on Definition 3.3. Lines 9–10 are used to ensure that fixed matching leaf nodes have the same values and local orders. Next, we determine the *matching leaf nodes groups* from a set of fixed matching leaf nodes (line 2, Figure 4.23). The SQL query in Figure 4.24(b) is used

to determine matching leaf nodes groups from a set of fixed matching leaf nodes. The idea behind of this SQL query is simple. That is, we just group the fixed matching leaf nodes by their *PID1* and *PID2* attributes (line 4, Figure 4.24(b)). Observe that the *PID1* and *PID2* attributes store the parent node id of fixed matching leaf nodes in the old and new versions, respectively.

The next step is to determine *matching leaf nodes groups* from *shifted matching leaf nodes* (line 3, Figure 4.23). We use the SQL query that is depicted in Figure 4.24(c) to determine the matching leaf nodes groups from shifted matching leaf nodes. Line 6 is to ensure that two matching leaf nodes have the same values. Lines 7–9 are used to filter out leaf nodes in the *old version* that already have been matched when the algorithm finds the fixed matching leaf nodes. Similarly, lines 10–12 are used to filter out the ones in the *new version*. Finally, the shifted matching leaf nodes are grouped by their *PID1* and *PID2* attributes. Suppose we want to determine the matching leaf node groups from *Interest* node. Then, the intermediate result of SQL Query in Figure 4.24(c) is depicted in Figure 4.25.

At this point of time, we have two sets of matching leaf nodes groups, that is, one from fixed matching leaf nodes and another from shifted matching leaf nodes. The next step is to merge these sets of matching leaf nodes groups (line 4, Figure 4.23). Figure 4.24(d) depicts the SQL query to merge two sets of matching leaf nodes groups. We only need to use “UNION” operator (line 6) to merge these sets. The final step is to update the information of matching leaf nodes groups (lines 5–7, Figure 4.23). We update the values of the *Total*, *Counter*, and *Score* attributes. This is because we do not have information on the values of these attributes when we merge two sets of matching leaf nodes groups. We use three SQL queries as depicted in Figures 4.24(e)–4.24(g).

Suppose we have two set of leaf nodes, $\mathcal{G}_1$ and $\mathcal{G}_2$, whose parent nodes are i_1 and i_2 , respectively, where $\mathcal{G}_1 \Leftrightarrow \mathcal{G}_2$. The value of the *Total* attribute is equal to $(|t_1| + |t_2|)$, where $|t_1|$ and $|t_2|$ are the numbers of leaf nodes whose parent nodes are i_1 and i_2 , respectively. That is, lines 3–6 and lines 7–10 in Figure 4.24(e) are used to calculate the values of $|t_1|$ and $|t_2|$, respectively. The value of the *Counter* attribute is equal to $(2|A| + |B|)$, where $|A|$ and $|B|$ are the numbers of fixed and shifted matching leaf nodes in $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively. That is, lines 3–5 and lines 7–9 in Figure 4.24(f) are used to calculate the values of $2|A|$ and $2|B|$, respectively. Finally, the value of the *Score* attribute is equal to the value of the *Counter* attribute divided by the value of the *Total* attribute. That is, the value of the *Score* attribute is $\frac{2|A|+|B|}{|t_1|+|t_2|}$ as defined in Definition 3.6.

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.25: Intermediate Result of SQL Query in Figure 4.24(c).

The results of the *findMatchingLeafNodesGroups* algorithm are temporary tables M_{c_x} in which the information of matching groups of non-inlined leaf nodes c_x are stored. The schema of the M_{c_x} table is the same as the one of the `tempTb1` table as depicted in Figure 4.27(a). The semantics of attributes of the `tempTb1` table are depicted in Figure 4.27(b). For instance, in our example, we have the “interest” node as non-inlined leaf nodes. The *findMatchingLeafNodesGroups* algorithm will result $M_{interest}$ as depicted in Figure 4.26(a).

4.3.2 Bottom-up Matching Phase

The next step is to propagate the matchings in bottom-up fashion (lines 5–14, Figure 4.22). First, the algorithm determines the highest level of the non-inlined internal nodes in DTD U (line 5). Then, the algorithm starts to find best matching internal nodes in bottom-up

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.26: HELIOS-O: Temporary Matching Tables and the Matching Table.

Figure 4.27: HELIOS-O: Temporary and Delta Table Descriptions.

fashion. There are two sub steps, that is, finding matching internal nodes (line 11) and determining best matching subtrees (line 12) by finding *best matching configurations*. We shall elaborate on each step in turn.

Finding Matching Internal Nodes

For each non-inlined internal node $i_w \in N_i$, the *findBestMatchingSubtrees* algorithm invokes the *findMatchingInternalNodes* algorithm (line 11, Figure 4.22) after it defines three parameters (lines 9–10, Figure 4.22) that will be the input parameters of the *findMatchingInternalNodes* algorithm.

The *findMatchingInternalNodes* algorithm executes the SQL query as depicted in Figure 4.28(a). This SQL query is used to find matching internal nodes from the temporary matching tables of child nodes of i_w . The intuition is to group the matching nodes according to their parent nodes (line 18). Line 12 is used to capture the parent-child relationship between matching leaf nodes and their parent nodes. Lines 13–17 are used to remove duplicates of parent nodes if the parent nodes have more than one non-inlined matching child node.

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

<pre> 1 INSERT INTO [tempTb] 2 SELECT 3 A1.DOC_ID AS DID1, A2.DOC_ID AS DID2, 4 A1.ID AS ID1, A2.ID AS ID2, 5 A1.LOCALORDER AS LO1, A2.LOCALORDER AS LO2, 6 A1.PID AS PID1, A2.PID AS PID2, 7 0 AS COUNTER, 0 AS TOTAL, 0 AS SCORE, 0 AS FLAG 8 FROM [tempMChild] AS A, 9 [tbName] AS A1, [tbName] AS A2 10 WHERE 11 A.DID1 = doc_id1 AND A.DID2 = doc_id2 AND 12 A1.DOC_ID = doc_id1 AND A2.DOC_ID = doc_id2 AND 13 A1.ID = A.PID1 AND A2.ID = A.PID2 AND 14 NOT EXISTS 15 (SELECT ID1, ID2 FROM [tempTb] AS B 16 WHERE B.DID1 = doc_id1 AND 17 B.DID2 = doc_id2 AND 18 B.ID1 = A.ID1 AND B.ID2 = A.ID2) 19 GROUP BY A1.DOC_ID, A2.DOC_ID, A1.PID, A2.PID, 20 A1.ID, A2.ID </pre>	<pre> 1 INSERT INTO [tempTb] 2 SELECT 3 A1.DOC_ID AS DID1, A2.DOC_ID AS DID2, 4 A1.ID AS ID1, A2.ID AS ID2, 5 A1.LOCALORDER AS LO1, A2.LOCALORDER AS LO2, 6 A1.PID AS PID1, A1.PID AS PID2, 7 0 AS COUNTER, 0 AS TOTAL, 0 AS SCORE, 0 AS FLAG 8 FROM [tbName] AS A1, [tbName] AS A2 9 WHERE 10 A1.DOC_ID = doc_id1 AND 11 A2.DOC_ID = doc_id2 AND 12 [moreConditions] AND 13 NOT EXISTS 14 (SELECT ID1, ID2 FROM [tempTb] AS B 15 WHERE B.DID1 = doc_id1 AND 16 B.DID2 = doc_id2 AND 17 B.ID1 = A.ID1 AND B.ID2 = A.ID2) 18 GROUP BY A1.DOC_ID, A2.DOC_ID, A1.PID, A2.PID, 19 A1.ID, A2.ID </pre>
(a) Finding Matching Internal Nodes (1)	(b) Finding Matching Internal Nodes (2)

Figure 4.28: HELIOS-O: SQL Queries for Finding Matching Internal Nodes.

For example, the SQL query shall determine the matching “univStaff” nodes from the temporary matching relation of node “staff” (the M_{staff} table). Suppose we want to determine the matching node “staff”. Then, the intermediate result of SQL Query in Figure 4.24(a) is depicted in Figure 4.29.

If node d_x has inlined child leaf nodes, then the *findMatchingInternalNodes* algorithm also executes the SQL query depicted in Figure 4.28(b). This SQL query is necessary because, in some cases, it is possible that i_w only has matching inlined child leaf nodes. For example, consider T_3 and T_4 as depicted in Figures 4.30(a) and 4.30(b), respectively. The SQL query in Figure 4.28(a) shall return $t_1 \simeq t_{12}$ as they have the matching “interest” nodes. If we do not execute the SQL query depicted in Figure 4.28(b), then we miss $t_1 \simeq t_{11}$. In fact, t_1 and t_{11} are best matching subtrees ($t_1 \approx t_{11}$). Therefore, the SQL query depicted in Figure 4.28(b) shall facilitate us to find “missing” matching non-inlined internal nodes.

Before executing the SQL query in Figure 4.28(b), the algorithm defines one more parameter, namely, *moreConditions* that is used to replace a part of the SQL query (line 12). This objective of this parameter is to match the inlined leaf nodes. Note that in the previous phase we only match non-inlined leaf nodes. Formally, the value of *moreConditions* is as follows. Suppose we have two internal nodes i_1 and i_2 from the old and new versions of XML documents, respectively. Nodes i_1 and i_2 are matching internal nodes ($i_1 \simeq i_2$) if $\exists a_{1_z} \exists a_{2_z}$ such that $value(a_{1_z}) = value(a_{2_z})$, where $a_{1_z} = child(i_1)$, $a_{2_z} = child(i_2)$, and $attribute(a_{1_z}) = attribute(a_{2_z})$. For example, given the “Staff” table, the value of parameter *moreConditions* is equal to “(A1.NAME = A2.NAME OR A1.RANK = A2.RANK)”. The result of the SQL query in Figure 4.28(b) is also stored in a temporary matching table M_{i_w} . For example,

Figure 4.29: Intermediate Result of SQL Query in Figure 4.28(a).

the matching “staff” nodes will be kept in the M_{staff} table as depicted in Figure 4.26(b). Similarly, the matching “univStaff” nodes will be stored in the $M_{univStaff}$ table as depicted in Figure 4.26(c). Suppose we want to determine the matching node “staff”. Then, the intermediate result of SQL Query in Figure 4.24(b) is depicted in Figure 4.31.

Finding Best Matching Internal Nodes

The task in this step is to find *best matching configurations* that facilitate us to find best matching internal nodes. This phase is important as an internal node in the first version of an XML document can be matched to more than one internal node in the second version, and vice versa. The best matching internal nodes is defined in Definition 3.7. Observe that the problem of finding best matching configuration is similar to the problem of finding *maximum weighted bipartite matching*. In our implementation, we use the Hungarian method [PS82] that addresses the problem of finding maximum weighted bipartite matching. The algorithm for finding best matching configurations is similar to the one discussed in Section 4.2.2. After we determine the best matching configurations, the algorithm annotates the matching

Figure 4.30: HELIOS-O: Example.

internal nodes whose parent nodes are not used in the best matching configuration by setting the *Flag* attribute in the M_{i_w} table to “1”. The annotations mean that these subtrees may be moved to different parent nodes.

Observe that we also need to update the values of the *Counter*, *Total*, and *Score* attributes accordingly as initially their values are equal to “0”. Intuitively, the value of *Counter* attribute is equal to $(p + \sum q_z)$, where p is the number of matching inlined leaf nodes, and q_z is the number of matching non-inlined leaf nodes retrieved from the temporary matching table. The value of *Total* attribute is equal to $(2k + \sum l_z)$, where k is the total number of inlined leaf nodes, and l_z the total number of non-inlined leaf nodes retrieved from the temporary matching table.

4.3.3 Collecting Best Matching Internal Nodes Phase

The result of the previous step is the best matching internal nodes partitioned in several relations. The objectives of this step are to merge/collect the best matching internal nodes from different relations and to determine the best matching inlined internal nodes. Observe that the moved subtree candidates are also in the temporary matching tables. The values of the *Flag* attribute of moved subtree candidates in the temporary matching tables are equal to “1”. Lines 15–25 in Figure 4.22 depict the algorithm for collecting the best matching internal nodes. The intuition behind this algorithm is to collect best matching internal nodes in top-down fashion, starting from the root nodes to the highest level of matching internal nodes. To retrieve the best matching internal nodes in each level, the algorithm invokes the *retrieveMatching* algorithm (line 24).

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.31: Intermediate Result of SQL Query in Figure 4.28(b).

Given an internal node x as the input, the *retrieveMatching* algorithm works as follows. First, the algorithm checks whether node x is a root node, an inlined internal node, or a non-inlined internal node. If node x is a root node, then the algorithm executes the SQL query depicted in Figure 4.32(a). The intuition of this SQL query is to retrieve a tuple that stores information on the matching root nodes from the M_x table. Observe that there will only be one tuple in the M_x table as there is only one root node from each version.

If node x is a non-inlined internal node, then the algorithm executes the SQL query depicted in Figure 4.32(b). The intuition behind the SQL query in Figure 4.32(b) is to find best matching node x in the M_x table whose parent nodes are best matching node in the **MATCHING** table. Line 10 is used to capture the parent-child relationships between best matching node x and the parent nodes that are best matching node in the **MATCHING** table. Line 11 is used to guarantee that the best matching nodes are not moved node candidates. Note that if the value of *FLAG* attribute is “1”, then the matching nodes of corresponding tuples are moved node candidates.

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.32: HELIOS-O: SQL Queries for Collecting Best Matching Internal Nodes.

If node x is an inlined internal node, the algorithm executes the SQL query depicted in Figure 4.32(c). The intuition behind this SQL query is as follows. An inlined internal node is stored as an attribute in its parent node's relation. Hence, we are able to find the best matching inlined internal node from the **MATCHING** table and the relations that store the parent nodes. We need to query the parent node relations because we have to check that the matching inlined internal node must be available in both versions. That is, the values of the attributes storing the information on inlined internal nodes must be "TRUE" (lines 18–19, Figure 4.32(c)). Line 17 is used to describe that the parent nodes of the inlined internal nodes must be in the **MATCHING** table.

Given the **M_univStaff** and **M_staff** tables (Figures 4.26(b) and 4.26(c), respectively) and the relations in which XML documents are shredded (Figure 4.5), the *retrieveMatching* algorithm results the **MATCHING** table as depicted in Figure 4.26(d). The semantics of the **MATCHING** table is depicted in Figure 4.27. The **MATCHING** table keeps the best matching internal nodes of two XML documents that will be used to detect the changes (Phase 2). Having found the best matching subtrees, we are ready to detect the changes. We shall discuss how the changes are detected by HELIOS-O in Section 4.5.

4.4 Detecting the Changes Phase in XANDY-O

In this section we shall elaborate on how XANDY-O detects the changes. We use the best matching subtrees determined in Phase 1 to detect the changes. Note that the types of

Input: <i>did1</i> and <i>did2</i> Output: XDelta tables <pre> 1 detectInsertedInternalNodes (<i>did1</i>, <i>did2</i>); 2 detectDeletedInternalNodes (<i>did1</i>, <i>did2</i>); 3 detectInsertedLeafNodes (<i>did1</i>, <i>did2</i>); 4 detectDeletedLeafNodes (<i>did1</i>, <i>did2</i>); 5 detectAbsoluteUpdate (<i>did1</i>, <i>did2</i>); 6 detectRelativeUpdate (<i>did1</i>, <i>did2</i>); 7 updateCorrector (<i>did1</i>, <i>did2</i>); 8 detectMoveAmongSibling (<i>did1</i>, <i>did2</i>); 9 detectMovedInternalNode (<i>did1</i>, <i>did2</i>); 10 detectMovedLeafNode (<i>did1</i>, <i>did2</i>); </pre>
--

Figure 4.33: XANDY-O: The *changeDetector* Algorithm.

changes that can be detected by XANDY-O are discussed in Chapter 1. The algorithm for detecting changes in XANDY-O is depicted in Figure 4.33. The algorithm works as follows. First, we detect the inserted internal nodes by issuing an SQL query. The deleted internal nodes are also detected by invoking an SQL query. Next, we find the insertions and deletions of leaf nodes by issuing two SQL queries each. Then, the absolute updated leaf nodes will be determined by using an SQL query followed by detecting the relative updated leaf nodes. Having detected the relative updated leaf nodes, we invoke the *updateCorrector* algorithm. We shall elaborate on the *updateCorrector* algorithm further in the subsequent section. Next, we detect the nodes that are moved among their siblings by invoking the *detectMoveAmongSiblings* algorithm in which several SQL queries are invoked. Finally, we detect the internal and leaf nodes that are moved to different parent nodes. We shall elaborate on each step in turn.

4.4.1 Insertion of Internal Nodes

Recall that the inserted internal nodes are ones that are in the new version (T_2), but not in the old version (T_1). The inserted internal nodes must not be the root nodes of best matching subtrees. Definition 3.8 formally defines the inserted internal nodes.

According to Definition 3.8, we need to use the **AncestorInfo** in which the information of internal nodes are stored and the **MATCHING** table in which the information of best matching internal nodes are stored. Observe that an internal node from an XML document in the **AncestorInfo** is identified by the *Level*, *MinSibOrder*, and *MaxSibOrder* attributes. Therefore, the intuition is to find tuples in the **AncestorInfo** table whose the *Level*, *MinSibOrder*, and *MaxSibOrder* attributes do not exist in the **MATCHING** table. Figure 4.34(a) depicts the SQL query to detect a set of newly inserted internal nodes. The *did1* and *did2* refer to the document id of the old and new versions of an XML document, respectively.

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

```

1 SELECT DISTINCT did1, did2, NODENAME,
2   NODELEVEL, MINSIBORDER, MAXSIBORDER,
3   LOCALORDER
4 FROM ANCESTORINFO AS A
5 WHERE
6   DOC_ID = did2 AND
7   NOT EXISTS
8     (SELECT * FROM MATCHING AS M
9      WHERE M.DID1 = did1 AND
10           M.DID2 = did2 AND
11           M.MINSO2 = A.MINSIBORDER AND
12           M.MAXSO2 = A.MAXSIBORDER AND
13           M.LEVEL = A.NODELEVEL)

```

(a) Detecting Inserted Internal Nodes.

```

1 SELECT DISTINCT did1, did2, L.LEAFORDER,
2   L.SIBLINGORDER, L.PATH_ID, L.LEAFVALUE,
3   L.LOCALORDER, L.LEVEL
4 FROM LEAFVALUE AS L, INS_INT AS I
5 WHERE
6   L.DOC_ID = did2 AND
7   I.DID1 = did1 AND I.DID2 = did2 AND
8   L.SIBLINGORDER BETWEEN I.MINSO AND
9     I.MAXSO AND
10    L.LEVEL = I.LEVEL+1

```

(b) Detecting Inserted Leaf Nodes (1).

```

1 SELECT
2   did1, did2, L.LEAFORDER, L.LEVEL, L.SIBLINGORDER,
3   L.PATH_ID, L.LEAFVALUE, L.LOCALORDER
4 FROM LEAFVALUE AS L,
5   (SELECT DISTINCT M.MINSO1, M.MAXSO1, M.MINSO2,
6    M.MAXSO2, A.PATH_ID, A.LEAFVALUE
7   FROM MATCHING AS M, LEAFVALUE AS A
8   WHERE M.DID1 = did1 AND M.DID2 = did2 AND
9        A.DOC_ID = did2 AND
10       A.LEVEL = M.LEVEL+1 AND
11       A.SIBLINGORDER BETWEEN M.MINSO2 AND M.MAXSO2
12  EXCEPT ALL
13  SELECT DISTINCT M.MINSO1, M.MAXSO1, M.MINSO2,
14   M.MAXSO2, B.PATH_ID, B.LEAFVALUE
15  FROM MATCHING AS M, LEAFVALUE AS B
16  WHERE M.DID1 = did1 AND M.DID2 = did2 AND
17        B.DOC_ID = did1 AND
18        B.LEVEL = M.LEVEL+1 AND
19        B.SIBLINGORDER BETWEEN M.MINSO1 AND M.MAXSO1) AS D
20 WHERE
21   L.DOC_ID = did2 AND
22   L.SIBLINGORDER BETWEEN D.MINSO2 AND D.MAXSO2 AND
23   L.PATH_ID = D.PATH_ID AND
24   L.LEAFVALUE = D.LEAFVALUE

```

(c) Detecting Inserted Leaf Nodes (2).

Figure 4.34: SQL Queries: Insertion of Internal and Leaf Nodes.

The SQL query depicted in Figure 4.34(a) is able to find all inserted internal nodes. Suppose we have an internal node n . If n is in the new version, then the condition (line 6) in the WHERE-clause of the SQL query is true. If n has a corresponding node in the old version, then the condition in lines 7–13 will be false. But if n has no corresponding node in the old version, then the condition in lines 7–13 will be true. Therefore, node n that is in the result of the SQL query is an inserted node. The result of the SQL query depicted in Figure 4.34(a) is stored in the INS_INT table as depicted in Figure 4.36(a). The semantics of attributes of the INS_INT table are depicted in Figures 4.35(a) and 4.35(b).

4.4.2 Deletion of Internal Nodes

The deleted internal nodes can also be found by using the AncestorInfo and MATCHING tables. We use the similar attributes as for finding inserted internal nodes. The *DID1*, *DID2*, *Level*, *MinSO1*, and *MaxSO1* attributes of the MATCHING table are used to find deleted internal nodes. The SQL query depicted in Figure 4.34(a) can be used to detect a set of deleted internal nodes after slight modifications. We replace “MINSO2” (line 11) and “MAXSO2” (line 12) with “MINSO1” and “MAXSO1”, respectively. We also replace the “*did2*” in line 6 with “*did1*”. The modified SQL query is able to detect all deleted internal nodes. The correctness of the modified SQL query can be examined by following the similar intuitions as examining the correctness of the SQL query for detecting inserted internal nodes. The

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

DEL_INT	(DID1, DID2, NAME, LEVEL, MINSO, MAXSO, LO)
INS_INT	(DID1, DID2, NAME, LEVEL, MINSO, MAXSO, LO)
DEL_LEAF	(DID1, DID2, LEAFORDER, LEVEL, LO, SO, PATH_ID, VALUE)
INS_LEAF	(DID1, DID2, LEAFORDER, LEVEL, LO, SO, PATH_ID, VALUE)
UPD_LEAF	(DID1, DID2, LORDER1, LORDER2, SO1, SO2, LEVEL, PATH_ID, VALUE1, VALUE2, LO1, LO2)
MOV_INT	(DID1, DID2, NAME, LEVEL, LO1, LO2, MINSO1, MAXSO1, MINSO2, MAXSO2)
MOV_LEAF	(DID1, DID2, LORDER1, LORDER2, LEVEL, LO1, LO2, PATH_ID, SO1, SO2, VALUE)

(a) Tables and Attributes

Attribute	Description	Attribute	Description
DID1	The document id of the first document	LORDER2	The leaf order of a leaf node in the second version
DID2	The document id of the second document	LO	The local order of a node
NAME	The internal node's name	LO1	The local order of a node in the first version
LEVEL	The node's level	LO2	The local order of a node in the second version
MINSO	The minimum sibling order of an internal node	SO1	The sibling order of a leaf node in the first version
MAXSO	The maximum sibling order of an internal node	SO2	The sibling order of a leaf node in the second version
SO	The sibling order of a leaf node	VALUE1	The old value of an updated node
PATH_ID	The path id of a leaf node	VALUE2	The new value of an updated node
LORDER1	The leaf order of a leaf node in the first version	VALUE	The leaf node's value
		LEAFORDER	The leaf order of a leaf node

(b) Descriptions of Attributes

Figure 4.35: Delta Tables Attributes and Attribute Descriptions.

DID1	DID2	Name	Level	MinSO	MaxSO	LO
1	2	staff	2	5	6	3
1	2	research	3	6	6	3

(a) Inserted Internal Nodes (INS_INT Table)

DID1	DID2	Name	Level	MinSO	MaxSO	LO
1	2	staff	2	1	2	1
1	2	research	3	2	2	3

(b) Deleted Internal Nodes (DEL_INT Table)

DID1	DID2	Leaf Order	Path_ID	SO	LO	Level	Value
1	2	10	4	6	1	4	Semantic Web
1	2	8	2	5	1	3	Steve
1	2	9	3	5	2	3	Asst Prof
1	2	4	4	2	2	4	Information Retrieval
1	2	2	3	1	2	3	Prof

(c) Inserted Leaf Nodes (INS_LEAF Table)

DID1	DID2	Leaf Order	Path_ID	SO	LO	Level	Value
1	2	7	4	4	1	4	Data Mining
1	2	1	2	1	1	3	Smith
1	2	3	4	2	1	4	Web Mining
1	2	2	3	1	2	4	Assoc Prof
1	2	4	4	2	2	4	Multimedia Mining
1	2	10	3	5	2	3	Assoc Prof

(d) Deleted Leaf Nodes (DEL_LEAF Table)

DID1	DID2	LOrder1	LOrder2	Path_ID	LO1	LO2	SO1	SO2	Level	Value1	Value2
1	2	10	2	3	2	2	5	1	3	Assoc Prof	Prof

(e) Updated Leaf Nodes (UPD_LEAF Table)

Figure 4.36: Delta Tables: Insertion, Deletion, and Updates.

result of this SQL query is stored in the DEL_INT table as depicted in Figure 4.36(b). The semantics of attributes of the DEL_INT table are depicted in Figures 4.35(a) and 4.35(b).

4.4.3 Insertion of Leaf Nodes

We use the *Doc_ID*, *Path_ID*, *Level*, and *SiblingOrder* attributes of the *LeafValue* table to detect the inserted leaf nodes in inserted subtrees. The *DID1*, *DID2*, *Level*, *MinSO* and *MaxSO* attributes of the INS_INT table are also used. We also use the *Doc_ID*, *Path_ID*, *Level*, and *SiblingOrder* attributes of the *LeafValue* table to detect the inserted leaf nodes in inserted subtrees. We need to use the *DID1*, *DID2*, *Level*, *MinSO1*, *MaxSO1*, *MinSO2*, and *MaxSO2* attribute of the MATCHING table. Note that by using these properties, we also

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.37: Intermediate Result of SQL query in Figure 4.34(c).

detect the updated leaf nodes as they can be decomposed into a pair of deleted and inserted leaf nodes.

The SQL query depicted in Figure 4.34(b) is able to detect all inserted leaf nodes in newly inserted subtrees. The SQL query basically retrieves the child leaf nodes of inserted internal nodes (lines 8–9). The inserted leaf nodes in best matching subtrees are able to be detected by using the SQL query depicted in Figure 4.34(c). Observe that the SQL query depicted in Figure 4.34(c) has two sub queries (lines 5–11 and lines 13–19). Let r_1 and r_2 be two root nodes of best matching subtrees in first and second versions, respectively. Let $\mathcal{L}_{r_1}$ and $\mathcal{L}_{r_2}$ be two sets of leaf nodes which are the child nodes of r_1 and r_2 , respectively. Let $\mathcal{L}_{(r_2-r_1)}$ be a set of inserted leaf nodes in best matching subtrees rooted at r_1 and r_2 . Then, $\mathcal{L}_{(r_2-r_1)} = \mathcal{L}_{r_2} - \mathcal{L}_{r_1}$. This is based on Definition 3.9. The sub queries in lines 5–11 and lines 13–19 of the SQL query are used to retrieve leaf nodes that are the child nodes of r_2 and r_1 , respectively. The “EXCEPT ALL” statement is used to find $y_i \in \mathcal{L}_{r_2}$ where $\forall a_j \in \mathcal{L}_{r_1}$ such that $(a_j \not\rightarrow y_i)$. In other words, this statement is used to find $\mathcal{L}_{(r_2-r_1)}$. Finally, we need to find other detailed information on inserted leaf nodes, such as the sibling orders and local orders, by joining $\mathcal{L}_{(r_2-r_1)}$ with $\mathcal{L}_{r_2}$ (lines 21–24). Let us now elaborate on this SQL query by using an example. Given the MATCHING table (Figure 4.21) and the LeafValue table (Figure 4.3), the sub query in lines 5–11 returns a set of tuples

R_1 as depicted in Figure 4.37. The sub query in lines 13–19 returns a set of tuples R_2 as depicted in Figure 4.37. Note that the highlighted tuples in R_1 and R_2 denote the tuples that are in both R_1 and R_2 . The “EXCEPT ALL” statement in line 12 is used to find the tuples that are in R_1 but not in R_2 . Hence, the sub query in lines 5–19 returns a set of tuples (D) that are only available in R_1 . Then, D is joined with the `LeafValue` table in order to get other detailed information on inserted leaf nodes.

The results of these SQL queries are stored in the `INS_LEAF` table as depicted in Figure 4.36(c). The semantics of attributes of the `INS_LEAF` table are depicted in Figures 4.35(a) and 4.35(b). Observe that the highlighted tuple in Figure 4.36(c) is actually an updated leaf node detected as an inserted leaf node because it can be decomposed as a deleted leaf nodes and an inserted leaf node.

4.4.4 Deletion of Leaf Nodes

Similar to the inserted leaf nodes, the deleted leaf nodes are also classified into two categories: *deleted leaf nodes in deleted subtrees*, and *deleted leaf nodes in best matching subtrees*. We also use the SQL queries depicted in Figures 4.34(b) and 4.34(c) for detecting the deleted leaf nodes after slight modifications. We replace “INS_INT” in line 4 in Figure 4.34(b) by “DEL_INT”. We also replace the “*did2*” in line 6 in Figure 4.34(b) and in lines 9 and 21 in Figure 4.34(c) with “*did1*”. The “*did1*” in line 17 in Figure 4.34(c) is replaced by “*did2*”. We also replace “MINS02” and “MAXS02” in lines 11 and 22 in Figure 4.34(c) by “MINS01” and “MAXS01”, respectively. The “MINS01” and “MAXS01” in line 19 in Figure 4.34(c) are replaced by “MINS02” and “MAXS02”, respectively.

The results of these SQL queries are stored in the `DEL_LEAF` table as shown in Figure 4.36(d). The semantics of attributes of the `DEL_LEAF` table are depicted in Figures 4.35(a) and 4.35(b). Observe that the highlighted tuple in Figure 4.36(d) is actually an updated leaf node detected as a deleted leaf node because it can be decomposed as a deleted leaf nodes and an inserted leaf node.

4.4.5 Content Updates of Leaf Nodes

Recall that update operation is classified into *absolute update* operation and *relative update* operation. In this section, we shall discuss how each type of update operation is detected by XANDY-O.

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

<pre> 1 SELECT DISTINCT 2 did1, did2, D.LEAFORDER, I.LEAFORDER, D.PATH_ID, 3 D.LEVEL, D.LO AS LO1, I.LO AS LO2, 4 D.SO AS SO1, I.SO AS SO2, 5 D.VALUE AS V1, I.VALUE AS V2 6 FROM DEL_LEAF AS D, INS_LEAF AS I, MATCHING AS C 7 WHERE 8 D.VALUE != I.VALUE AND 9 D.PATH_ID = I.PATH_ID AND D.LO = I.LO AND 10 D.SO BETWEEN C.MINSO1 AND C.MAXSO1 AND 11 D.LEVEL = (C.LEVEL+1) AND 12 I.SO BETWEEN C.MINSO2 AND C.MAXSO2 AND 13 I.LEVEL = (C.LEVEL+1) AND 14 I.DID1 = did1 AND I.DID2 = did2 AND 15 D.DID1 = did1 AND D.DID2 = did2 </pre>	<pre> 1 SELECT DISTINCT 2 did1, did2, D.LEAFORDER, I.LEAFORDER, D.PATH_ID, 3 D.LEVEL, D.LO AS LO1, I.LO AS LO2, 4 D.SO AS SO1, I.SO AS SO2, 5 D.VALUE AS V1, I.VALUE AS V2 6 FROM DEL_LEAF AS D, INS_LEAF AS I, MATCHING AS C 7 WHERE 8 D.VALUE != I.VALUE AND D.LO != I.LO AND 9 D.PATH_ID = I.PATH_ID AND 10 D.SO BETWEEN C.MINSO1 AND C.MAXSO1 AND 11 D.LEVEL = (C.LEVEL+1) AND 12 I.SO BETWEEN C.MINSO2 AND C.MAXSO2 AND 13 I.LEVEL = (C.LEVEL+1) AND 14 I.DID1 = did1 AND I.DID2 = did2 AND 15 D.DID1 = did1 AND D.DID2 = did2 </pre>
---	--

(a) Detecting Absolute Updated Leaf Nodes.

(b) Detecting Relative Updated Leaf Nodes.

```

1 DELETE FROM DEL_LEAF AS A
2 WHERE
3   DID1 = did1 AND DID2 = did2 AND
4   EXISTS
5     (SELECT * FROM UPD_LEAF AS U
6      WHERE U.DID1 = did1 AND U.DID2 = did2 AND
7           D.LEAFORDER = U.LORDER1)

```

(c) Delete Updated Leaf Nodes Detected as Deleted Leaf Nodes

Figure 4.38: SQL Queries: Updated Leaf Nodes.

According to Definition 3.11, updated leaf nodes can be detected from two sets of inserted and deleted leaf nodes. That is, the inserted and deleted leaf nodes are kept in the `INS_LEAF` and `DEL_LEAF` tables. As updated leaf nodes must be available in both versions of XML documents, we also need to make sure that their parent nodes are best matching subtrees. Hence, we also use the `MATCHING` table.

Absolute Update Operation

The SQL query for detecting the absolute updated leaf nodes is depicted in Figure 4.38(a). The SQL query joins the `INS_LEAF`, `DEL_LEAF`, and `MATCHING` tables (line 4). The conditions in `WHERE`-clause of this SQL query is based on Definition 3.11. Suppose we have an absolute updated leaf node u decomposed as a deleted leaf node e and an inserted leaf node y . Line 8 is used to check that the values of nodes e and y are different. Line 9 is used to make sure that the path of node e is the same as one of node y and the local order of node e is also the same as one of node y . Recall that the parent nodes of nodes e and y must be best matching internal nodes. Lines 10–13 are used to guarantee that the parent nodes of nodes e and y are best matching internal nodes.

After we detect an absolute updated leaf node u , we need to delete the corresponding tuple of a deleted leaf nodes e in the `DEL_LEAF` table and one of an inserted leaf nodes y in the `INS_LEAF` table. Figure 4.38(c) depicts the SQL query for deleting the corresponding tuples of updated leaf nodes detected as deleted leaf nodes in the `DEL_LEAF` table. For deleting the corresponding tuples of updated leaf nodes detected as inserted leaf nodes in the

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.39: Intermediate Result of SQL query in Figure 4.38(a).

INS_LEAF table, we use the SQL query depicted in Figure 4.38(c) after slight modifications. The modifications are as follows. We replace “DEL_LEAF” in line 1 with “INS_LEAF”. The “LORDER1” (line 7) is replaced by “LORDER2”.

For example, consider the INS_LEAF and DEL_LEAF tables depicted in Figures 4.36(c) and 4.36(d), respectively, and MATCHING tables depicted in Figure 4.21. Figure 4.39 depicts the intermediate results of SQL query depicted Figure 4.38(a). *R1* is retrieved by joining the INS_LEAF and DEL_LEAF tables (*Join1*). Then, *R1* and the MATCHING table are joined in order to find the tuples that satisfy the conditions in lines 8–11 (Figure 4.38(a)). Finally, the SQL query returns the attributes that are specified in lines 2–3.

Relative Update Operation

The SQL query for detecting the relative updated leaf nodes is depicted in Figure 4.38(b). The SQL query joins the INS_LEAF, DEL_LEAF, and MATCHING tables (line 6). The conditions in WHERE-clause of this SQL query is also based on Definition 3.11. Suppose we have a relative updated leaf node *u* decomposed as a deleted leaf node *e* and an inserted leaf node *y*. Line 8 is used to check that the values and local orders of nodes *e* and *y* are different.

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.41: The *updateCorrector* Algorithm and SQL Queries.

We use the *updateCorrector* algorithm to correct the result. Figure 4.41(a) depicts the *updateCorrector* algorithm. The algorithm works as follows. First, the algorithm determines the updated leaf nodes in the first version that are detected as updated leaf nodes more than once (line 2, Figure 4.41(a)) by using the SQL query *Q1* as depicted in Figure 4.41(b). Subquery in lines 5–12 is used to retrieve tuples of corresponding updated leaf nodes that are detected more than once. Lines 20–21 in Figure 4.41(b) are used to retrieve only one row. The SQL query *Q1* returns *R*. For example, consider the UPD_LEAF as depicted in Figure 4.40(c). The intermediate results the SQL query in Figure 4.41(c) using the UPD_LEAF in Figure 4.40(c) is depicted in Figure 4.42.

Next, the algorithm deletes the *incorrect tuples* (line 3, Figure 4.41(b)) by using the SQL query in Figure 4.41(c). A tuple *t* is an *incorrect tuple* if one and only one of the following conditions is satisfied: 1) the *LORDER1* of tuple *t* is equal to *LORDER1* of *R*, 2) the *LORDER2* of tuple *t* is equal to *LORDER2* of *R*. We also do the same process for the updated leaf nodes in the second version that are detected as updated leaf nodes more than once. Note that the SQL query *Q2* is generated by slightly modifying the query in Figure 4.41(b). We replace the “LORDER1” in lines 6, 11, 12, and 19 in Figure 4.41(b) with “LORDER2”.

After we detect a relative updated leaf node *u*, we also need to delete the corresponding tuple of a deleted leaf nodes *e* in the DEL_LEAF table and one of an inserted leaf nodes *y* in the INS_LEAF table.

Figure 4.42: Intermediate Results of SQL Query in Figure 4.41(c).

4.4.6 Move Operation

In this section, we shall discuss how the move operations are detected. Recall that move operations are classified into *move among siblings* and *move to different parent nodes*. Let us elaborate further how to detect each type of move operations.

Move Among Siblings

The naive approach of detecting the movement of nodes among the siblings is to check whether or not the local order of a node has changed. However, this approach may lead to the detection of *non-optimal* deltas in certain situations. We illustrate this with a simple example. Suppose we have two versions of XML trees as depicted in Figure 4.43(a). Node *e2* with value “New” is a newly inserted node. If we do not consider this newly inserted node during the move detection process, then we may detect that nodes *e2* with values “C” and “D” are moved among their siblings since they have different local order values in the old and new versions. Hence, the detected delta consists of two move operations and an insert operation. However, the optimal delta should consist of only an insert operation. To

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.43: Move Among Siblings: Example, Algorithm, and SQL Queries.

overcome this problem, we need to simulate the insertions and deletions occurring under the same parent before detecting moved nodes.

Figure 4.43(b) depicts the algorithm for detecting the movement of nodes among their siblings. The *detectMoveAmongSiblings* algorithm works as follows. First, the algorithm generates the `moveList` table that initially consists of nodes that are in the matching subtrees (both internal and leaf nodes). The SQL queries shown in Figures 4.43(c) and 4.43(d) are used to generate the `moveList` table. The intuition behind the SQL query depicted in Figure 4.43(c) is to find the matching leaf nodes in the best matching subtrees. The intermediate result of this SQL query is depicted in Figure 4.44. The intuition behind the SQL query depicted in Figure 4.43(d) is to retrieve the best matching internal nodes from the `MATCHING` table. For example, Figure 4.45(a) depicts the `moveList` table after the SQL queries depicted in Figures 4.43(c) and 4.43(d) are issued (partial view only).

The second step of the algorithm is to simulate the insertions and deletions by adjusting the local orders of the nodes affected by these operations. This is based on our observation that the local order of a node may be changed because there are insertions/deletions of its sibling nodes. Our observations are as follows.

- A deletion of node n , that has local order equal to k , will decrease the local orders of its siblings, that have local order greater than k , by one. For example, consider two trees as depicted in Figures 4.46(a) and 4.46(b). We notice that node “C” is deleted and the local orders of node “D” and node “E” are decreased by one. This is because their local orders are greater than local order of node “C”.

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.44: Intermediate Result of SQL Query in Figure 4.43(c).

- An insertion of node n to be the k -th child of a parent node p will increase the local orders of the child nodes of node p , that have local order greater than or equal to k , by one. For example, consider two trees as depicted in Figures 4.46(a) and 4.46(c). We notice that node “X” is inserted as 3rd child of node “P”. The local orders of nodes “C”, “D”, and “E” are increased by one as their local orders are greater or equal to three.
- For instance, Figure 4.45(b) depicts the `moveList` table after the insertions and deletions are simulated (partial view only). Note that the highlighted tuples are ones that are affected by the adjustments. Finally, we delete the tuples in the `moveList` table that have same values of the `LO1` and `LO2` attributes. The same values of the `LO1` and `LO2` attributes indicate that the nodes are *not* moved among their siblings. The `moveList` table after deleting the tuples that have same values of the `LO1` and `LO2` attributes is shown in Figure 4.45(c) (partial view only).

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

DID1	DID2	Min SO1	Max SO1	Min SO2	Max SO2	Value	LO1	LO2	Level
1	2	3	0	3	0	Chen	1	1	3
1	2	3	0	3	0	Assoc Prof	2	2	3
1	2	4	0	4	0	Web Data Management	2	1	4
1	2	5	0	1	0	Mark	1	1	3
1	2	6	0	2	0	Digital Libraries	1	1	4
1	2	7	0	7	0	Don	1	1	3
1	2	7	0	7	0	Asst Prof	2	2	3
1	2	8	0	8	0	Indexing	1	1	4
1	2	8	0	8	0	Security	2	2	4
1	2	1	8	1	8	-	1	1	1
1	2	3	4	3	4	-	2	2	2
1	2	5	6	1	2	-	3	1	2
1	2	7	8	7	8	-	4	4	2
1	2	4	4	4	4	-	3	3	3
1	2	6	6	2	2	-	3	3	3
1	2	8	8	8	8	-	3	3	3

(a) moveList Table: Intial state

DID1	DID2	Min SO1	Max SO1	Min SO2	Max SO2	Value	LO1	LO2	Level
1	2	3	0	3	0	Chen	1	1	3
1	2	3	0	3	0	Assoc Prof	2	2	3
1	2	4	0	4	0	Web Data Management	1	1	4
1	2	5	0	1	0	Mark	1	1	3
1	2	6	0	2	0	Digital Libraries	1	1	4
1	2	7	0	7	0	Don	1	1	3
1	2	7	0	7	0	Asst Prof	2	2	3
1	2	8	0	8	0	Indexing	1	1	4
1	2	8	0	8	0	Security	2	2	4
1	2	1	8	1	8	-	1	1	1
1	2	3	4	3	4	-	1	2	2
1	2	5	6	1	2	-	2	1	2
1	2	7	8	7	8	-	4	4	2
1	2	4	4	4	4	-	3	3	3
1	2	6	6	2	2	-	3	3	3
1	2	8	8	8	8	-	3	3	3

(b) moveList Table: After applying insertions and deletions

DID1	DID2	MinSO1	MaxSO1	MinSO2	MaxSO2	Value	LO1	LO2	Level
1	2	3	4	3	4	-	1	2	2
1	2	5	6	1	2	-	2	1	2

(c) moveList Table: Final state

Figure 4.45: The moveList Table.

The moved internal nodes among their siblings are stored in the MOV_INT table. The moved leaf nodes among their siblings are stored in the MOV_LEAF table. The attributes of the MOV_INT and MOV_LEAF tables are depicted in Figures 4.35(a) and 4.35(b).

Move To Different Parent Nodes

A particular node that is moved to different parent node is detected as a pair of deletion and insertion. Hence, we are able to determine the nodes that are moved to different parent nodes by querying the DEL_INT and INS_INT tables (for moved internal nodes), and the DEL_LEAF and INS_LEAF tables (for moved leaf nodes).

Figure 4.46: Effects of Insertion/Deletion to Local Order.

Figure 4.47: Example: Move To Different Parent Nodes.

According to Definition 3.12, the moved internal nodes is best matching internal nodes whose parent nodes are not best matching internal nodes. Let us elaborate on the significance of this. Consider two versions of an XML document as depicted in Figure 4.47. We notice that t_{58} can be matched to t_3 and t_{11} , but we can observe that t_{11} is more similar to t_{58} compared to t_3 . Hence, $t_{11} \approx t_{58}$. Since the parent nodes of nodes 11 and 58 are not best matching internal nodes, we conclude that t_{11} is move to be t_{58} . Hence, the XDelta of these XML documents are subtree rooted at node 3 (t_3) is deleted, t_{11} is moved to different parent node, and the value of node 13 is updated. Note that the XDelta consists of three edit operations. If we match t_{58} to t_3 instead of t_{11} , then the XDelta of these XML documents will be a deletion of subtree 11, subtree t_3 is moved to different parent nodes, the values of nodes 4 and 6 are updated to “Jane” and “jane@univ.edu”, respectively. That is, the XDelta consists of four edit operations. From this example, we notice that we are able to get better quality XDeltas by using Definition 3.12.

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

<pre>1 SELECT 2 D.DID1, D.DID2, D.LEAFORDER, 3 I.LEAFORDER, T.LEVEL, I.NAME, 4 D.LOCALORDER, I.LOCALORDER, 5 D.MINSO, D.MAXSO, I.MINSO, I.MAXSO 6 FROM TEMPMATCHING AS T, INS_INT AS I, 7 DEL_INT AS D 8 WHERE T.FLAG = 1 AND 9 I.NAME = D.NAME AND 10 I.DID1 = did1 AND I.DID2 = did2 AND 11 D.DID1 = did1 AND D.DID2 = did2 AND 12 T.MINSO1 = D.MINSO AND 13 T.MAXSO1 = D.MAXSO AND 14 T.LEVEL = D.LEVEL AND 15 T.MINSO2 = I.MINSO AND 16 T.MAXSO2 = I.MAXSO AND 17 T.LEVEL = I.LEVEL</pre>	<pre>1 SELECT 2 D.DID1, I.DID2, D.LEAFORDER, I.LEAFORDER, 3 D.LEVEL, D.PATH_ID, D.LOCALORDER, 4 I.LOCALORDER, D.SIBLINGORDER, I.SIBLINGORDER, 5 I.VALUE 6 FROM DEL_LEAF AS D, INS_LEAF AS I, 7 MATCHING AS C1, MATCHING AS C2 8 WHERE 9 D.DID1 = did1 AND D.DID2 = did2 AND 10 I.DID1 = did1 AND I.DID2 = did2 AND 11 C1.DID1 = did1 AND C1.DID2 = did2 AND 12 C2.DID1 = did1 AND C2.DID2 = did2 AND 13 D.PATH_ID = I.PATH_ID AND D.VALUE = I.VALUE AND 14 D.LEVEL = C1.LEVEL+1 AND I.LEVEL = C2.LEVEL+1 AND 15 C1.LEVEL = C2.LEVEL AND 16 D.MINSO >= C1.MINSO1 AND D.MAXSO <= C1.MAXSO1 AND 17 I.MINSO >= C2.MINSO2 AND I.MAXSO <= C2.MAXSO2</pre>
(a) Detecting Moved Internal Nodes	(b) Detecting Moved Leaf Nodes

Figure 4.48: SQL Queries: Move To Different Parent Nodes.

In the bottom-up approach, these moved internal nodes can be found by using the `DEL_INT`, `INS_INT`, and `TempMatching` tables. Recall that the `TempMatching` table has *Flag* attribute that is used to annotate the root nodes that are candidates to be the root nodes of moved subtrees. The possible moved subtrees have the *Flag* attribute equal to “1”. This indicates that subtrees *P* (in the old version) and *Q* (in the new version) rooted at nodes *p* and *q*, respectively are matching subtrees, but $parent(p) \neq parent(q)$. Then, $parent(p)$ and $parent(q)$ are detected as deleted and inserted internal nodes, respectively. Hence, nodes *p* and *q* are also determined as deleted and inserted internal nodes. The SQL query in Figure 4.48(a) is used to detect the internal nodes that are moved to different parent nodes. Line 8 is used to retrieve the best matching internal nodes whose parent nodes are not best matching internal nodes. Lines 12–14 are used to find the deleted internal nodes that are best matching internal nodes but their parent nodes are not best matching internal nodes. Similarly, lines 15–17 are used to find the inserted internal nodes that are best matching internal nodes but their parent nodes are not best matching internal nodes. The result of the SQL query is stored in the `MOV_INT` table. The next step is to find all the leaf nodes that are in moved subtrees. These leaf nodes can be found in the `DEL_LEAF` and `INS_LEAF` tables. The information on the leaf nodes that are in moved subtrees is stored in the `MOV_LEAF` table. The attributes of the `MOV_INT` and `MOV_LEAF` tables are depicted in Figures 4.35(a) and 4.35(b).

In the top-down approach, finding the internal nodes that are moved to different parent nodes is a time-consuming process. Suppose we have two subtrees *P* (in the old version) and *Q* (in the new version) rooted at nodes *p* and *q*, respectively. When $parent(p)$ has no matching subtree in the new version, all subtrees in the subtree rooted at $parent(p)$

(including subtree P) will not be compared to the subtrees in the new version. Similarly, when $\text{parent}(q)$ has no matching subtree in the old version, all subtrees in the subtree rooted at $\text{parent}(q)$ (including subtree Q) will not be compared to the subtrees in the old version. That is, we do not have information on the matching subtrees that are in the subtrees rooted at $\text{parent}(p)$ and $\text{parent}(q)$. This leads us to find the information on the matching subtrees that are in the subtrees rooted at $\text{parent}(p)$ and $\text{parent}(q)$. Hence, in the top-down approach, the moved internal nodes are detected as pairs of deleted and inserted internal nodes in order not to sacrifice the performance of the top-down approach.

The leaf nodes are also able to be moved to different parent nodes. Both approaches in XANDY-O are able to detect these moved leaf nodes. The SQL query depicted in Figure 4.48(b) is used to find the leaf nodes that are moved to different parent nodes. We also use the **Matching** table in order to make sure that the parent nodes of these moved leaf nodes are in both versions. Lines 14–15 are used to guarantee that the parent nodes of these moved leaf nodes are in both versions. Line 13 is used to ensure that these leaf nodes are matching leaf nodes. The result of the query is stored in the **MOV_LEAF** table.

4.5 Detecting the Changes Phase in HELIOS-O

In section, we discuss how the changes are detected by HELIOS-O after the best matching subtrees are determined. The *changeDetector* algorithm for detecting these types of changes is depicted in Figure 4.49.

The *changeDetector* algorithm works as follows. First, we detect the inserted and deleted non-inlined internal nodes (lines 5–9). They are detected by issuing two SQL queries for each non-inlined internal node in DTD U . Next, the inserted and deleted inlined internal nodes are detected by invoking four SQL queries for each inlined internal node in DTD U (lines 10–16). After that, we determine inserted, deleted, and updated inlined leaf nodes (lines 17–24) followed by detecting inserted, deleted, and updated non-inlined leaf nodes (lines 25–33). Next, the moved non-inlined and inlined internal nodes are detected (lines 34–37 and 38–42, respectively). Recall that the moved leaf nodes can be in the moved subtrees. The next step is to determine the moved non-inlined and inlined leaf nodes that are in the moved subtrees (lines 43–47 and 48–52, respectively). Finally, we detect the moved leaf nodes that are not in the moved subtrees and the nodes that are moved among their siblings. Observe that we shall invoke SQL queries (whenever appropriate) to detect the changes. The “[**param**]” in

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

```

Input
  U : DTD of the XML documents
  the Matching table
  Two versions of an XML document stored in RDBMS

Output
  the delta tables

1   $\mathbb{I}_\ell(U) \leftarrow$  all the inlined leaf nodes in U
2   $\mathbb{I}_i(U) \leftarrow$  all the inlined internal nodes in U
3   $\mathbb{N}_\ell(U) \leftarrow$  all the non-inlined leaf nodes in U
4   $\mathbb{N}_i(U) \leftarrow$  all the non-inlined internal nodes in U
5  for all  $i_x \in \mathbb{N}_i(U)$  do
6    nodeName  $\leftarrow$  name( $i_x$ ); tbName  $\leftarrow$   $r_a$ ;
7    detectInsertedInternalNodes(nodeName, tbName);
8    detectDeletedInternalNodes(nodeName, tbName);
9  end for
10 for all  $i_y \in \mathbb{I}_i(U)$  do
11   parentNode  $\leftarrow$  parent( $i_y$ ); attrName  $\leftarrow$  attribute( $i_y$ );
12   nodeName  $\leftarrow$  name( $i_y$ ); parentTbName  $\leftarrow$   $r_{\text{parentNode}}$ ;
13   parentNodeName  $\leftarrow$  name(parentNode);
14   detectInsertedInternalNodes(nodeName, attrName, parentTbName, parentNodeName);
15   detectDeletedInternalNodes(nodeName, attrName, parentTbName, parentNodeName);
16 end for
17 for all  $\ell_y \in \mathbb{I}_\ell(U)$  do
18   parentNode  $\leftarrow$  parent( $\ell_y$ ); nodeName  $\leftarrow$  name( $\ell_y$ );
19   attrName  $\leftarrow$  attribute( $\ell_y$ ); parentTbName  $\leftarrow$   $r_{\text{parentNode}}$ ;
20   parentNodeName  $\leftarrow$  name(parentNode);
21   detectInsertedLeafNodes(nodeName, attrName, parentTbName, parentNodeName);
22   detectDeletedLeafNodes(nodeName, attrName, parentTbName, parentNodeName);
23   detectUpdatedLeafNodes(nodeName, attrName, parentTbName, parentNodeName);
24 end for
25 for all  $\ell_x \in \mathbb{N}_\ell(U)$  do
26   nodeName  $\leftarrow$  name( $\ell_x$ ); parentNode  $\leftarrow$  parent( $\ell_x$ );
27   parentNodeName  $\leftarrow$  name(parentNode); tbName  $\leftarrow$   $r_a$ ;
28   detectInsertedLeafNodes(nodeName, tbName, parentNodeName);
29   detectDeletedLeafNodes(nodeName, tbName, parentNodeName);
30   detectAbsoluteUpdatedLeafNodes(nodeName, parentNodeName);
31   detectRelativeUpdatedLeafNodes(nodeName, parentNodeName);
32   refineUpdatedLeafNodes( $\ell_x$ );
33 end for
34 for all  $i_x \in \mathbb{N}_i(U)$  do
35   nodeName  $\leftarrow$  name( $i_x$ ); tbName  $\leftarrow$   $r_a$ ;
36   detectMovedInternalNodes(nodeName, tbName);
37 end for
38 for all  $i_y \in \mathbb{I}_i(U)$  do
39   parentNode  $\leftarrow$  parent( $i_y$ ); parentTbName  $\leftarrow$   $r_{\text{parentNode}}$ ;
40   nodeName  $\leftarrow$  name( $i_y$ );
41   detectMovedInternalNodes(nodeName, parentTbName);
42 end for
43 for all  $\ell_x \in \mathbb{N}_\ell(U)$  do
44   parentNode  $\leftarrow$  parent( $\ell_x$ ); nodeName  $\leftarrow$  name( $\ell_x$ );
45   parentNodeName  $\leftarrow$  name(parentNode);
46   detectMovedLeafNodesInMovedSubtrees(nodeName, parentNodeName);
47 end for
48 for all  $\ell_y \in \mathbb{I}_\ell(U)$  do
49   parentNode  $\leftarrow$  parent( $\ell_y$ ); nodeName  $\leftarrow$  name( $\ell_y$ );
50   parentNodeName  $\leftarrow$  name(parentNode);
51   detectMovedLeafNodesInMovedSubtrees(nodeName, parentNodeName);
52 end for
53 detectMovedLeafNodes();
54 detectMoveAmongSiblings();

```

Figure 4.49: HELIOS-O: The *changeDetector* Algorithm.

the SQL queries used in the later discussion will be replaced the parameter *param* defined in the algorithm.

4.5.1 Insertion of Internal Nodes

The intuition behind detecting inserted internal nodes is to find internal nodes that are only available in the *new* version. As the best matching internal nodes are available in the old and new versions, hence, the inserted internal nodes must not be best matching internal nodes. Recall that there are two types of internal nodes, namely, non-inlined and inlined internal nodes. In this section, we shall present how we detect the inserted non-inlined and inlined internal nodes from two versions of an XML document by using the best matching subtrees information. Note that the detected inserted internal nodes are stored in the **INS_INT** table. The moved internal nodes that are moved to other parts of the trees are also detected in this phase as they can be decomposed into pairs of deletions and insertions of internal nodes.

Non-inlined Internal Nodes

For each non-inlined internal node $i_x \in \mathbb{N}_i(U)$, the *changeDetector* algorithm defines two parameters (line 6, Figure 4.49), and invokes the *detectInsertedInternalNodes* function (line 7, Figure 4.49) for detecting inserted non-inlined internal nodes. The *detectInsertedInternalNodes* function executes SQL query as depicted in Figure 4.50(a). The SQL query is quite straightforward and based on Definition 3.8. Line 8 is to retrieve the tuples of corresponding internal nodes in the second versions that are stored in the **AncestorInfo**. Observe that “*doc_id2*” is replaced by the document id of the second version of an XML document. Lines 9–15 are used to ensure that the inserted internal nodes are not best matching internal nodes. Note that the best matching internal nodes are kept in the **MATCHING** table. The *detectInsertedInternalNodes* function returns inserted internal nodes stored in the **INS_INT** table.

Inlined Internal Nodes

For each inlined internal node $i_y \in \mathbb{I}_i(U)$, the *changeDetector* algorithm defines four parameters (lines 11–13, Figure 4.49). Next, the algorithm invokes the *detectInsertedInternalNodes* function (line 14, Figure 4.49) for detecting inserted inlined internal nodes. We observe that the parent nodes of inserted inlined internal nodes can be either best matching internal nodes or inserted non-inlined internal nodes. Hence, the *detectInsertedInternalNodes* function shall execute two SQL queries. The *detectInsertedInternalNodes* function executes the SQL queries as depicted in Figures 4.50(b) and 4.50(c).

Figure 4.50: HELIOS-O: SQL Queries for Detecting Inserted Internal Nodes.

The SQL query in Figure 4.50(b) is used to detect inserted inlined internal nodes whose parent nodes are best matching internal nodes. Line 8 is used to ensure that the inserted inlined internal nodes are only available in the new version. Observe that the information of an inlined internal node is stored as an attribute in the relation storing its parent nodes' information. If the value of the attribute is "FALSE", then it means the inlined internal node does not exist. Otherwise, the inlined internal node is available. Lines 9–14 are used to ensure that the parent nodes of the inserted inlined internal nodes are best matching internal nodes.

The SQL query depicted in Figure 4.50(c) is used to detect inserted inlined internal nodes whose parent nodes are inserted internal nodes. Lines 7–8 are used to get the inserted internal nodes whose name is the same as the name of parent node of node i_y . Line 10 is used to ensure that node i_y is in the second version. Finally, line 9 is used to capture the parent-child relationship between node i_y that is only available in second version and its parent node that is an inserted internal node. The *detectInsertedInternalNodes* function returns inserted internal nodes stored in the INS_INT table. Figure 4.51(a) presents the INS_INT table. The semantics of attributes of the INS_INT table are depicted in Figures 4.52(a) and 4.52(b)

Figure 4.51: HELIOS-O: Delta Tables.

4.5.2 Deletion of Internal Nodes

Similarly, the deleted internal nodes are ones that are in the first version, but not available in the second version. The deleted internal nodes must not be best matching internal nodes as they are available in the old and new versions. In this section, we shall present how we detect the inserted non-inlined and inlined internal nodes from two versions of an XML document by using the best matching subtrees information. Note that the detected deleted internal nodes are stored in the DEL_INT table. The moved internal nodes that are moved to other parts of the trees are also detected in this phase as they can be decomposed into pairs of deletions and insertions of internal nodes.

Non-Inlined Internal Nodes

Having defined two parameters (line 6, Figure 4.49), the *changeDetector* algorithm invokes the *detectDeletedInternalNodes* function (line 8, Figure 4.49) for each non-inlined internal node $i_x \in \mathbb{N}_i(U)$. Note that the *detectDeletedInternalNodes* function is invoked after the *changeDetector* algorithm invokes the *detectInsertedInternalNodes* function discussed in the former section.

The *detectDeletedInternalNodes* function executes SQL query as depicted in Figure 4.50(a) after slight modifications. The “INS_INT” in line 1 is replaced by “DEL_INT”. The “*doc.id2*” (line 8) and “ID2” (line 15) are replaced by “*doc.id1*” and “ID1”, respectively. The *detectDeletedInternalNodes* function returns deleted internal nodes stored in the DEL_INT table.

Inlined Internal Nodes

To detect the deleted inlined internal nodes the *detectDeletedInternalNodes* function also

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.52: HELIOS–O: Table Descriptions.

executes two SQL queries derived from the SQL queries depicted in Figures 4.50(b) and 4.50(c).

The “INS_INT” in line 1 (Figures 4.50(b) and 4.50(c)) is replaced by “DEL_INT”. The SQL query depicted in Figure 4.50(b) is modified as follows. The “A2” (line 4), “FALSE” (line 8), and “TRUE” (line 8) are replaced by “A1”, “TRUE”, and “FALSE”, respectively. We modify the SQL query depicted in Figure 4.50(c) as follows. The “INS_INT” (line 5) and “doc_id2” (line 10) are replaced by “DEL_INT” and “doc_id1”, respectively. The *detectDeletedInternalNodes* function returns deleted internal nodes stored in the DEL_INT table. Figure 4.51(b) presents the DEL_INT table. The semantics of attributes of the DEL_INT table are depicted in Figures 4.52(a) and 4.52(b).

4.5.3 Insertion of Leaf Nodes

Recall that there are two types of inserted leaf nodes: *inserted leaf nodes in newly inserted subtrees*, and *inserted leaf nodes in best matching subtrees*. Intuitively, if an inserted node is in the *newly inserted subtrees*, then its parent node must be an inserted internal node. That is, we can find the information of its parent node in the INS_INT table. On the other hand, if an inserted node is in the *best matching subtrees*, then its parent node must be a best matching internal node. Hence, the information of its parent node must be available in the MATCHING table. In this section, we shall elaborate on how we determine inserted leaf nodes, for both non-inlined and inlined leaf nodes.

Inlined Leaf Nodes

For each inlined leaf node $\ell_y \in \mathbb{I}_\ell(U)$, the *changeDetector* algorithm defines four parameters

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.53: HELIOS-O: SQL Queries for Detecting Inserted Leaf Nodes.

(lines 18–20, Figure 4.49). Next, the algorithm invokes the *detectInsertedLeafNodes* function (line 21, Figure 4.49) for detecting inserted inlined leaf nodes.

The *detectInsertedLeafNodes* function executes two SQL queries as depicted in Figures 4.53(a) and 4.53(b). Figure 4.53(a) is used to detect inserted inlined leaf nodes that are in the *best matching subtrees*. As inlined leaf nodes are stored as attributes in the parent nodes' relations, hence, we join the relations in which their parent nodes are stored. We also need to guarantee that the parent nodes are best matching internal nodes (lines 10–14, Figure 4.53(a)). Lines 8–9 are used to ensure that inserted inlined leaf nodes do not exist in the old version (line 8), but are available in the new version (line 9).

To detect inserted inlined leaf nodes that are in the newly inserted subtree we use SQL query depicted in Figure 4.53(b). The idea behind the SQL query in Figure 4.53(b) is to find the inlined leaf nodes that are the child nodes of inserted internal nodes. Intuitively, if an inlined leaf node is a child node of an inserted internal node, then it is also only available in the second version. Line 10 is used to indicate that inserted inlined leaf nodes must be only available in the new version. Lines 7–8 are used to retrieve inserted internal nodes from the *INS_INT* table whose node name is the same as the name of the parent node of node ℓ_y . Finally, line 9 is used to capture the parent-child relationship between inserted internal nodes and inserted inlined leaf nodes. The *detectInsertedLeafNodes* function returns inserted inlined leaf nodes stored in the *INS_LEAF* table.

Non-Inlined Leaf Nodes

For each non-inlined leaf node $\ell_x \in N_\ell(U)$, the *changeDetector* algorithm defines three parameters (lines 26–27, Figure 4.49). Next, the *changeDetector* algorithm invokes the *detectInsertedLeafNodes* (line 28, Figure 4.49) for detecting the inserted non-inlined leaf nodes. Note that lines 30–32 are used to detect the updated leaf nodes that will be discussed in the later section.

The *detectInsertedLeafNodes* function shall execute the SQL queries as depicted in Figures 4.53(c) and 4.53(d). Figure 4.53(c) is used to detect inserted non-inlined leaf nodes that are in the best matching subtrees. Lines 5–9 and lines 11–15 are used to find the non-inlined leaf nodes that are in the new and old versions, respectively. Operator “EXCEPT ALL” in line 10 is used to find non-inlined leaf nodes that are only available in the new version. Figure 4.53(d) is used to detect inserted non-inlined leaf nodes that are in the inserted subtrees. Line 7 is used to guarantee that the parent nodes of the leaf nodes are inserted internal nodes. The *detectInsertedLeafNodes* function returns inserted non-inlined leaf nodes stored in the `INS_LEAF` table.

The `INS_LEAF` table is depicted in Figure 4.51(c). The tuples with “#” are the tuples of inserted inlined leaf nodes, while the ones without “#” are of inserted non-inlined leaf nodes. The highlighted tuple is the tuple of an updated leaf node. Recall that the updated leaf nodes can be decomposed as pairs of deleted and inserted leaf nodes. The semantics of attributes of the `INS_LEAF` table are depicted in Figures 4.52(a) and 4.52(b).

4.5.4 Deletion of Leaf Nodes

Similarly, there also are two types of deleted leaf nodes: *deleted leaf nodes in deleted subtrees*, and *deleted leaf nodes in best matching subtrees*. Intuitively, if a deleted node is in the deleted subtrees, then its parent node must be a deleted internal node. That is, we can find the information of its parent node in the `INS_INT` table. On the other hand, if a deleted node is in the best matching subtrees, then its parent node must be a best matching internal node. Hence, the information of its parent node must be available in the `MATCHING` table. In this section, we shall elaborate on how we determine deleted leaf nodes, for both non-inlined and inlined leaf nodes.

Inlined Leaf Nodes

Having defined four parameters (lines 18–20, Figure 4.49), the *changeDetector* algorithm invokes the *detectDeletedLeafNodes* function (line 22, Figure 4.49) for detecting deleted inlined

leaf nodes. Note that the *detectDeletedLeafNodes* function is invoked after the *changeDetector* algorithm invokes the *detectInsertedLeafNodes* function discussed in the former section.

The deleted inlined leaf nodes can be determined by using the SQL queries depicted in Figures 4.53(a) and 4.53(b) after slight modifications. The “INS_LEAF” in line 1 (Figures 4.53(a) and 4.53(b)) is replaced by the “DEL_LEAF”. The “I2” in lines 3 and 4 is replaced by “I1”. The “IS NULL” (line 8) and “IS NOT NULL” (line 9) are replaced by “IS NOT NULL” and “IS NULL”, respectively. The SQL query in Figure 4.53(b) is modified as follows. The “INS_INT” (line 5) and “doc_id2” (line 10) are replaced by “DEL_INT” and “doc_id1” respectively. The deleted inlined leaf nodes are stored in the DEL_LEAF table.

Non-Inlined Leaf Nodes

To detect the deleted non-inlined leaf nodes the *detectDeletedLeafNodes* function also executes two SQL queries derived from the SQL queries depicted in Figures 4.53(c) and 4.53(d). The “INS_LEAF” in line 1 (Figures 4.53(c) and (d)) is replaced by the “DEL_LEAF”. The “doc_id2” in lines 7 and 16 (Figure 4.53(c)) and line 5 (Figure 4.53(d)) is replaced by “doc_id1”. We replace the “doc_id1” in line 13 (Figure 4.53(c)) with “doc_id2”. The “PID2” in lines 7 and 16 (Figure 4.53(c)) is replaced by “PID1”. We replace the “PID1” in line 13 (Figure 4.53(c)) with “PID2”. The deleted non-inlined leaf nodes are also kept in the DEL_LEAF table.

The DEL_LEAF table is depicted in Figure 4.51(d). The tuples with “#” are the tuples of deleted inlined leaf nodes, while the ones without “#” are of deleted non-inlined leaf nodes. The highlighted tuple is the tuple of an updated leaf node. The semantics of attributes of the DEL_LEAF table are depicted in Figures 4.52(a) and 4.52(b).

4.5.5 Content Updates of Leaf Nodes

Intuitively, the updated leaf nodes are the leaf nodes that are available in both versions and have the same node names, but have different values. In addition to this, the parent nodes of the updated leaf nodes must be the best matching internal nodes. In the *changeDetector* algorithm, the updated leaf nodes are detected after the inserted and deleted leaf nodes are detected.

We notice that the positions among siblings of the updated non-inlined leaf nodes can be different before and after the update operation. Hence, we classify the update operations of the non-inlined leaf nodes into the *absolute update operations* and the *relative update*

Figure 4.54: HELIOS-O: SQL Queries for Detecting Updated Inlined Leaf Nodes.

operations. In the absolute update operation, only the content value of an updated leaf node is changed, while its position among siblings remains the same. In relative update operation, the content value and position among siblings of an updated leaf node are changed. For the inlined leaf nodes, we only have absolute update operations as they occur once under the same parent nodes.

Inlined Leaf Nodes

The *detectUpdatedLeafNodes* function executes the SQL query as depicted in Figure 4.54(a) to determine the updated inlined leaf nodes. Lines 15–17 are used to ensure that the updated inlined leaf nodes are available in both versions (lines 15–16) and they have different values (line 17). Lines 18–29 are used to guarantee that the parent nodes of the updated inlined leaf nodes are best matching internal nodes. The result of the SQL query depicted in Figure 4.54(a) is stored in the `UPD_LEAF` table. The semantics of attributes of the `UPD_LEAF` table are depicted in Figures 4.52(a) and 4.52(b). Suppose we want to detect updated “rank” node. Note that “rank” node is an inlined leaf node. Figure 4.55 depicts the intermediate result of SQL query in Figure 4.54(a) when it detects updated “rank” node. Having determined the updated inlined leaf nodes, the *detectUpdatedLeafNodes* function deletes the corresponding tuples of the updated inlined leaf nodes in the `DEL_LEAF` and `INS_LEAF` by executing the SQL queries depicted in Figures 4.54(b) and 4.54(c), respectively.

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.55: HELIOS-O: Intermediate Result of SQL Query in Figure 4.54(a).

Non-Inlined Leaf Nodes

To detect the absolute updated non-inlined leaf nodes the *detectAbsoluteUpdatedLeafNodes* function shall execute the SQL query depicted in Figure 4.56. We notice that we join three tables, namely, the DEL_LEAF, INS_LEAF, and MATCHING tables. This is because the updated non-inlined leaf nodes are already decomposed into pairs of deleted and inserted non-inlined leaf nodes stored in DEL_LEAF and INS_LEAF, respectively. Line 13 is used to guarantee that the parent nodes of the deleted and inserted leaf nodes are the best matching internal nodes. The absolute updated leaf nodes must have the same node name and the same local order, but different values (lines 15–17). The result of the SQL query depicted in Figure 4.56 is stored in the UPD_LEAF table. We also need to delete the corresponding tuples of the updated inlined leaf nodes in the DEL_LEAF and INS_LEAF by executing the SQL queries depicted in Figures 4.54(b) and 4.54(c), respectively.


```

1  INSERT INTO UPD_LEAF
2  SELECT DISTINCT
3    doc_id1 AS DID1, doc_id2 AS DID2,
4    D.ID AS ID1, I.ID AS ID2,
5    D.LO AS LO1, I.LO AS LO2,
6    D.PID AS PID, I.PID AS PID2, D.NAME,
7    D.VALUE AS VALUE1, I.VALUE AS VALUE2
8  FROM INS_LEAF AS I, DEL_LEAF AS D, MATCHING AS M
9  WHERE
10   I.DID1 = doc_id1 AND I.DID2 = doc_id2 AND
11   D.DID1 = doc_id1 AND D.DID2 = doc_id2 AND
12   M.DID1 = doc_id1 AND M.DID2 = doc_id2 AND
13   M.ID1 = D.PID AND M.ID2 = I.PID AND
14   M.NAME = '[parentNodeName]' AND
15   I.NAME = '[nodeName]' AND
16   D.NAME = '[nodeName]' AND
17   I.VALUE != D.VALUE AND I.LO = D.LO

```

Figure 4.56: HELIOS-O: SQL Queries for Detecting Updated Non-Inlined Leaf Nodes.

Having detected the absolute updated non-inlined leaf nodes, the *changeDetector* algorithm determines the *relative* updated non-inlined leaf nodes by invoking the *detectRelativeUpdatedLeafNodes* function. The *detectRelativeUpdatedLeafNodes* function executes the SQL query depicted in Figure 4.56 after slight modifications. The modification is as follows. We replace “ $I.LO = D.LO$ ” with “ $I.LO \neq D.LO$ ”. Recall that the relative updated leaf nodes must have the same node name, but different values and local orders. We observed that the *detectRelativeUpdatedLeafNodes* function for detecting relative updated non-inlined leaf nodes may return *incorrect* results in some conditions as follows. First, there is more than one relative updated non-inlined leaf node under the same parent nodes. Second, there are deletion/insertion and update of non-inlined leaf nodes occurred under the same parent nodes. Therefore, we use the *refineUpdatedLeafNodes* function to correct the result of the *detectUpdatedLeafNodes* function. The intuition behind the *refineUpdatedLeafNodes* algorithm HELIOS-O in is similar to the one in XANDY-O. The result of the SQL query depicted in Figure 4.56 (after slight modification) is also stored in the UPD_LEAF table.

In our example, the UPD_LEAF table is depicted in Figure 4.51(e). The highlighted tuples in the INS_LEAF (Figure 4.51(c)) and DEL_LEAF (Figure 4.51(d)) tables will be deleted as they are the corresponding tuples of the updated leaf nodes.

Move Operations

The move operations are classified into two categories. First, the moved node changes its position among its siblings in the XML tree. That is, before and after the move operation, it has *same* parent but different position among its siblings. Second, the node (subtree) is moved to be the child of a different parent. That is, before and after the move operation, it has *different* parents, and may have different position among its siblings.

CHAPTER 4. DETECTING THE CHANGES TO ORDERED XML

Figure 4.57: HELIOS-O: SQL Queries for Detecting Moved Nodes.

Move to Different Parent Nodes. A particular node that is moved to different parent node is detected as a pair of deletion and insertion. Hence, we are able to determine the nodes that are moved to different parent nodes by querying the DEL_INT and INS_INT tables (for moved internal nodes), and the DEL_LEAF and INS_LEAF tables (for moved leaf nodes). However, for the moved internal nodes, the subtrees that are rooted at these moved internal nodes should be matching subtrees. This leads us to have a better quality of delta. Note that we only consider the movement of nodes to different parent nodes at the same level. The movement of nodes to different parent nodes at different level will be detected as pairs of deletion and insertion.

Figures 4.57(a) and (b) are used to find the moved *non-inlined* and *inlined* internal nodes that are moved to different parent nodes. Note that we only consider the moved internal nodes that have similarity scores equal or greater than “0.500”. Otherwise, they are detected as pairs of deleted and inserted internal nodes. If an internal node i is moved to different parents, then, intuitively, the subtree rooted at node i is also moved. That is, we need to detect the moved leaf nodes that are the descendants of the moved internal nodes. Figure 4.57(c) is used to find the moved *non-inlined* leaf nodes that are the descendants of the moved internal nodes. To find the *inlined* ones, we used the modified SQL query of the SQL query depicted in Figure 4.57(c). We replace “ID1” and “ID2” in line 10 with “PID1”

and “PID2” respectively. Note that we need to delete the corresponding tuples of the moved nodes that are stored in the `DEL_INT`, `INS_INT`, `DEL_LEAF`, and `INS_LEAF` tables.

We also notice that some leaf nodes can also be moved to be the child nodes of different parent nodes. Note that these moved leaf nodes are not the descendants of the moved internal nodes. Figure 4.57(d) is used to find the moved leaf nodes that are not the descendants of the moved internal nodes. Note that we also need to delete the corresponding tuples of the moved leaf nodes that are stored in the `DEL_LEAF`, and `INS_LEAF` tables.

Move Among Siblings. To detect the movement of nodes among their siblings, we are able to use the similar algorithm as presented in Section 4.4.

4.6 Summary

In this chapter, we have described two approaches for detecting the changes to *ordered* XML documents, namely, `XANDY-O` and `HELIOS-O`. We have presented the extensions of `SUCXENT` schema and *Shared Inlining* schema that are used by `XANDY-O` and `HELIOS-O`, respectively. We have presented the algorithms as well as the SQL queries used in `XANDY-O` and `HELIOS-O` to determine the best matching subtrees (Phase 1). We have elaborated on how `XANDY-O` detects the changes (Phase 2). Finally, we have discussed the algorithms and the SQL queries for detecting the changes (Phase 2) in `HELIOS-O`. In Chapter 5, we shall study the performances of `XANDY-O` and `HELIOS-O` compared to the state-of-the-arts approaches. We shall also analyze the quality of XDeltas detected by `XANDY-O` and `HELIOS-O`.

Chapter 5

Performance Study of Xandy–O and Helios–O

In previous chapter, we have elaborated on our approaches for detecting the changes to *ordered* XML documents, namely, XANDY–O [LB] and HELIOS–O [LB05b]. In this chapter, we shall present the performance study of XANDY–O and HELIOS–O. Note that, in the rest of this chapter, XANDY–O refers to the bottom-up approach of XANDY–O.

This chapter is organized as follows. In Section 5.1, we shall present the experimental settings. In Section 5.2, we shall study the performance of our approaches built on top of two different commercial relational database systems. Section 5.3 shall present the performance study of XANDY–O and HELIOS–O for different numbers of nodes. We shall compare the performances of Phase 1 and Phase 2 in our approaches in Section 5.4. We shall also discuss the effects of file size to the execution times of our approaches in Section 5.5. The performance study on our approaches for different percentages of changes shall be presented in Section 5.6. Furthermore, we shall analyze the execution time contribution of each type of changes to the total execution time of Phase 2 in Section 5.7. This helps us to observe the performance bottleneck in Phase 2. We shall analyze the execution times of XANDY–O and HELIOS–O when there is only a particular type of changes (insertion/deletion/update/move among siblings) in Section 5.8. In Section 5.9, we shall discuss the effects of depth in XML documents to the performance of XANDY–O and HELIOS–O. Section 5.10 shall elaborate the effects of similarity score threshold θ to the performance and result quality of our approaches. We shall analyze the result quality of our approaches compared to the ones of the prior works in Section 5.11. Finally, we shall summarize our discussion in Section 5.12.

Dataset Code	Number of Nodes			Max Fan Out			Filesize (KB)
	Internal	Leaf	Total	SigmodRecord	Articles	Authors	
SIGMOD-01	73	258	331	5	7	7	13
SIGMOD-02	117	427	554	7	9	7	21
SIGMOD-03	187	703	890	9	11	7	34
SIGMOD-04	389	1,437	1,826	14	14	7	70
SIGMOD-05	567	2,151	2,718	17	19	7	104
SIGMOD-06	983	3,734	4,717	23	24	7	180
SIGMOD-07	1,801	6,993	8,794	35	29	7	337
SIGMOD-08	3,883	14,983	18,866	75	29	7	721
SIGMOD-09	7723	30,002	37,725	150	29	7	1,444
SIGMOD-10	18,067	71,256	89,323	150	65	7	3,431
SIGMOD-11	34,845	137,909	172,754	250	79	7	6,635
SIGMOD-12	58,587	231,952	290,539	365	89	7	11,167
SIGMOD-13	71,991	283,930	355,921	625	89	7	13,688
SIGMOD-14	91,604	361,085	452,689	815	89	7	17,398
SIGMOD-15	125,411	494,812	620,223	1065	89	7	23,816

(a) SIGMOD Record Data Sets

Dataset	Number of Nodes			Size (KB)	Dataset	Number of Nodes			Size (KB)
	Internal	Leaf	Nodes			Internal	Leaf	Nodes	
TCSD-01	253	986	1,239	51	TCSD-05	3,390	13,072	16,462	686
TCSD-02	371	1,434	1,805	75	TCSD-06	5,241	20,455	25,696	1,075
TCSD-03	617	2,437	3,054	129	TCSD-07	8,519	33,241	41,760	1,745
TCSD-04	1,036	4,048	5,084	212	TCSD-08	13,984	54,711	68,695	2,842

(b) Dictionary Data Sets

Figure 5.1: Data Sets.

5.1 Experimental Settings and Data Sets

Our approaches, namely, XANDY-O and HELIOS-O are implemented in Java. We used two commercial relational database systems¹ (hereafter called as $\mathcal{RDB}_A$ and $\mathcal{RDB}_B$) for storing XML documents. We conducted the experiments on a Microsoft Windows XP Professional machine having Intel Pentium 4 1.7 GHz processor with 512 MB of memory (unless otherwise stated). Note that the Java applications and RDBMS were running in the same machine. We created two databases for XANDY-O and HELIOS-O. Recall that XANDY-O and HELIOS-O use SUCXENT schema and the Shared-Inlining approach, respectively. We also used tools from $\mathcal{RDB}_A$ and $\mathcal{RDB}_B$ that are used to suggest the indexes based on the query workloads.

We used two synthetic data sets based on the SIGMOD Record DTD [ACM] (SIGMOD Data sets) and Oxford English Dictionary [Dic] (TCSD Data sets) [YÖK04]. SIGMOD data sets represent the *data-centric documents*, and TCSD data sets represent the *text-centric documents*. We generated the second version of each XML document by using our own

¹Due to the license agreement, we cannot mention their name.

CHAPTER 5. PERFORMANCE STUDY OF XANDY-O AND HELIOS-O

Figure 5.2: Experimental Results: RDB_A vs RDB_B (HELIOS-O).

XML changes generator tool. We distributed the percentage changes equally for each type of changes. Figures 5.1(a) and 5.1(b) show the characteristics of the SIGMOD and TCSD data sets, respectively. Note that we focus on the number of nodes in our data sets. This is because the database engine will involve more numbers of tuples for processing when the number of nodes in a tree is increased. We shall elaborate further on this issue in Section 5.5.

We compared the performance of our approaches [LB, LB05b] to the Java version of X-Diff [WDyC03] (downloaded from <http://www.cs.wisc.edu/~yuanwang/xdiff.html>), and C version of XyDiff [CAM02] (downloaded from <http://pauillac.inria.fr/cdrom/www/xydiff/index-eng.htm>). Note that despite our best efforts (including contacting the authors), we could not get the Java version of XyDiff. The C version of XyDiff was run in a Pentium 4 1.7 GHz processor with 512 MB of memory with Red Hat Linux 9 operating system.

Figure 5.3: Query Execution Plans for SQL Query in Figure 4.24(c).

5.2 Different RDBMS

In this section, we analyze the performance of our approaches built on top of different relational database systems. We use SIGMOD data sets (only the first ten data sets) and set the percentages of changes to 3% and 9%. The XML documents are stored in RDB_A and RDB_B before the changes are detected. We used the default settings provided by RDB_A and RDB_B . *Note that this set of experiments is not used to judge the performance of RDB_A (RDB_B), but rather to observe the performance of our approaches when they are built on the top of RDB_A (RDB_B).*

5.2.1 HELIOS-O

Figures 5.2(a) and 5.2(d) depict the performances of the first phase of our approaches. We notice that the performance of HELIOS-O built on top of RDB_A is different from the

Figure 5.4: Query Execution Plans for SQL Query in Figure 4.50(a).

one built on top of $\mathcal{RDB}_B$. The performance of HELIOS-O built on top of $\mathcal{RDB}_B$ is up to 45.1 times faster than the one built on top of $\mathcal{RDB}_A$. This is because of at least the following reasons. $\mathcal{RDB}_A$ and $\mathcal{RDB}_B$ generated different query execution plans for the SQL queries. Obviously, this causes the execution time differences. Consider the SQL query depicted in Figure 4.24(c). We analyzed that $\mathcal{RDB}_A$ and $\mathcal{RDB}_B$ generated different query execution plans. $\mathcal{RDB}_A$ generated a query execution plan as depicted in Figure 5.3(a). Figure 5.3(b) depicts the query execution plan of SQL query in Figure 4.24(c) generated by $\mathcal{RDB}_B$. We notice that $\mathcal{RDB}_A$ uses *hash join* (marked by “*”), while $\mathcal{RDB}_B$ uses *nested-loop join* (marked by “#”).

Figures 5.2(b) and 5.2(e) depict the performances of the second phase of our approaches. Similarly, we also find out that the performance of HELIOS-O built on top of $\mathcal{RDB}_B$ is up to 55.7 times faster than of $\mathcal{RDB}_A$ because of the above reasons. Consider SQL query for detecting insertion/deletion of internal nodes as depicted in Figure 4.50(a). $\mathcal{RDB}_A$ and $\mathcal{RDB}_B$ generated query execution plans as shown in Figures 5.4(a) and 5.4(b), respectively. $\mathcal{RDB}_A$ generated a query execution plan that uses *hash join*, while $\mathcal{RDB}_B$ generated a query execution plan that uses *nested-loop join*.

The overall performance of our approaches is depicted in Figures 5.2(c) and 5.2(f). We notice that the performance of HELIOS-O built on top of $\mathcal{RDB}_A$ is different from the one built on top of $\mathcal{RDB}_B$. The performance of HELIOS-O built on top of $\mathcal{RDB}_B$ is up to 45.8 times faster than of $\mathcal{RDB}_A$. In summary, the performance of HELIOS-O is affected by the choice of relational database systems.

CHAPTER 5. PERFORMANCE STUDY OF XANDY-O AND HELIOS-O

Figure 5.5: Experimental Results: $\mathcal{RDB}_A$ vs $\mathcal{RDB}_B$ (XANDY-O).

5.2.2 XANDY-O

Now, we shall present the performance comparisons of XANDY-O built on top of $\mathcal{RDB}_A$ and $\mathcal{RDB}_B$.

Figure 5.5(a) depicts the performance of the first phase. We observe that as the number of nodes is increased, XANDY-O using $\mathcal{RDB}_B$ performs better than one using $\mathcal{RDB}_A$. For small number of nodes, $\mathcal{RDB}_A$ is up to 9.8 times faster than $\mathcal{RDB}_B$. $\mathcal{RDB}_A$ is up to 6.2 times faster than $\mathcal{RDB}_B$ when the number of nodes is larger than 5000 nodes. Figure 5.5(b) depicts the performance of Phase 2. We analyzed that XANDY-O using $\mathcal{RDB}_B$ and the one using $\mathcal{RDB}_A$ have almost similar performances. The differences between the performances of $\mathcal{RDB}_A$ and $\mathcal{RDB}_B$ are not significant. Figure 5.5(c) depicts the overall performance. Overall performance of XANDY-O using $\mathcal{RDB}_B$ performs better than one using $\mathcal{RDB}_A$ as the number of nodes is increased. For small number of nodes, $\mathcal{RDB}_A$ is up to 7.8 times faster than $\mathcal{RDB}_B$ (SIGMOD-01). $\mathcal{RDB}_A$ is up to 4.8 times faster than $\mathcal{RDB}_B$ for larger data sets. Similarly, the performance of XANDY-O is also influenced by the choice of relational database systems. Hereafter, we only use $\mathcal{RDB}_B$ for storing XML documents before we detect the changes as a representative relational database.

5.3 Numbers of Nodes

In this section, we compare the performances of our approaches for different numbers of nodes in XML documents. We used two data sets: SIGMOD data sets (Figure 5.1(a)) and TCSD data sets (Figure 5.1(b)). We set the similarity score threshold θ to “0.00”. Note that when $\theta = 0.0$, we get the upper-bound of the execution times of our approaches. We only compare the bottom-up approach of XANDY-O to HELIOS-O and other state-of-the-arts approaches.

5.3.1 Data-centric Data Set

Figures 5.6(a) and 5.6(d) depict the performance of Phase 1 in our approaches when the percentages of changes are set to 3% and 9%, respectively. We notice that the performances of HELIOS-O and XANDY-O are comparable until we use SIGMOD-05. For larger data set, HELIOS-O outperforms XANDY-O (up to 20.5 times). We also observe that, for data sets larger than SIGMOD-12, XANDY-O did not return the result after more than 100,000 seconds, hence, we do not plot the result of XANDY-O for data sets larger than SIGMOD-12. The performances of Phase 2 in our approaches when the percentages of changes are set to 3% and 9% are depicted in Figures 5.6(b) and 5.6(e), respectively. We notice that HELIOS-O is faster than XANDY-O (up to 109 times).

Figures 5.6(c) and 5.6(f) depict the overall performance of our approaches when the percentages of changes are set to 3% and 9%, respectively. We compare their performance to the ones of X-Diff and XyDiff (C version). From the experimental results, we can see that the relational-based approaches are more scalable than the memory-based approaches. We notice that X-Diff is only able to detect the changes up to SIGMOD-06 due to lack of main memory. On the other hand, XyDiff fails to detect the changes to data sets larger than SIGMOD-12 as its process was killed by Linux kernel.

Let us now compare the performance of these approaches. We find out that HELIOS-O is up to 21.89 times faster than XANDY-O. Compared to XANDY-O; X-Diff is up to 10.3 times faster for the first four data sets. For “SIGMOD-05” and “SIGMOD-06”, XANDY-O is up to 24.79 times faster than X-Diff. X-Diff overcomes HELIOS-O for the first three data sets (up to 8.15 times). For larger data sets, HELIOS-O is up to 43.7 times faster than X-Diff. Note that the performances of XANDY-O and HELIOS-O is slower than main memory-based approaches for smaller data sets as the database I/O cost is more expensive. We also notice

CHAPTER 5. PERFORMANCE STUDY OF XANDY-O AND HELIOS-O

Figure 5.6: Execution Time vs Number of Nodes (Log scale).

that the difference of execution time between our relational-based approaches and X-Diff reduces as the number of nodes increases. Finally, our relational-based approaches become faster than X-Diff (after SIGMOD-03 data set). This is because the cost of matching process in X-Diff becomes more expensive than the I/O of querying the database.

XANDY-O is less scalable than HELIOS-O because of the following reasons. Recall that in SUCXENT, all the leaf and internal nodes are stored in the `LeafValue` and `AncestorInfo` tables, respectively. In Shared-Inlining technique, the XML documents are partitioned into several relations according to node level. When the number of nodes is increased, obviously, number of tuples in the relations is also increased. However, the increment of numbers of tuples in the `LeafValue` and `AncestorInfo` tables are more significant than the one in the

CHAPTER 5. PERFORMANCE STUDY OF XANDY-O AND HELIOS-O

Figure 5.7: Execution Time vs Number of Nodes (Log scale, 1GB RAM).

relations in Shared-Inlining technique as the increment of numbers of tuples in the relations in Shared-Inlining technique is spread across the relations. In summary, the numbers of tuples in the `LeafValue` and `AncestorInfo` tables are generally larger than the ones in the relations in Shared-Inlining technique. Note that the *inlined* internal/leaf nodes are stored as attributes in their parent relations. Each of the *non-inlined* internal/leaf node is stored in separate relations. This explains why XANDY-O did not return the result after more than 100,000 seconds for data sets larger than SIGMOD-12. Note that the number of tuples in a relation influences the query performance.

In the next set of experiments, we increase the main memory to 1 GB (from previously 512MB). We set the percentage of changes to 3%. The similarity score threshold θ is set to "0.00". This set of experiments is conducted to see the effects of memory size to the memory-based XML change detection approaches. Figure 5.7(a) depicts the performance of Phase 1 in XANDY-O and HELIOS-O. We notice that XANDY-O and HELIOS-O are comparable for the first three data sets. Other than this data set, HELIOS-O is up to 25.34 times faster than XANDY-O. Figure 5.7(b) depicts the performance of Phase 2 in XANDY-O and HELIOS-O. In Phase 2, XANDY-O is 2.3 times faster than HELIOS-O (only for the first data set). Note

that the difference is only around 600 msec. For the second and third data sets, XANDY-O and HELIOS-O are comparable. Afterwards, HELIOS-O outperforms XANDY-O (up to 76 times faster). Figure 5.7(c) depicts the overall performance of all approaches. XANDY-O outperforms HELIOS-O for the first data set. For the larger data sets, HELIOS-O becomes up to 27 times faster than XANDY-O. Compared to X-Diff; for the first three data sets, X-Diff is 3–11 times faster than HELIOS-O. However, for the larger data sets, HELIOS-O outperforms X-Diff (up to 285 times faster). We observe that X-Diff is 1.5–8 times faster than XANDY-O (for the first four data sets). Afterwards, XANDY-O is up to 43 times faster than X-Diff. Compared to the C version of XyDiff; XyDiff is up to 19 times faster than HELIOS-O, and up to 70 times faster than XANDY-O. We believe that the Java version of XyDiff will be much slower and less scalable than the C version and hence will adversely affect the response time and scalability further. In addition, we shall show in Section 5.11 that the result quality of HELIOS-O and XANDY-O is much better than the one of XyDiff. We observe that when we increase the size of main memory, X-Diff and XyDiff become more scalable than previously. For X-Diff, it can detect the changes to XML documents with 8,794 nodes (previously, 4,717 nodes). XyDiff is now able to detect the changes to XML documents with 620,223 nodes (previously, 290,539 nodes). However, increasing the size of main memory is not *a good solution* to have a more scalable memory-based change detection system. This is because there is still a limit of space in main memory to store both XML trees. Therefore, using relational database systems for detecting the changes to XML document is a better and more scalable solution.

Next, we investigate the performance gain of the memory approaches when we increase the size of the main memory. Figure 5.8(a) depicts the performance gain of X-Diff and XyDiff. Note that the ratio is defined as $\frac{a}{b}$, where a is the performance of an approach when the size of memory is 512MB and b is the performance of an approach when the size of memory is 1GB. We notice that X-Diff (1GB) is up to 1.21 times faster X-Diff (512MB). For data set SIGMOD-02, X-Diff (512MB) is 1.1 times faster than X-Diff (1GB); however the execution time difference is only 30 msec. XyDiff gets significant performance gain when the size of main memory is increased to 1GB. XyDiff (1GB) is up to 3.06 times faster XyDiff (512MB). For the first five data sets, XyDiff (1GB) is slower than XyDiff (512MB); however, the execution time differences are between 20–810 msec. This set of experiments shows that the performance of memory-based approaches is improved when we increase the size of main

Figure 5.8: Performance Gain.

memory (for larger data set). This is because when the size of the XML trees in the main memory is larger than the size of main memory, the operating system will use the virtual memory. Note that the virtual memory is kept in the disk. That is, there will be additional I/O cost. When the size of main memory is increased, more portions of XML trees are stored in the main memory instead of in the virtual memory.

Figure 5.8(b) depicts the performance gain of XANDY-O and HELIOS-O. We observe that HELIOS-O (512MB) is faster than HELIOS-O (1GB) for the first six data sets; however, it is not significant as the execution time differences are, in average, 1 second. For larger data sets, HELIOS-O (1GB) is up to 1.5 times faster than HELIOS-O (512MB). For XANDY-O, we notice that for almost all data sets its performance is up to 1.65 times faster when the size of memory is increased to 1GB. In other words, the performance of relational-based approaches is improved when we increase the size of main memory (for larger data set).

5.3.2 Text-centric Data Set

Figure 5.9(a) depicts the performance of Phase 1 in our approaches when the percentage of changes is set to 3%. We notice that XANDY-O is up to 1.26 times faster than HELIOS-O for the first two data sets. This is because in TCSD data sets there are nine non-inlined leaf nodes. This causes HELIOS-O finds the matching leaf nodes groups for each non-inlined leaf nodes. That is, HELIOS-O needs to query nine different relations. As the number of nodes increases, HELIOS-O becomes up to 7.22 times faster than XANDY-O. This is because the number of tuples in the `LeafValue` and `AncestorInfo` tables that store the leaf and internal nodes, respectively, is significantly increased. Observe that XANDY-O uses SUCXENT schema that consists of the `LeafValue`, `AncestorInfo`, `Document`, and `Path` tables. Figure 5.9(b) depicts the performance of Phase 2 in our approaches when the

CHAPTER 5. PERFORMANCE STUDY OF XANDY-O AND HELIOS-O

Figure 5.9: TCSD: Execution Time vs Number of Nodes (Log scale).

percentage of changes is set to 3%. We observe that HELIOS-O is up to 5 times faster than XANDY-O. On average, HELIOS-O is 2.73 times faster than XANDY-O.

Figure 5.9(c) depicts the overall performance of our approaches when the percentage of changes is set to 3%. We observe that XANDY-O is slightly faster than HELIOS-O for the first two data sets; however, the difference is not significant. For larger data sets, HELIOS-O is up to 6.4 times faster than XANDY-O (on average, 3.83 times faster). Let us now compare our approaches to XyDiff and X-Diff. For the first two data sets, X-Diff is up to 2.94 times and 2.51 times faster than HELIOS-O and XANDY-O, respectively. This can be because of the I/O cost needed by our approaches. For larger data sets, HELIOS-O is up to 4.84 times faster than X-Diff. Similarly, XANDY-O is up to 2.77 times faster than X-Diff. Compare to the C version of XyDiff; our approaches are much slower. XyDiff is, on average, 37.5 times and 187.75 times faster than HELIOS-O and XANDY-O, respectively. We also observe that X-Diff becomes less scalable. This is because X-Diff needs more memory space to keep the contents of leaf nodes. Observe that the length of the contents of leaf nodes in TCSD data sets is greater than the one in DCSD data sets.

CHAPTER 5. PERFORMANCE STUDY OF XANDY-O AND HELIOS-O

Figure 5.10: Experimental Results: Phase 1 and Phase 2 Performance Comparison.

5.4 Phase 1 vs Phase 2

In this section, we compare the performance of Phase 1 (“Finding the best matching subtrees” phase) to the one of Phase 2 (“Detecting the Changes” phase).

Figures 5.10(a) and 5.10(b) depict the comparisons between Phase 1 and Phase 2 in HELIOS-O and XANDY-O, respectively. We observe that the performance of Phase 1 in HELIOS-O takes up to 99.70% of the overall performance (“SIGMOD-15”, Figure 5.10(a)). On average, Phase 1 contributes 84.06% of the overall performance (with $\sigma = 0.1414$). Now, let us analyze the comparison between Phase 1 and Phase 2 in XANDY-O. Up to 88.04% of the overall performance of XANDY-O is contributed by the execution time of Phase 1 (“SIGMOD-11”). Phase 1 takes, on average, 69.64% of the overall performance (with $\sigma = 0.109$).

In our approaches, the best matching subtrees are determined iteratively in the bottom-up fashion (Chapter 4). In HELIOS-O, we notice that the execution time of finding matching *non-inlined* node is increase significantly when the data size is increased. In our case, they are the execution times of finding matching “*article*” and “*issue*” nodes. Figure 5.10(c) depicts the execution time of each sub phase in Phase 1. Furthermore, we found out that

CHAPTER 5. PERFORMANCE STUDY OF XANDY-O AND HELIOS-O

Figure 5.11: Experimental Results: Query Execution Plan Comparison.

there are two sub phases that cause the performance degradation, that is, SQL query in Figure 4.28(b) and the sub phase of finding the *best matching internal nodes* (Section 4.3). Note that the execution times of these sub phases are increased significantly compared to other sub phases when the number of nodes is increased. As the number of nodes is increased, we notice that the fan-out of nodes is also increased (see Figure 5.1). As a result the number of nodes in the *weighted bipartite graph* used to find the best matching internal nodes is also increased. Obviously, the performance of finding *maximum weighted bipartite matching* becomes slower. Note that the problem of finding the best matching internal nodes is similar to the problem of finding maximum weighted bipartite matching. In addition, many SQL queries in Phase 1 consist self-joins. Note that self-join operation is still not optimized in RDBMS. Hence, this type of SQL query can be expensive.

We analyzed that the query execution plans of several SQL queries are changed when the number of nodes is increased. For example, the query execution plan of SQL query in Figure 4.28(b) is changed when the number of nodes is increased. Figures 5.11(a) and 5.11(b) depict the query execution plans of this SQL query when we used "SIGMOD-05" and "SIGMOD-11", respectively. In addition, we also observe that the cost of nested loop join in the query execution plan is increased. When "SIGMOD-05" is used, it only takes 5% of the

overall query cost. When we use “SIGMOD-11” as our data set, the cost of nested loop join becomes 67% of the overall query cost. Note that the query execution plans are the ones executed by the database engine.

The degradation of the performance of Phase 1 in XANDY-O is mainly caused by the populating best matching subtrees phase and bottom-up matching process (Figure 5.10(d)). The execution of populate best matching subtrees is significant in Phase 1 because the SQL query for populating best matching subtrees uses self-joins. As the number of tuples in the `TempMatching` table is increased, the cost of this SQL query is also increased significantly. Note that the self-join in relational database is not optimized yet. In the bottom-up matching process, the execution times of SQL query depicted in Figure 4.17(b) and the one of the *maximizeSimilarityScore* function contribute significant percentage to the performance of Phase 1. We observe that the performance of Phase 2 in XANDY-O is also significantly affected by the increment of number of nodes. This is because of the following reason. In SUCXENT, all internal and leaf nodes are stored in the `AncestorInfo` and `LeafValue` tables. That is, when the number of nodes is increased, the numbers of tuples in both tables are also increased. As a result the performance of several SQL queries is affected. The affected SQL queries are ones depicted in Figures 4.34(c) (for detecting insertion and deletion of leaf nodes) and 4.48(b) (for detecting moved leaf nodes to other parent nodes).

5.5 File Size vs Execution Time

In this section, we shall analyze the effect of file size to the performance of our approaches. We use “SIGMOD-06” data set. We fixed the number of nodes and increased the file size by increasing the length of each leaf node value. For each leaf node, we multiply its content length by a factor Ω . Hence, the file size of the XML documents will be changed (from 180KB up to 1,013KB). Figure 5.12(a) depicts the correlation between factor Ω and file size. We set the percentage of changes and similarity threshold θ to “3%” and “0.0”, respectively. Note that this set of experiments also shows us the effects of the increment of content length of leaf nodes to the performance of our approaches.

Figure 5.12(b) depicts the performance of Phase 1 in our approaches for different file sizes. We observe that our approaches are slightly affected by the file size increment. When the file size is increased up to more than 5 times, the execution times of HELIOS-O and XANDY-O are only increased 18.5% and 9%, respectively. Similarly, the performance of

Figure 5.12: Execution Time vs File Size.

Phase 2 in our approaches for different file sizes is not significantly affected (Figure 5.12(c)). The performances of Phase 2 in HELIOS-O and XANDY-O become 31.8% and 17.5% slower, respectively. The overall performance of our approaches for different file size is depicted in Figure 5.12(d). We observe that the overall performance of HELIOS-O and XANDY-O become 21.9% and 11.9% slower, respectively, when we increase the file size up to more than 5 times. The performance of XyDiff becomes up to 31.9% slower as the file size is increased up to 5 times. Figure 5.12(e) depicts the performance of X-Diff for different file size. We observe that the execution time of X-Diff is only increased up to 0.66% when file size is increased up to more than 5 times. The memory-based approaches will become less scalable as the file size is increased. This is because they store XML trees in main memory.

5.6 Percentage of Changes

In this section, we shall observe the effects of percentage of changes to the performances of XANDY-O, HELIOS-O, X-Diff, and XyDiff. We use “Sigmod-03” and “Sigmod-05” data sets and set the similarity score threshold θ to “0.00”. Observe that the percentages of changes are equally distributed to different types of changes.

CHAPTER 5. PERFORMANCE STUDY OF XANDY-O AND HELIOS-O

Figure 5.13: Experimental Results: Execution Time vs Percentage of Changes (SIGMOD-03).

Figures 5.13(a) (for SIGMOD-03 data set) and 5.14(a) (for SIGMOD-05 data set) depict the performance of Phase 1 of XANDY-O and HELIOS-O for different percentages of changes. The performances of XANDY-O and HELIOS-O are affected by the percentages of changes. When the percentage of changes is increased by 1%, the performances of XANDY-O and HELIOS-O become, on average, 0.95% and 1.43% slower, respectively. The performances of Phase 2 of XANDY-O and HELIOS-O for different percentages of changes are depicted in Figures 5.13(b) (for SIGMOD-03 data set) and 5.14(b) (for SIGMOD-05 data set). We observe that the performance of XANDY-O is more affected by percentage of changes compared to HELIOS-O. The performance of XANDY-O becomes, on average, 3.88% slower when the percentage of changes is increased by 1%. On the other hand, the performance of HELIOS-O becomes, on average, 1.24% slower for the same case. Figures 5.13(c) (for SIGMOD-03 data set) and 5.14(c) (for SIGMOD-05 data set) show the overall performance of XANDY-O and HELIOS-O for different percentages of changes. From the experimental result, we observe that XANDY-O, HELIOS-O, and X-Diff are affected by the changes to the percentage of changes. XyDiff is slightly affected by the increment of the percentage of changes.

The performances of our approaches are affected by the percentages of changes because of the following reason. When an XML document is changed, the number of matching

Figure 5.14: Experimental Results: Execution Time vs Percentage of Changes (SIGMOD-05).

subtrees will also be changed in two directions. First, it is possible that the number of matching subtrees is reduced. This happens when the values of inserted leaf nodes and/or the new values of updated leaf nodes are distinct from the ones in the first version of an XML document. Note that matching subtrees must have at least one matching leaf nodes (Definition 3.5). Second, it is also possible that the number of matching subtrees is increased. This case is a general case and often happens in real world. When the number of matching subtrees is increased, the number of tuples storing this information is also increased. The time for finding the best matching subtrees becomes slower. In our experiments, the second case was happened. Therefore, we observe that as the XML documents are changed more significantly, the execution time of Phase 1 of our approaches is increased. For Phase 2, when the percentage of changes is increased, our approaches have to discover more changed nodes. Consequently, the performance of detecting the changes is affected.

Now, we shall elaborate on the performance of Phase 2 that is depicted in Figure 5.14(b). We observe that the execution time of XANDY-O when the percentage of changes is set to 18% is higher than the one when the percentage of changes is set to 24%. Figure 5.14(d) depicts the execution times of sub phases in Phase 2. The execution time of detecting the absolute update operations explains the above situation. Furthermore, we analyze the query

Figure 5.15: Query Execution Plan.

execution plans of SQL query used for detecting the absolute update operations when the percentages of changes are set to 15%, 18%, and 24%. Figure 5.15(a) depicts the query execution plans of SQL query used for detecting the absolute update operations when the percentages of changes are set to 15% and 18%. The query execution plans of SQL query used for detecting the absolute update operations when the percentage of changes is set to 24% is shown in Figure 5.15(b). We notice that there is a slight change to the query execution plans. The dotted box in Figure 5.15(b) is the change to the query execution plan by the Optimizer.

5.7 Execution Time Proportion of Different Types of Changes

In this set of experiments, we shall see the execution time proportion of each type of changes. We used first seven data sets and set the percentages of changes to 3% and 9%.

5.7.1 HELIOS-O

Figures 5.16(a) and 5.16(b) depict the proportion of execution times of detecting insertion, deletion, updates, and moves when the percentages of changes are set to 3% and 9%, respectively. We notice that detecting move operation takes up to 67.17% of the total execution

CHAPTER 5. PERFORMANCE STUDY OF XANDY-O AND HELIOS-O

Figure 5.16: HELIOS-O: Execution Time Proportion for Different Types of Changes.

time of Phase 2. The execution times of detecting insertions and deletions contribute up to 23.18% and 23.46%, respectively. The execution time of detecting updates is up to 31.61% of the total execution time of Phase 2.

The performance contributions of insertion/deletion internal and leaf nodes are depicted in Figures 5.16(c) and 5.16(d) when the percentages of changes are set to 3% and 9%, respectively. We observe that detecting the insertions/deletions leaf node has more contribution to the total execution time of Phase 2. Detecting the insertion of leaf nodes takes up to 19.71% of the total execution time of Phase 2. On the other hand, detecting the insertion of internal nodes only contributes up to 10.59%. Detecting the deletion of leaf nodes takes up to 17.17% of the total execution time of Phase 2. Detecting the insertion of internal

nodes only contributes up to 6.30%. Furthermore, we analyze the reason behind the higher execution time percentage of detecting the insertion/deletion of leaf nodes. We observe that this happens because of SQL query in Figure 4.53(c). When an internal node has more than one *inlined* leaf nodes, the **WHERE**-clause of this SQL query will contain OR-condition and AND-condition. Each OR-condition essentially forces the compiler to run a separate query for each OR-condition. Hence, the performance of this SQL query is slower than the execution time of SQL queries for detecting insertion/deletion internal nodes.

Figures 5.16(e) and 5.16(f) depict the execution time proportions of detecting moved internal nodes, moved leaf nodes, and moves among siblings when the percentages of changes are set to 3% and 9%, respectively. We notice that the execution time of detecting moves among siblings is more significant compared to the ones of detecting moved internal nodes and moved leaf nodes. It takes up to 55.68% of the total execution time of Phase 2. The execution times of detecting moved internal nodes and moved leaf nodes contribute up to 16.36% and 19.13%, respectively, of the total execution time of Phase 2. The execution time of detecting moves among siblings is high because of the function of simulating the changes. Furthermore, we analyze that the increments of number of nodes and percentage of changes will affect its execution time. When we have more nodes in the XML, we shall have more changed nodes if we fix the percentage of changes. On the other hands, if we fix number of nodes and increase the percentage of changes, then there will be more changed nodes. The increment of numbers of changed nodes obviously will affect the performance of function of simulating the changes. In addition, more number of nodes means more tuples involved when we generate the `MoveList` table (Chapter 4).

5.7.2 XANDY-O

Figures 5.17(a) and 5.17(b) depict the proportion of execution times of detecting insertion, deletion, updates, and moves when the percentages of changes are set to 3% and 9%, respectively. Similar to HELIOS-O, detecting move operation is the most significant factor in the execution time of Phase 2. It takes up to 70.80% of the execution time of Phase 2. This is because of the same reason in the above discussion. On average, it takes 45.18% of the execution time of Phase 2 (with $\sigma = 13.33$). The execution times of detecting insertions and deletions contribute up to 32.23% and 28.04%, respectively. We notice that the execution time of detecting updates takes up to 28.14% of the execution time of Phase 2.

CHAPTER 5. PERFORMANCE STUDY OF XANDY-O AND HELIOS-O

Figure 5.17: XANDY-O: Execution Time Contribution of Different Types of Changes.

Let us now analyze the performance contributions of insertion/deletion internal and leaf nodes to the performance of Phase 2. The performance contributions of insertion/deletion internal and leaf nodes are depicted in Figures 5.17(c) and 5.17(d) when the percentages of changes are set to 3% and 9%, respectively. For small data sets, we notice that detecting the insertions/deletions internal node has more contribution to the total execution time of Phase 2. Detecting the insertion of internal nodes takes up to 23% of the total execution time of Phase 2, while detecting the deletion of internal nodes only contributes up to 18.20%. On the other hand, for larger data sets, detecting the insertions/deletions leaf node has more contribution to the total execution time of Phase 2. Detecting the insertion of leaf nodes and the deletion of leaf nodes take up to 17.26% and 14.58%, respectively. In general,

%	Phase 1	Phase 2	Total	%	Phase 1	Phase 2	Total	%	Phase 1	Phase 2	Total
1	5.866	1.138	7.004	1	5.620	1.138	6.758	1	5.824	1.293	7.117
3	5.953	1.229	7.182	3	5.702	1.496	7.198	3	5.955	1.268	7.223
6	6.051	1.900	7.951	6	6.064	1.352	7.416	6	5.848	1.293	7.142
9	6.039	1.782	7.821	9	6.111	1.857	7.968	9	6.186	1.426	7.612

(a) Deletion (b) Insertion (c) Append

%	Phase 1	Phase 2	Total	%	Phase 1	Phase 2	Total
1	5.729	1.117	6.845	1	5.782	1.148	6.930
3	5.805	2.066	7.870	3	5.604	1.106	6.710
6	5.761	2.016	7.777	6	5.988	1.094	7.082
9	5.825	1.960	7.785	9	5.929	0.911	6.840

(d) Update (e) Move among siblings

Figure 5.18: HELIOS-O: Different Types of Changes.

detecting the insertion/deletion of leaf nodes is more expensive than the one of internal nodes. This is because we have to use two SQL queries to retrieve the inserted/deleted leaf nodes. Recall that the inserted/deleted leaf nodes can be either in best matching subtrees or inserted/deleted subtrees. Furthermore, we analyze that the SQL query for detecting the inserted/deleted leaf nodes in the best matching subtrees (Figure 4.34(c)) is more expensive than the one for detecting the inserted/deleted leaf nodes in the inserted/deleted subtrees (Figure 4.34(b)). This is because the SQL query in Figure 4.34(c) has several sub queries.

Figures 5.17(e) and 5.17(f) depict the execution time proportions of detecting moved internal nodes, moved leaf nodes, and moves among siblings when the percentages of changes are set to 3% and 9%, respectively. Similar to HELIOS-O, detecting the move among siblings takes major percentage of the overall execution time of Phase 2. It takes up to 59.04% of the overall execution time of Phase 2. On average, it contributes 27.13% of the overall execution time of Phase 2 (with $\sigma = 14.128$). The performance of detecting move among siblings becomes worst when the data sets become larger. This is because of the similar reasons as discussed in the previous section.

5.8 Different Types of Changes

In this section, we shall observe the effects of percentage of different types of changes to the performances of XANDY-O and HELIOS-O. We use “Sigmod-05” data set as the first version and set similarity score threshold θ to “0.00”. The percentages of changes are between “1%” to “9%”. We generated four second versions by using our changes generator such that the changes are content updates only (denoted by “Update”), deletions only (denoted by

“Deletion”), insertions only, and moves among siblings (denoted by “Move among siblings”). We observe that the insertions can occur in two following ways. First, the insertion may occur in any parts of the XML documents (denoted by “Insertion”). Second, the insertion may occur only at the end of the XML documents (denoted by “Append”). The later case may be happened in real life XML documents, such as DBLP XML records [Bib].

5.8.1 HELIOS-O

Figure 5.18 depicts the performance of HELIOS-O. We observe that the performance of Phase 1 in HELIOS-O is not significantly affected. The difference of the maximum and minimum execution times is less than 1 second. Similarly, the performance of Phase 2 in HELIOS-O is also not significantly affected as the difference of the maximum and minimum execution times is only at most 0.949 second (Figure 5.18(d)). For the overall performance of HELIOS-O, we also notice that its performance is not significantly affected.

Furthermore, we analyze the execution time of each step in Phase 2. When there are only deletions, the execution time of detecting the deletions contributes, on average, 24.04% of the overall performance of Phase 2. Even though there is no move operation, we find out that the execution times of detecting move to different parent node and move among siblings are, on average, 18.67% and 26.90% of the overall performance of Phase 2. For detecting moves among siblings, we need to retrieve all nodes that are possible to move among siblings. After simulating the changes, we delete the tuples whose local order in the first and second versions are same. We observe that, whether or not there are moves among siblings, there is a minimal cost of detecting moves among siblings. Unfortunately, the minimal cost is relatively higher than detecting other types of changes. When there are only insertions, the execution time of detecting the insertions (“Insertion”) contributes, on average, 21.35% of the overall performance of Phase 2. We observe that 20.02% of the overall performance of Phase 2 is taken by the execution time of detecting the insertions (“Append”) when there are only append operations. Our observation also indicates that the execution time of detecting moves among siblings is dominant in the overall performance of Phase 2.

Next, we observe that the execution time of detecting the update operation takes, on average, 26.16% of the overall performance of Phase 2. The execution time of detecting moves among siblings takes 24.56% of the overall performance of Phase 2. Even though detecting the update operation is slightly more expensive than detecting moves among

%	Phase 1	Phase 2	Total	%	Phase 1	Phase 2	Total	%	Phase 1	Phase 2	Total
1	10.888	3.755	14.643	1	11.003	3.603	14.606	1	11.002	3.562	14.564
3	11.085	3.944	15.028	3	11.450	3.716	15.165	3	11.053	3.714	14.767
6	10.832	4.580	15.412	6	10.260	4.396	14.656	6	10.915	3.657	14.573
9	10.563	5.182	15.746	9	10.094	4.620	14.714	9	11.479	3.568	15.047

(a) Deletion

(b) Insertion

(c) Append

%	Phase 1	Phase 2	Total	%	Phase 1	Phase 2	Total
1	10.807	3.774	14.581	1	10.954	3.633	14.587
3	10.777	4.100	14.877	3	11.330	3.747	15.077
6	10.525	4.308	14.833	6	10.280	3.842	14.122
9	10.101	5.895	15.996	9	9.384	4.079	13.463

(d) Update

(e) Move among siblings

Figure 5.19: XANDY-O: Different Types of Changes.

siblings, we still consider the detecting moves among siblings is expensive as there is no move among sibling in this case. Furthermore, we see that detecting the relative updates contributes 17.65% of the overall performance of Phase 2. When there are only move operations, the execution time of detecting move operations takes, on average, 37.18% of the overall performance of Phase 2.

5.8.2 XANDY-O

Figure 5.19 depicts the performance of XANDY-O. We observe that the performance of Phase 1 in XANDY-O is not significantly affected. The difference of the maximum and minimum execution times is less than 2 seconds. The performance of Phase 2 in XANDY-O is also not significantly affected when there are only append operations. Otherwise, we observe that the performance of Phase 2 in XANDY-O is affected. This indicates that the performance of XANDY-O is sensitive to the percentage of changes of these changes. For the overall performance of XANDY-O, we also notice that its performance is affected. From this set of experiments, we observe that detecting the changes to leaf nodes and move operations contributes significantly to the performance of Phase 2 in XANDY-O.

5.9 Depths of XML Documents

In this section, we analyze the effects of the depth of XML documents to the performance of our approaches.

5.9.1 XANDY-O

We generate a set of synthetic XML documents whose maximum node levels are between 5 and 14. The number of nodes of the first versions is fixed to 2860. The percentage of changes and the similarity score threshold θ are set to “3%” and “0.00”, respectively. Figure 5.20(a) depicts the performance of XANDY-O for different maximum node levels. Phase 1 is significantly influenced by the depth of XML documents. We analyze that the performance of XANDY-O becomes 12.63% slower when the depth of XML documents is increased from 5 to 9. Similarly, its execution time is increased around 12.03% when the depth of XML documents is increased from 9 to 14. Furthermore, we observe that the *finding matching sibling order* phase and the *bottom-up matching* phase cause this performance degradation. The *finding matching sibling order* phase becomes slower when the depth is increased because of the following reason.

The execution time of the *bottom-up matching* phase is increased because the number iteration in the *findBestMatchingSubtrees* algorithm (lines 4–8, Figure 4.11) is increased. That is, more SQL queries will be invoked. As a result the overall performance of the *bottom-up matching* phase becomes slower. Next, we observe the performance of Phase 2 in XANDY-O for different maximum node level. The performance of Phase 2 in XANDY-O is relatively stable. In other words, it is not significantly influenced by the increment of depth in XML documents. This is because the number of executed SQL queries is not changed. In addition, the number of nodes, the percentage of changes, and the similarity score threshold are not changed. The overall performance of XANDY-O is affected by the changes to the maximum node levels. This is because the execution time of Phase 1 is increased. In addition, most of SQL queries in Phase 1 use self-join. Observe that the self-join is not optimized yet by RDBMS. This exacerbates the performance of Phase 1.

5.9.2 HELIOS-O

For HELIOS-O, the changes to the maximum node levels will cause the changes to the numbers of *non-inlined* nodes. Hence, we analyze the performance of HELIOS-O for different levels by increasing numbers of *non-inlined* leaf nodes. We increase the number of *non-inlined* leaf nodes up to three additional *non-inlined* leaf nodes while the numbers of nodes in the first versions of XML documents are fixed to 2345 nodes. Figure 5.20(b) depicts the performance of HELIOS-O for different numbers of non-inlined leaf nodes. We observed

Figure 5.20: Experimental Results: Depths.

that HELIOS-O is influenced by number of *non-inlined* leaf nodes. In HELIOS-O, we shall have more relations for storing the *non-inlined* leaf nodes. The size of each relation will be smaller. That is, we have less number of tuples to join for each SQL query.

5.10 Similarity Score Threshold

In this section, we shall analyze the effects of similarity score threshold θ to our approaches. First, we shall analyze the influences to the execution time. Then, we shall observe the result quality for different values of similarity score threshold θ .

5.10.1 Similarity Score Threshold vs Execution Time

In this set of experiments, we set the value of similarity score threshold θ between “0.00” and “0.60”. We use “Sigmod-05” data set with “3%” and “9%” changes.

Figures 5.21(a) and 5.21(d) depict the performance of Phase 1 in HELIOS-O and XANDY-O when the percentages of changes are set to 3% and 9%, respectively. We observe that as the similarity score threshold θ increases, the execution time of Phase 1 in HELIOS-O and XANDY-O decreases. This is because as follows. Suppose HELIOS-O/XANDY-O matches subtrees at level l . When it finds out that some of these matching subtrees have similarity score less than the similarity score threshold θ , it deletes these subtrees from the TempMatch table. Hence, the number of matching subtrees at level l is reduced. This causes the number of matching subtrees at level $l - 1$ is also reduced. We can see that as the value of similarity score threshold θ becomes larger, the number of matching subtrees that are deleted as their have similarity scores are less than the similarity score threshold θ is increased. This means that the number of subtrees to be matched at the lower level is lesser. That is, if the number of matching subtrees for each level is lesser, then it will cause the faster performance.

CHAPTER 5. PERFORMANCE STUDY OF XANDY-O AND HELIOS-O

Figure 5.21: Similarity Score Threshold θ vs Execution Time.

Figures 5.21(b) and 5.21(e) depict the performance of Phase 2 in HELIOS-O and XANDY-O when the percentages of changes are set to 3% and 9%, respectively. The execution time of our approaches increases as the similarity score threshold θ becomes larger. This is because the number of best matching subtrees in T_1 and T_2 is lesser. In other words, there are lesser number of subtrees in T_1 that are matched to the ones in T_2 . Hence, we shall have more changed nodes to be discovered. Figures 5.21(c) and 5.21(f) depict the overall performance of HELIOS-O and XANDY-O when the percentages of changes are set to 3% and 9%, respectively. We notice that the overall performance are slightly affected by the changes to the similarity score threshold θ . As the similarity score threshold θ increases, our approaches become slightly faster. Observe that the overall performance of our approaches

Figure 5.22: Similarity Score Threshold θ vs Result Quality.

is significantly influenced by the performance of Phase 1.

5.10.2 Similarity Score Threshold vs Result Quality

In this set of experiments, we analyze the *result quality* of XANDY-O and HELIOS-O for different similarity score thresholds θ . We use “Sigmod-05” data set. Then two new versions are generated by setting the percentages of the changes to 3% and 15%. For each percentage of changes, XANDY-O and HELIOS-O were run by varying similarity score threshold θ . The *result quality* in this set of experiments is defined as $\frac{p}{q}$, where p is the number of edit operations in delta detected by an approach when similarity score threshold is set to x ($0.10 \leq x \leq 0.50$), and q is the one in the delta detected by an approach when similarity score threshold is set to “0”.

The ratios are plotted in Figures 5.22(a) and 5.22(b). We observe that the similarity score thresholds θ may not significantly affect the result quality if the documents are changed slightly. On the other hand, the result quality is affected by the similarity score thresholds θ if the documents are changed significantly. Now, let us compare the result quality of our approaches in Figures 5.22(a) and 5.22(b) when $\theta = 0.3$. We observe that even when the similarity score threshold θ is not changed, the result quality can be reduced if the percentage of changes is increased. We conclude that the result quality of the deltas detected by XANDY-O and HELIOS-O is influenced by the percentage of changes and the similarity score thresholds θ . In addition, we also analyze that the result quality of the deltas detected by XANDY-O and HELIOS-O is also influenced the distribution of the changes. The distribution of the changes influences the result quality in the following way. Suppose we have subtree t_1 and t_2 in the first and second versions, respectively, $t_1 \bowtie t_2$, and the percentage of changes is equal to $p\%$. There will be two cases as following. The first case is t_1 and t_2 are slightly

Figure 5.23: Experimental Results: Result Quality.

Figure 5.24: Result Quality: Examples (1).

different. This happens when the changes are distributed to all subtrees. The second case is t_1 and t_2 are significantly different. This happens when the changes are concentrated in t_1 . Suppose we set the similarity score threshold equal to z . It is possible that, in the first case, $\mathcal{R}(t_1, t_2) \geq z$, but, in the second case, $\mathcal{R}(t_1, t_2) < z$. Obviously, in the second case, t_1 and t_2 are considered as unmatching subtrees. Consequently, the result quality in the first case is higher than the one in the second case.

5.11 Result Quality

In this section, we analyze the result quality of HELIOS-O and XANDY-O. We also compare the result quality of HELIOS-O and XANDY-O to the ones of XyDiff [CAM02] and X-Diff [WDyC03].

First, we compared the result quality of HELIOS-O, XANDY-O and X-Diff. We used data set “SIGMOD-02”, and constructed the second versions with various percentages of changes. As X-Diff addressed the problem of detecting the changes to unordered XML documents, we do not consider the move operations in computing the result quality. Figure 5.23(a) depicts the result quality comparisons between HELIOS-O, XANDY-O and X-Diff. We observe that the result quality of HELIOS-O is almost similar to the one of XANDY-O. The result quality

Figure 5.25: Result Quality: Examples (2).

of HELIOS-O is comparable to the one of X-Diff. In some cases, the result quality of X-Diff can be better than the ones of HELIOS-O and XANDY. Note that X-Diff does not detect the move operations as it is designed for unordered XML documents. Let us elaborate on this further by using an example. Consider the example depicted in Figure 5.24(a). HELIOS-O and XANDY-O shall detect deltas that consist of two update operations (nodes 3 and 4), a deletion of a subtree (subtree rooted at node 5), and an insertion of a subtree (subtree rooted at node 15). X-Diff shall detect a delta that consists of a deletion of a subtree (subtree rooted at 1) and an insertion of a subtree (subtree rooted at node 11). We notice that, in some other cases, the result qualities of HELIOS-O and of XANDY-O are better than the one of X-Diff. Consider the example depicted in Figure 5.24(b). X-Diff shall detect as two update operations (nodes 2 and 3). Note that updating the “ID” node is not *semantically* correct as ID number should not be updated. However, HELIOS-O and XANDY-O shall detect as a deletion of a subtree rooted at node 1 and an insertion of a subtree rooted at node 5. That is, X-Diff detects as a set of update operations, and HELIOS-O and XANDY-O detect as a pair of deletion and insertion of subtrees.

Next, we compare the result quality of HELIOS-O, XANDY-O and XyDiff. We use a small data set (based on SIGMOD Record DTD) with 100 nodes and generate the second versions with various percentages of changes (2%–12%). We compare the results of HELIOS-O, XANDY-O, and XyDiff with the optimal deltas. We observe that the result quality of HELIOS-O and XANDY-O are comparable. The result qualities of HELIOS-O and XANDY-O are better than the one of XyDiff. In XyDiff’s deltas, we observe that there are some unnecessary move operations, and, in some case, XyDiff mismatches the best matching subtrees. Consider the example depicted in Figure 5.25. The delta detected by XANDY-O contains *delete*(1) and *update*(10, “Asst Prof”, “Assoc Prof”). However, the delta generated

by XyDiff contains $move(9, 1, 2)^2$, $delete(8)$, and $delete(2)$. Note that the delta detected by HELIOS-O and XANDY-O is optimal delta. Note that this issue has also been discussed in [WDyC03].

5.12 Summary

We have presented our performance study of our approaches compared to other state-of-the-arts approaches. We summarize our performance study as follows.

- We find out that our relational-based approaches are RDBMS-dependent. This means the performance of our approaches are influenced by the choice of relational database systems.
- The relational-based approaches are more scalable than the state-of-the-arts memory-based approaches. We observe that X-Diff is not able to detect the changes to the data sets larger than SIGMOD-06/TCSD-04 due to lack of memory. XyDiff (C version) fails to detect the changes to the data sets larger than SIGMOD-12 as its process was killed by Linux kernel.
- For DCSD data sets, XANDY-O is up to 25 times faster than X-Diff for SIGMOD-05 and SIGMOD-06 data sets. HELIOS-O is faster X-Diff (up to 44 times faster) for larger data sets. Compared to XANDY-O; HELIOS-O is up to 109 times faster. We also find out that HELIOS-O is more scalable than XANDY-O. Compared to XyDiff; HELIOS-O has comparable performance. Note that we compare our approaches to the C version of XyDiff. We also notice that HELIOS-O outperforms XANDY-O (up to 22 times faster).
- When we use TCSD data sets, we observe that our approaches are up to 5 times faster than X-Diff for data sets larger than TCSD-02. However, our approaches are slower than XyDiff. XyDiff is up to 187 times faster than our approaches. HELIOS-O also outperforms XANDY-O.
- HELIOS-O is more scalable than XANDY-O as XANDY-O does not return results for data sets larger than SIGMOD-12 after 100,000 seconds.

²This operation means “move node 9 to the second child node of node 1”

- The number of nodes has more effects to the performance of our approaches compared to the file size. When the file size is increased up to more than 5 times, the performance of our approaches becomes up to 21% slower.
- Phase 1 of our approaches is the bottle neck. It can take up to more than 90% of the overall performance.
- Detecting move operations takes up to more than 65% of the execution time of Phase 2. The performance of detecting insertions/deletions of leaf nodes is slower than the one of detecting insertions/deletions of internal nodes. In detecting move operations, we discover that detecting moves among sibling is more expensive than detecting the moved internal/leaf nodes to other parent nodes. Then, we present the experiments to see the effect of having only a particular type of changes to the performance of our approaches.
- The percentages of changes indeed have influences to the performance of our approaches. When the percentage of changes is increased by 1%, the performance of our approaches becomes, on average, 3% slower.
- As our approaches find the best matching subtrees in bottom-up fashion, we also present the performance of our approaches for different depths of XML documents. In XANDY-O, we find out that, indeed, the performance of Phase 1 is affected, specifically, the performance of bottom-up matching phase. We also notice that the performance of HELIOS-O is also affected by the depths of XML documents.
- In our approaches, we calculate the similarity score in order to know how similar two subtrees are. We also define the similarity score threshold (called θ). We did a set of experiments in order to know the effects of similarity score threshold θ to the performance and result quality of our approaches. We find out that the performance and result quality of our approaches are affected. The performances of both Phase 1 and Phase 2 are slightly affected by the changes to similarity score threshold θ . The result quality of our approaches is also affected by similarity score threshold θ . In addition, we also show that the result quality is also influenced by the percentages of changes.
- Our approaches have superior result quality compared to XyDiff. We also find out that the result qualities of X-Diff and our approaches are comparable.

Chapter 6

Detecting the Changes to Unordered XML

In Chapter 4, we have presented XANDY-O and HELIOS-O that are the variants of XANDY and HELIOS, respectively, for detecting the changes to *ordered* XML documents. In this chapter, we shall elaborate on XANDY-U [LBM05] and HELIOS-U [LB05a] that address the problem of detecting the changes to *unordered* XML documents. Note that we have presented the overview of our approaches and the terminology that will be used in this dissertation in Chapter 3. Note that the preliminary works of these approaches can be found in [LBM05, LB05a].

This chapter is organized as follows. In Section 6.1, we shall discuss how the relational schemas that are used for storing XML documents before the changes are detected are extended. Section 6.2 shall elaborate how XANDY-U determines the best matching subtrees (Phase 1). We shall discuss Phase 1 of HELIOS-U in Section 6.3. Sections 6.4 and 6.5 shall discuss how XANDY-U and HELIOS-U, respectively, detect the changes after determining the best matching subtrees. Finally, we shall summarize the discussion in this chapter in Section 7.11. Note that the performance study of XANDY-U and HELIOS-U will be presented in Chapter 7.

6.1 The Relational Schema Extensions

In this section, we shall present the extensions of SUCXENT and *Shared-Inlining* schemas. Note that XANDY-U and HELIOS-U use SUCXENT and *Shared-Inlining* schemas, respectively, for storing XML documents before detecting the changes. The extensions are necessary as the properties of existing relational schemas are not enough to be used for detecting the changes to unordered XML documents.

Document (<u>Doc_ID</u> , Doc_Name)
Path (<u>Path_ID</u> , Path_Exp)
LeafValue (<u>Doc_ID</u> , <u>LeafOrder</u> , <u>Path_ID</u> , <u>Level</u> , SiblingOrder, LeafValue, LeftSibIxNLevel)
AncestorInfo (<u>Doc_ID</u> , <u>NodeLevel</u> , <u>MinSibOrder</u> , <u>MaxSibOrder</u> , NodeName)

Figure 6.1: XANDY-U: Modified SUCXENT Schema.

AncestorInfo					LeafValue						
Doc_ID	Node Level	MinSib Order	MaxSib Order	NodeName	Doc_ID	Leaf Order	Path_ID	Level	Sibling Order	LeafValue	LeftSib IxnLevel
1	1	1	8	SigmodRecord	1	1	1	3	1	10	-1
1	2	1	5	issue	1	2	2	3	1	1	-1
1	3	2	5	articles	1	3	3	5	2	Web Mining	2
1	4	2	3	article	...	...	...	...	...	...	...
...	...	...	...	...	1	8	6	6	3	Utt	4
2	1	1	8	SigmodRecord	...	...	...	...	...	...	...
2	2	1	3	issue	2	1	1	3	1	10	-1
2	3	2	3	articles	2	2	2	3	1	1	-1
2	4	2	3	article	2	3	3	5	2	Web Usage Mining	2
...	...	...	...	...	...	...	...	...	...	...	...
2	5	8	8	authors	2	20	6	6	8	Greg	4

Path		Document	
Path_ID	Path_Exp	Doc_ID	Doc_Name
1	/SigmodRecord/issue/volume	1	Sigmod-01.xml
2	/SigmodRecord/issue/number	2	Sigmod-02.xml
3	/SigmodRecord/issue/articles/article/title		
4	/SigmodRecord/issue/articles/article/initPage		
5	/SigmodRecord/issue/articles/article/endPage		
6	/SigmodRecord/issue/articles/article/authors/author		

Figure 6.2: XANDY-U: XML in RDBMS.

6.1.1 XANDY-U

The extension of SUCXENT schema for XANDY-U is different from the one for XANDY-O. In SUCXENT schema for XANDY-U, we do not need to keep the local order of nodes as XANDY-U addresses the problem of detecting the changes to *unordered XML*. Consequently, the extension SUCXENT schema for XANDY-U is as follows. We add the *Level* attribute in the *LeafValue* table to store the level of leaf nodes. The extended SUCXENT schema for XANDY-U is depicted in Figure 6.1. Consider two versions of an unordered XML document as depicted in Figures 1.7(a) and 1.7(b). Figure 6.2 depicts these XML documents shredded in relational database (partial view only).

6.1.2 HELIOS-U

Similarly, the modification of the *Shared-Inlining* schema for HELIOS-U is different from the one for HELIOS-O. The difference is the same as above, that is, we do not need to keep

Figure 6.3: HELIOS-U: DTD and Modified Schema.

the local order of nodes. The *Shared-Inlining* schema for HELIOS-U is extended as follows. We add the **Document** table for storing different versions of XML documents. We store the information on *inlined* internal nodes as a **BOOLEAN** attribute in its parent relation. We also add the *Doc_ID* attribute to all relations as we need to store more than one version of XML documents. Suppose two versions of an unordered XML document as depicted in Figures 1.7(a) and 1.7(b) have DTD as depicted in Figure 6.3(b). Its DTD tree is shown in Figure 6.3(a). The modified relational schema is depicted in Figure 6.3(c). Observe that the **Issue** and **Article** tables in the original schema (Figure 4.4(c)) do not have the *Articles* and *Authors* attributes, respectively. Figure 6.4 depicts these XML documents shredded in relational database.

6.2 Finding Best Matching Subtrees Phase in XANDY-U

In this section, we shall elaborate on how XANDY-U determines the best matching subtrees. XANDY-U finds the best matching subtrees in bottom-up fashion. This is because of the reasons discussed in Chapter 4.

The *findBestMatchingSubtrees* algorithm in XANDY-U is similar to the one in XANDY-O that is depicted in Figure 4.11 (without line 9). Line 9 is used to collect the best matching subtrees from the **TempMatching** table. Recall that XANDY-O stores the best matching subtrees and the moved subtree candidates in the **TempMatching** table. As we

Document

Doc_ID	Doc_Name
1	Sigmod-01.xml
2	Sigmod-02.xml

SigmodRecord

Doc_ID	ID	PID
1	1	-1
2	2	-1

Author

Doc_ID	ID	PID	Value
1	1	1	Bobby
1	2	1	Marcella
...	...	...	...
1	8	3	Greg
2	1	1	Utt
2	2	1	Bobby
...	...	...	...
2	7	3	Greg

Article

Doc_ID	ID	PID	Title	initPage	endPage	Authors
1	1	1	Web Mining	1	9	TRUE
1	2	1	XML Mining	10	20	TRUE
1	3	2	Text Mining	1	12	TRUE
2	1	1	Web Usage Mining	1	9	TRUE
2	2	2	Text Mining	1	12	TRUE
2	3	2	Active Database	13	30	TRUE

Issue

Doc_ID	ID	PID	Volume	Number	Articles
1	1	1	10	1	TRUE
1	2	1	10	3	TRUE
2	1	1	10	1	TRUE
2	2	1	10	2	TRUE

Figure 6.4: HELIOS-U: XML in RDBMS.

address the problem of detecting the changes to *unordered* XML documents, there will no moved subtree candidates. Such subtrees will be directly deleted as they are not in the *best matching configurations*.

The *findBestMatchingSubtrees* algorithm works as follows. Suppose we have two versions of an unordered XML document, namely, D_1 and D_2 , shredded in relational database. The tree representations of D_1 and D_2 are T_1 and T_2 , respectively. First, the *findBestMatchingSubtrees* algorithm compares the root nodes of T_1 and T_2 . If their node names are different, then we assume that T_1 and T_2 are different. Consequently, the XDelta will consist of a deletion of T_1 and an insertion of T_2 . Otherwise, XANDY-U shall determine the *matching sibling orders*. Then, the algorithm starts propagating the matchings in bottom-up fashion. For each level, it finds the matching subtrees. As a subtree in T_1 can be matched to more than one subtree in T_2 , XANDY-U needs to determine the best matching subtrees for a set of matching subtrees. We shall elaborate further on each step.

6.2.1 Finding Matching Sibling Orders

The matching sibling orders can be determined by using the *findMatchingSiblingOrder* algorithm that is depicted in Figure 6.5. Let us elaborate further on each phase. The first step is to find the matching leaf nodes (line 1, Figure 6.5(a)). The SQL query used to find the matching leaf nodes is depicted in Figure 6.5(b). This SQL query is based on Definition 3.2. The matching leaf nodes are stored in the TempLV table. The semantic of the attributes of the TempLV tables is depicted in Figure 6.6.

CHAPTER 6. DETECTING THE CHANGES TO UNORDERED XML

Figure 6.5: XANDY-U: The *findMatchingSiblingOrder* algorithm, An Example, and SQL Queries.

Attribute	Description
ID1	The leaf order of a leaf node in the first version
ID2	The leaf order of a leaf node in the second version
SO1	The sibling order of a leaf node in the first version
SO2	The sibling order of a leaf node in the second version
Counter	The number of matching leaf nodes
Total	The total number of leaf nodes
Score	The similarity score
DID1	The document id of the first document
DID2	The document id of the second document
LEVEL	The node's level
MINSO1	The minimum sibling order of an internal node in the first version
MAXSO1	The maximum sibling order of an internal node in the first version
MINSO2	The minimum sibling order of an internal node in the second version
MAXSO2	The maximum sibling order of an internal node in the second version

(a) Attributes of Tables

(b) Description of Attributes

Figure 6.6: XANDY-U: Table Descriptions.

The next step is to refine the detected matching leaf nodes. Suppose we have two sets of leaf nodes, namely, $\mathcal{L}_p \subseteq \mathcal{L}(T_1)$ and $\mathcal{L}_q \subseteq \mathcal{L}(T_2)$, whose parent nodes are nodes p and q , respectively. This step is needed if $\exists \ell_{1x} \in \mathcal{L}_p$ such that $\forall \ell_{2y} \in L (\ell_{1x} \leftrightarrow \ell_{2y})$, where $L \subseteq \mathcal{L}_q$ and $|L| > 1$. For example, consider T_1 and T_2 as depicted in Figure 6.5(c). We notice that $\ell_2 \leftrightarrow \ell_{12}$ and $\ell_2 \leftrightarrow \ell_{13}$. Hence, we need to prune either one of $\ell_2 \leftrightarrow \ell_{12}$ and $\ell_2 \leftrightarrow \ell_{13}$. This is because if we do not prune it, then we can get incorrect result. For example, the number of matching leaf nodes in T_1 and T_2 is equal to 2. If we do not do pruning, then it will be equal to 3. We call the pruned tuples as *overlapping* tuples.

The intuition behind the *refineMatchingLV* algorithm for correcting the TempLV table is

Figure 6.7: XANDY-U: SQL Queries Used in The *refineMatchingLV* algorithm.

similar to the one of the *updateCorrector* algorithm that is depicted in Figure 4.41(a). The *refineMatchingLV* algorithm invokes the SQL query Q1 (Figure 6.7(a)). The SQL query in Figure 6.7(b) is used to prune the *overlapping* tuples in the **TempLV** table. Lines 4–8 are used to retrieve tuples of corresponding leaf nodes that are matched more than once. Then, the *findMatchingSiblingOrder* algorithm starts to determine the matching sibling orders (line 3, Figure 6.5(a)) by invoking the SQL query as depicted in Figure 6.5(d). The intuition is to group the matching leaf nodes according to their node level and sibling orders. Note that the algorithm needs to updates the value of the *Total* attribute as its value is still zero. The SQL query for calculating the value of *Total* attribute is similar to the one depicted in Figure 4.13(g).

Given the **LeafValue** table as depicted in Figure 6.2, the matching leaf nodes detected by the *findMatchingSiblingOrder* algorithm are depicted in Figure 6.9(a) (the **TempLV** table). The intermediate result of SQL query depicted in Figure 6.7(b) is shown in Figure 6.8. The matching sibling orders determined by the *findMatchingSiblingOrder* algorithm are shown in Figure 6.9(b) (the **TempS0** table). The semantic of the attributes of the **TempS0** tables is depicted in Figure 6.6.

6.2.2 Bottom-up Matching

After the matching sibling orders are determined, the *findBestMatchingSubtrees* algorithm starts propagating the matchings in bottom-up fashion (lines 3–8, Figure 4.11). The first step is to get the maximum level *maxLevel* of the internal nodes in T_1 and T_2 by querying the **AncestorInfo** table. Then, the best matching subtrees are determined iteratively starting from level *maxLevel* to level 1 (lines 4–8, Figure 4.11). For each level

Figure 6.8: XANDY-U: Intermediate Result of SQL Query in Figure 6.7(b).

$1 \leq curLevel \leq maxLevel$, the algorithm finds the matching internal nodes at level $curLevel$ (line 5, Figure 4.11). As an internal node in T_1 can be matched to more than one internal node in T_2 and vice versa, the algorithm determines the best matching subtrees by finding the *best matching configurations* (line 6, Figure 4.11). Finally, it prunes the unmatching subtrees (line 7, Figure 4.11). We shall elaborate on each phase further.

Finding Matching Internal Nodes

The SQL query for finding matching internal nodes in XANDY-U is similar to the one in XANDY-O that is depicted in Figure 4.17(b) (without line 4). Line 4 is used to project the *LocalOrder* attribute. We do not need this line as we do not have the *LocalOrder* attribute in our modified SUCXENT schema for XANDY-U.

After executing this SQL query, we need to update the *Total* and *Score* attributes as they are still “0”. The matching internal nodes are stored in the **TempMatching** table. The semantic of the attributes of the **TempMatching** table is depicted in Figure 6.6. Figures 6.10(a)–6.10(e) depict the **TempMatching** table where $1 \leq curLevel \leq 5$. The highlighted tuples in the **TempMatching** table (Figures 6.10(a)–6.10(e)) are deleted tuples as they are the corresponding tuples of unmatching subtrees.

Level	ID1	ID2	SO1	SO2	Level	ID1	ID2	SO1	SO2	Level	SO1	SO2	Counter	Total
3	1	1	1	1	3	14	8	6	4	3	1	1	4	4
3	1	8	1	4	5	16	10	7	5	3	1	4	2	4
5	2	2	1	1	5	17	4	7	2	5	2	2	4	6
5	4	4	2	2	5	17	11	7	5	5	2	5	2	6
5	4	10	2	5	5	18	12	7	5	6	3	3	4	5
5	5	5	2	2	6	19	13	8	6	3	6	1	2	4
6	6	7	3	3	6	20	14	8	6	3	6	4	2	4
6	8	6	3	3	6	21	16	8	6	5	7	5	6	6
3	14	1	6	1	6	21	20	8	8	5	7	2	2	6
										6	8	6	6	7
										6	8	8	2	4

(a) TempLV Table

(b) TempSO Table

Figure 6.9: XANDY-U: The TempLV and TempSO Tables.

Maximizing Similarity Score

The objective of this phase is to find the best matching configuration at level $curLevel$ in order to maximize the similarity score of subtrees at level $(curLevel - 1)$. This phase is similar to the one in XANDY-O. We also use the Hungarian method [PS82] to find the best matching configurations. Recall that the problem of finding best matching configurations is similar to the problem of finding maximum weighted bipartite matching. After finding the best matching configurations, we delete the tuples of corresponding matching subtrees that are not used in the best matching subtrees.

For example, consider T_1 and T_2 as depicted in Figure 1.8. We notice that $t_{25} \simeq t_{117}$ and $t_{25} \simeq t_{126}$. The tuples with “#” in Figure 6.10(b) are the corresponding tuples of $t_{25} \simeq t_{117}$ and $t_{25} \simeq t_{126}$. Then, $\mathfrak{R}(t_{24}, t_{116})$ is maximized if we choose $t_{25} \simeq t_{117}$ instead of $t_{25} \simeq t_{126}$. If we choose $t_{25} \simeq t_{126}$, then $\mathfrak{R}(t_{24}, t_{116}) = 0.118$. If $t_{25} \simeq t_{117}$ is chosen, then $\mathfrak{R}(t_{24}, t_{116}) = 0.706$. Therefore, $t_{25} \simeq t_{117}$ is included in the best matching configuration. Consequently, we need to delete the corresponding tuple of $t_{25} \simeq t_{126}$. The highlighted tuple with “#” in Figure 6.10(c) is the corresponding tuple of $t_{25} \simeq t_{126}$, while the one with “@” in Figure 6.10(c) is also deleted because $t_{29} \simeq t_{130}$ are in $t_{25} \simeq t_{126}$. The tuples with “%” in Figure 6.10(c) are the corresponding tuples of $t_{24} \simeq t_{116}$.

Deleting Unmatching Subtrees

The next step is to delete the matching internal nodes that have similarity scores less than the similarity score threshold θ . Recall that such matching subtrees are considered as

CHAPTER 6. DETECTING THE CHANGES TO UNORDERED XML

Level	MinSO1	MaxSO1	MinSO2	MaxSO2	Counter	Total	Score
5	3	3	3	3	4	5	0.800
5	8	8	6	6	6	7	0.857
5	8	8	8	8	2	4	0.500

(a) Level 5

Level	MinSO1	MaxSO1	MinSO2	MaxSO2	Counter	Total	Score
5	3	3	3	3	4	5	0.800
5	8	8	6	6	6	7	0.857
5	8	8	8	8	2	4	0.500
4	2	3	2	3	8	11	0.727
4	2	3	5	6	2	13	0.154
4	7	8	2	3	2	11	0.182
4	7	8	5	6	12	13	0.923
4	7	8	7	8	2	10	0.200

(b) Level 4

Level	MinSO1	MaxSO1	MinSO2	MaxSO2	Counter	Total	Score
5	3	3	3	3	4	5	0.800
5	8	8	6	6	6	7	0.857
4	2	3	2	3	8	11	0.727
4	2	3	5	6	2	13	0.154
4	7	8	2	3	2	11	0.182
4	7	8	5	6	12	13	0.923
3	2	5	2	3	8	16	0.500
3	2	5	5	8	2	22	0.091
3	7	8	5	8	12	17	0.706
3	7	8	2	3	2	11	0.182
2	1	5	1	3	12	20	0.600
2	1	5	4	8	4	26	0.154
2	6	8	4	8	14	21	0.667
2	6	8	1	3	4	15	0.267

(d) Level 2

Level	MinSO1	MaxSO1	MinSO2	MaxSO2	Counter	Total	Score
5	3	3	3	3	4	5	0.800
5	8	8	6	6	6	7	0.857
4	2	3	2	3	8	11	0.727
4	2	3	5	6	2	13	0.154
4	7	8	2	3	2	11	0.182
4	7	8	5	6	12	13	0.923
3	2	5	2	3	8	16	0.500
3	2	5	5	8	2	22	0.091
3	7	8	5	8	12	17	0.706
3	7	8	2	3	2	11	0.182
2	1	5	1	3	12	20	0.600
2	1	5	4	8	4	26	0.154
2	6	8	4	8	14	21	0.667
2	6	8	1	3	4	15	0.267
1	1	8	1	8	26	41	0.634

(e) Level 1

Figure 6.10: XANDY-U: The TempMatching Tables.

unmatching subtrees. The algorithm continues to find the best matching subtrees at level $(curLevel - 1)$ iteratively until $curLevel = 1$.

After the bottom-up matching process is done, the tuples left in the TempMatching table are the corresponding tuples of the best matching subtrees. As not all attributes in the TempMatching table are useful in Phase 2, we move the needed ones to the MATCHING table by using SQL query in Figure 6.11(a). The MATCHING table is depicted in Figure 6.11(b). The semantic of the attributes of the MATCHING table is depicted in Figure 6.6.

6.3 Finding Best Matching Subtrees Phase in HELIOS-U

In this section, we shall elaborate on how HELIOS-U determines the best matching subtrees.

The *findBestMatchingSubtrees* algorithm in HELIOS-U is similar to the one in HELIOS-O that is depicted in Figure 4.22. The *findBestMatchingSubtrees* algorithm consists of three phases, namely, the *finding matching leaf nodes groups* phase, the *bottom-up matching* phase, and the *collecting best matching internal nodes* phase. We shall discuss each phase in turn. Note that “[param]” in the SQL queries used in the later discussion will be replaced

Figure 6.11: XANDY-U: SQL Query and The MATCHING Table.

Figure 6.12: HELIOS-U: Finding Matching Leaf Nodes Groups.

the parameter *param* defined in the algorithm.

6.3.1 Finding Matching Leaf Nodes Groups Phase

The *findMatchingLeafNodesGroups* algorithm as depicted in Figure 6.12(a) is used to determine the *matching leaf nodes groups*. It works as follows.

The first step in the *findMatchingLeafNodesGroups* algorithm is to find matching leaf nodes (line 1, Figure 6.12(a)) by using SQL query in Figure 6.12(b). Line 9 is used to ensure that matching leaf nodes have the same value. The next step is to refine the detected matching leaf nodes (line 2, Figure 6.12(a)). This is because of the same reasons as discussed in the previous section. The intuition behind the *refineMatchingLV* algorithm in HELIOS-U is similar to the one in XANDY-U.

Having refined the detected matching leaf nodes, the algorithm starts finding the matching leaf nodes groups (line 3, Figure 6.12(a)). It executes the SQL query in Figure 6.12(c). The intuition is to group the matching leaf nodes according to their parent nodes (line 8, Figure 6.12(c)). The result of this SQL query is stored in the M_ℓ table. The schema of the M_ℓ table is the same as the one of the `tempTb1` table as depicted in Figure 4.27(a). The semantics of attributes of the `tempTb1` table are depicted in Figure 4.27(b). For instance, in our example, we have node “*author*” as non-inlined leaf nodes. The *findMatchingLeafNodesGroups* algorithm will result the `M_Author` table as depicted in Figure 6.13(a).

6.3.2 Bottom-up Matching Phase

Next, the algorithm propagates the matchings in bottom-up fashion (lines 5–14, Figure 4.22). First, the algorithm determines the highest level of the non-inlined internal nodes in DTD U (line 5). Then, it starts to find best matching internal nodes in bottom-up fashion. There are two sub steps, that is, finding matching internal nodes (line 11) and determining best matching subtrees (line 12) by finding *best matching configurations*. We shall elaborate on each step in turn.

Finding Matching Internal Nodes

For each non-inlined internal node $i_w \in N_i$, the *findBestMatchingSubtrees* algorithm invokes the *findMatchingInternalNodes* algorithm (line 11, Figure 4.22) after it defines three parameters (lines 9–10, Figure 4.22) that will be the input parameters of the *findMatchingInternalNodes* algorithm. Then, it executes the SQL query as depicted in Figure 4.28(a) after slight modifications. Line 5 is removed. The “0 AS FLAG” in line 7 is also deleted. If node $i_w \in N_i$ has inlined child leaf nodes, then the algorithm also executes the SQL query depicted in Figure 4.28(b) after slight modifications. The modifications are as following. We delete line 5 in Figure 4.28(b). The “0 AS FLAG” in line 7 is also removed. The result of these SQL queries is stored in a temporary matching table M_{i_w} . The schema of the M_{i_w} table is the same as the one of the `tempTb2` table as depicted in Figure 4.27(a). The semantics of attributes of the `tempTb2` table are depicted in Figure 4.27(b).

For example, consider T_1 and T_2 as depicted in Figure 1.8. Figure 6.4 depicts T_1 and T_2 shredded in relational database (partial view only). Suppose we want to find matching “*article*” nodes. The values of parameters “[tempTb]”, “[tbName]”, and “[tempMChild]” are “ $M_{article}$ ”, “*article*”, and “ M_{author} ”, respectively. The value of “[moreConditions]”

CHAPTER 6. DETECTING THE CHANGES TO UNORDERED XML

DID1	DID2	PID1	PID2	Counter	Total	Score
1	2	1	1	4	5	0.800
1	2	3	2	6	7	0.857
1	2	3	3	2	4	0.500

(a) M_Author Table

DID1	DID2	ID1	ID2	PID1	PID2	Counter	Total	Score
1	2	1	1	1	1	0	0	0
1	2	3	2	2	2	0	0	0
1	2	3	3	2	2	0	0	0

(b) M_Article Table (1)

DID1	DID2	ID1	ID2	PID1	PID2	Counter	Total	Score
1	2	1	2	1	2	0	0	0
1	2	3	1	2	1	0	0	0

(c) M_Article Table (2)

DID1	DID2	ID1	ID2	PID1	PID2	Counter	Total	Score
1	2	1	1	1	1	8	11	0.727
1	2	3	2	2	2	12	13	0.923
1	2	3	3	2	2	2	10	0.200
1	2	1	2	1	2	2	13	0.154
1	2	3	1	2	1	2	11	0.182

(d) M_Article Table (3)

DID1	DID2	ID1	ID2	PID1	PID2	Counter	Total	Score
1	2	1	1	1	1	12	20	0.600
1	2	1	2	1	1	4	26	0.154
1	2	2	2	1	1	14	21	0.667
1	2	2	1	1	1	4	15	0.267

(e) M_Issue

DID1	DID2	ID1	ID2	PID1	PID2	Score	Name
1	2	1	1	null	null	0.634	SigmodRecord
1	2	1	1	1	1	0.600	issue
1	2	2	2	1	1	0.667	issue
1	2	null	null	1	1	0.500	articles
1	2	null	null	2	2	0.706	articles
1	2	1	1	1	1	0.727	article
1	2	3	2	2	2	0.923	article
1	2	null	null	1	1	0.800	authors
1	2	null	null	3	2	0.857	authors

(f) MATCHING

Figure 6.13: HELIOS-U: The M_Author, M_Article, M_Issue, and MATCHING Tables.

is “ $A1.title = A2.title$ AND $A1.initPage = A2.initPage$ AND $A1.endPage = A2.endPage$ ”. Then, the results of SQL queries depicted in Figures 4.28(a) and 4.28(b) (after the above modifications) are shown in Figures 6.13(b) and 6.13(c), respectively. Thus, the result of this phase is the M_Article table as depicted in Figure 6.13(d) (with the values of the *Counter*, *Total*, and *Score* attributes are equal to “0”).

Finding Best Matching Internal Nodes

The next step is to determine the best matching configurations that help us to determine the best matching subtrees. This phase is similar to the one in HELIOS-O. The difference is as follows. In HELIOS-O, if the matching internal nodes have parent nodes that are not used in the best matching configurations, then the algorithm sets the values of their *Flag* attribute to “1”. In HELIOS-U, we delete such matching internal nodes. Observe that we also need to update the values of the *Counter*, *Total*, and *Score* attributes accordingly as initially their values are equal to “0”. Intuitively, the value of *Counter* attribute is equal to $(p + \sum q_z)$, where p is the number of matching inlined leaf nodes, and q_z is the number of matching non-inlined leaf nodes retrieved from the temporary matching table. The value of *Total* attribute is equal to $(2k + \sum l_z)$, where k is the total number of inlined leaf nodes, and l_z the total number of non-inlined leaf nodes retrieved from the temporary matching

Input: *did1* and *did2*
Output: XDelta tables

```
1 detectInsertedInternalNodes (did1, did2);  
2 detectDeletedInternalNodes (did1, did2);  
3 detectInsertedLeafNodes (did1, did2);  
4 detectDeletedLeafNodes (did1, did2);  
5 detectUpdate (did1, did2);  
6 updateCorrector (did1, did2);
```

Figure 6.14: XANDY-U: The *changeDetector* Algorithm.

table. The **M_Article** and **M_Issue** tables after this phase are depicted in Figures 6.13(d) and 6.13(e).

6.3.3 Collecting Best Matching Internal Nodes Phase

Finally, the *findBestMatchingSubtrees* algorithm merges/collects the best matching internal nodes from different relations and determines the best matching inlined internal nodes. It invokes the *retrieveMatching* algorithm (line 24). The *retrieveMatching* algorithm in HELIOS-U is similar to the one in HELIOS-O. The differences are only in the executed SQL queries. The SQL queries executed by the *retrieveMatching* algorithm in HELIOS-U are similar to the ones in HELIOS-O (Figure 4.32) after slight modifications. The modification for the SQL query in Figure 4.32(a) is as follows. We delete the “L01, L02,” in line 3. The SQL query in Figure 4.32(b) is modified as follows. We remove “I.L01,” and “I.L02,” in lines 2 and 3, respectively. Line 11 is removed. The modification for the SQL query in Figure 4.32(c) is as following. Line 5 is deleted.

The best matching subtrees are stored in the **MATCHING** tables whose schema is depicted in Figure 4.27(a). The semantics of attributes of the **MATCHING** table are depicted in Figure 4.27(b). For example, consider T_1 and T_2 as depicted in Figure 1.8. Figure 6.4 depicts T_1 and T_2 shredded in relational database (partial view only). The best matching subtrees detected by the *findBestMatchingSubtrees* are depicted in Figure 6.13(f).

6.4 Detecting the Changes Phase in XANDY-U

In this section, we shall discuss how the changes are detected by XANDY-U. Note that the types of changes that can be detected by XANDY-U are discussed in Chapter 1. The *changeDetector* algorithm for detecting the changes is shown in Figure 6.14. We shall elaborate on each step in turn.

Figure 6.15: XANDY-U: The Delta Tables.

6.4.1 Insertion of Internal Nodes

The inserted internal nodes can be detected by using the similar SQL query as shown in Figure 4.34(a) after slight modification. We delete line 3 in Figure 4.34(a). The INS_INT table stores the detected inserted internal nodes. The semantics of attributes of the INS_INT table are depicted in Figures 4.35(a) and 4.35(b). Note that the schema of the INS_INT table in Figure 4.35(a) is without the L0 attribute. For example, given T_1 and T_2 shredded in relational database (Figure 6.2), the inserted internal nodes are stored in the INS_INT table as depicted in Figure 6.15(a).

6.4.2 Deletion of Internal Nodes

Similarly, the deleted internal nodes can be detected by using SQL query depicted in Figure 4.34(a) after slight modifications. The modifications are as following. We delete line 3 in Figure 4.34(a). We replace “*did2*” in line 6 with “*did1*”. The “MINSO2” (line 11) and “MAXSO2” (line 12) are replaced by “MINSO1” and “MAXSO1”, respectively. The deleted internal nodes are stored in the DEL_INT table. The semantics of its attributes are depicted in Figures 4.35(a) and 4.35(b). Note that the schema of the DEL_INT table in Figure 4.35(a) is also without the L0 attribute. For example, given T_1 and T_2 shredded in relational database (Figure 6.2), the inserted internal nodes are stored in the DEL_INT table as depicted in Figure 6.15(b).

6.4.3 Insertion of Leaf Nodes

To detect the inserted leaf nodes, XANDY-U executes two SQL queries as depicted in Figures 4.34(b) and 4.34(c) after slight modifications. Recall that the inserted leaf nodes can be in the inserted subtrees or in the matching subtrees. Figure 4.34(b) is used to detect the inserted leaf nodes whose parents are inserted internal nodes. The inserted leaf nodes that are in the best matching subtrees are detected by using SQL query that is depicted in Figure 4.34(c). The modifications of these SQL queries are as follows. We remove “L.LOCALORDER,” in line 3 in Figure 4.34(b) and “L.LOCALORDER” in line 3 in Figure 4.34(c). The inserted leaf nodes are stored in the `INS_LEAF` table. The semantics of its attributes are depicted in Figures 4.35(a) and 4.35(b). Note that the schema of the `INS_LEAF` table in Figure 4.35(a) is without the `L0` attribute. Given T_1 and T_2 shredded in relational database (Figure 6.2), the inserted leaf nodes are stored in the `INS_LEAF` table as depicted in Figure 6.15(c). Note that the highlighted tuples in Figure 6.15(c) are the updated leaf nodes detected as inserted leaf nodes. An updated leaf node can be decomposed a pair of a deleted leaf node and an inserted leaf node.

6.4.4 Deletion of Leaf Nodes

Similarly, XANDY-U also executes two SQL queries as depicted in Figures 4.34(b) and 4.34(c) after slight modifications to detect the deleted leaf nodes. The deleted leaf nodes can be in the deleted subtrees or in the matching subtrees. The modifications of SQL queries in Figure 4.34(b) are as follows. We remove “L.LOCALORDER” in line 3. We replace “*did2*” (line 6) and “`INS_INT`” (line 4) with “*did1*” and “`DEL_INT`”, respectively. The SQL queries in Figure 4.34(c) are modified as follows. We remove “L.LOCALORDER” in line 3. In lines 10 and 22, the “`MINS02`” and “`MINS02`” are replaced by “`MINS01`” and “`MINS01`”, respectively. The “`MINS01`” and “`MINS01`” in line 19 are replaced by “`MINS02`” and “`MINS02`”, respectively. We replace “*did1*” (line 17) and “*did2*” (lines 9 and 21) with “*did2*” and “*did1*”, respectively.

The `DEL_LEAF` table stores the deleted leaf nodes. The semantics of its attributes are depicted in Figures 4.35(a) and 4.35(b). Note that the schema of the `DEL_LEAF` table in Figure 4.35(a) is without the `L0` attribute. Given T_1 and T_2 shredded in relational database (Figure 6.2), the inserted leaf nodes are stored in the `DEL_LEAF` table as depicted in Figure 6.15(d). Note that the highlighted tuples in Figure 6.15(d) are the updated leaf nodes detected as deleted leaf nodes. An updated leaf node can be decomposed a pair of a deleted leaf node and an inserted leaf node.

6.4.5 Content Updates of Leaf Nodes

To detect the updated leaf nodes, XANDY-U executes an SQL query as depicted in Figure 4.38(b) after slight modifications. We remove “D.L0 AS L01, I.L0 AS L02,” in line 3 and “D.L0 != I.L0” in line 8. Note that we do not distinguish the updated leaf nodes in unordered XML as the position among siblings is not important. We observed that the modified SQL query may also return incorrect result as discussed in Chapter 4. Therefore, we need to correct the result by invoking the *updateCorrector* algorithm as depicted in Figure 4.41(a). The intuition of the *updateCorrector* algorithm is similar to the one discussed in Chapter 4. The *updateCorrector* algorithm in XANDY-U also executes the SQL queries depicted in Figures 4.41(b) and 4.41(c). The details on the *updateCorrector* algorithm can be found in Chapter 4. The updated leaf nodes are stored in the UPD_LEAF table. The semantics of its attributes are depicted in Figures 4.35(a) and 4.35(b). Note that the UPD_LEAF table is without the L01 and L02 attributes. In our example, we notice that there are two updated leaf nodes that are stored in the UPD_LEAF table as depicted in Figure 6.15(e). After we detect the updated leaf nodes, we have to delete the corresponding tuples of the updated leaf nodes in the DEL_LEAF and INS_LEAF tables. The highlighted tuples in Figures 6.15(c) and 6.15(d) are deleted.

6.5 Detecting the Changes Phase in HELIOS-U

In this section, we shall present Phase 2 in HELIOS-U that is used for detecting the changes after the best matching subtrees are determined. HELIOS-U is able to detect different types of changes in unordered XML documents as presented in Chapter 1. The *changeDetector* algorithm for detecting the changes is shown in Figure 6.16. We shall elaborate on each step in turn.

6.5.1 Insertion of Internal Nodes

In this section, we elaborate on how the inserted non-inlined and inlined internal nodes are detected by HELIOS-U.

Non-inlined Internal Nodes

For each non-inlined internal node $i_x \in N_i(U)$, the *changeDetector* algorithm defines two parameters (line 6, Figure 4.49), and invokes the *detectInsertedInternalNodes* function (line 7,

CHAPTER 6. DETECTING THE CHANGES TO UNORDERED XML

```

Input
  U : DTD of the XML documents
  the Matching table
  Two versions of an XML document stored in RDBMS
Output
  the delta tables

1   $\mathbb{I}_l(U) \leftarrow$  all the inlined leaf nodes in U
2   $\mathbb{I}_i(U) \leftarrow$  all the inlined internal nodes in U
3   $\mathbb{N}_l(U) \leftarrow$  all the non-inlined leaf nodes in U
4   $\mathbb{N}_i(U) \leftarrow$  all the non-inlined internal nodes in U
5  for all  $i_x \in \mathbb{N}_i(U)$  do
6    nodeName  $\leftarrow$  name( $i_x$ ); tbName  $\leftarrow$   $r_u$ ;
7    detectInsertedInternalNodes(nodeName, tbName);
8    detectDeletedInternalNodes(nodeName, tbName);
9  end for
10 for all  $i_y \in \mathbb{I}_i(U)$  do
11   parentNode  $\leftarrow$  parent( $i_y$ ); attrName  $\leftarrow$  attribute( $i_y$ );
12   nodeName  $\leftarrow$  name( $i_y$ ); parentTbName  $\leftarrow$   $r_{\text{parentNode}}$ ;
13   parentNodeName  $\leftarrow$  name(parentNode);
14   detectInsertedInternalNodes(nodeName, attrName, parentTbName, parentNodeName);
15   detectDeletedInternalNodes(nodeName, attrName, parentTbName, parentNodeName);
16 end for
17 for all  $\ell_y \in \mathbb{I}_l(U)$  do
18   parentNode  $\leftarrow$  parent( $\ell_y$ ); nodeName  $\leftarrow$  name( $\ell_y$ );
19   attrName  $\leftarrow$  attribute( $\ell_y$ );
20   parentTbName  $\leftarrow$   $r_{\text{parentNode}}$ ;
20   parentNodeName  $\leftarrow$  name(parentNode);
21   detectInsertedLeafNodes(nodeName, attrName, parentTbName, parentNodeName);
22   detectDeletedLeafNodes(nodeName, attrName, parentTbName, parentNodeName);
23   detectUpdatedLeafNodes(nodeName, attrName, parentTbName, parentNodeName);
24 end for
25 for all  $\ell_x \in \mathbb{N}_l(U)$  do
26   nodeName  $\leftarrow$  name( $\ell_x$ ); parentNode  $\leftarrow$  parent( $\ell_x$ );
27   parentNodeName  $\leftarrow$  name(parentNode); tbName  $\leftarrow$   $r_u$ ;
28   detectInsertedLeafNodes(nodeName, tbName, parentNodeName);
29   detectDeletedLeafNodes(nodeName, tbName, parentNodeName);
30   detectUpdatedLeafNodes(nodeName, parentNodeName);
31   refineUpdatedLeafNodes( $\ell_x$ );
32 end for

```

Figure 6.16: HELIOS-U: The *changeDetector* Algorithm.

Figure 4.49) for detecting inserted non-inlined internal nodes. Then, the *detectInsertedInternalNodes* function executes the SQL query depicted in Figure 4.50(a) after slight modification. We remove “LOCALORDER,” in line 4. The inserted non-inlined internal nodes are stored in the INS_INT table. The semantics of its attributes are depicted in Figures 4.52(a) and 4.52(b). Note that the schema of the INS_INT table in Figure 4.52(a) is without the LO attribute.

Inlined Internal Nodes

For each inlined internal node $i_y \in \mathbb{I}_i(U)$, the *changeDetector* algorithm defines four parameters (lines 11–13, Figure 4.49). Next, the algorithm invokes the *detectInsertedInternalNodes* function (line 14, Figure 4.49) for detecting inserted inlined internal nodes. Then, two SQL

DID1	DID2	ID	PID	Name
1	2	3	2	article
1	2	null	3	authors

(a) INS_INT Table

DID1	DID2	ID	PID	Name
1	2	2	1	article
1	2	null	2	authors

(b) DEL_INT Table

DID1	DID2	ID	PID	Name	Value
1	2	null	1	title	Web Usage Mining
1	2	null	2	number	2
1	2	5	2	author	Peter
1	2	null	3	title	Active Database
1	2	null	3	initPage	13
1	2	null	3	endPage	30
1	2	7	3	author	Greg

(c) INS_LEAF Table

DID1	DID2	ID	PID	Name	Value
1	2	null	1	title	Web Mining
1	2	2	1	author	Marcella
1	2	null	3	title	XML Mining
1	2	null	3	initPage	10
1	2	null	3	endPage	20
1	2	4	2	author	Steph
1	2	5	2	author	Tanaka
1	2	null	3	number	3

(d) DEL_LEAF Table

DID1	DID2	ID1	ID2	PID1	PID2	Name	Value1	Value2
1	2	null	null	1	1	title	Web Mining	Web Usage Mining
1	2	null	null	3	2	number	3	2

(e) UPD_LEAF Table

Figure 6.17: HELIOS-U: Delta Tables.

queries are executed by the *changeDetector* algorithm. They are depicted in Figures 4.50(b) and 4.50(c). Note that these SQL queries need to be modified for detecting the inserted internal nodes in unordered XML documents. The modifications are as follows. We remove “NULL AS L0,” in line 4 in Figures 4.50(b) and 4.50(c). The inserted inlined internal nodes are also stored in the INS_INT table.

For example, we have two versions of an XML document that are shredded in relational database as depicted in Figure 6.2. The first tuple in the INS_INT table depicted in Figure 6.17(a) is a non-inlined inserted internal node. This node is detected by the SQL query in Figure 4.50(a). The second tuple is an inlined inserted internal node that is detected by using SQL query in Figure 4.50(c) as it is in an inserted subtree.

6.5.2 Deletion of Internal Nodes

In this section, we elaborate on how the deleted non-inlined and inlined internal nodes are detected by HELIOS-U.

Non-inlined Internal Nodes

Deleted non-inlined internal nodes can be detected by using SQL query that is depicted in Figure 4.50(a) after slight modifications. We remove “LOCALORDER,” in line 4. The “doc_id2” in line 8 is replaced by “doc_id1”. We replace “ID2” in lines 10 and 15 with

“ID1”. The “INS_INT” in line 1 is replaced by “DEL_INT”. Observe that this modified SQL query is executed by the *detectDeletedInternalNodes* function (line 8, Figure 4.49) after setting two parameters that are in line 6 in Figure 4.49. The deleted non-inlined internal nodes are stored in the “DEL_INT” table. The semantics of its attributes are depicted in Figures 4.52(a) and 4.52(b). Note that the schema of the DEL_INT table in Figure 4.52(a) is without the L0 attribute.

Inlined Internal Nodes

To detect the deleted inlined internal nodes, the *changeDetector* algorithm invokes the *detectDeletedInternalNodes* function (line 14, Figure 4.49) after defining four parameters (lines 11–13, Figure 4.49). The *detectDeletedInternalNodes* function executes two SQL queries as depicted in Figures 4.50(b) and 4.50(c) after slight modifications.

The SQL query in Figure 4.50(b) is modified as following. The “NULL AS L0,” in line 4 is removed. We replace “INS_INT” in line 1 with “DEL_INT”. The “A2” (line 4), “FALSE” (line 8), and “TRUE” (line 8) are replaced by “A1”, “TRUE”, and “FALSE”, respectively. We modify the SQL query in Figure 4.50(c) as follows. The “INS_INT” in lines 1 and 5 is replaced by “DEL_INT”. The “doc_id2” in line 10 is replaced by “doc_id1”. The deleted inlined internal nodes are also stored in the “DEL_INT” table.

For example, consider two versions of an XML document that are shredded in relational database as depicted in Figure 6.2. The DEL_INT table that stores the results of the SQL queries depicted in Figures 4.50(a)–4.50(c) is shown in Figure 6.17(b). The first and second tuples are a non-inlined inserted internal node and an inlined inserted internal node, respectively.

6.5.3 Insertion of Leaf Nodes

In this section, we discuss how HELIOS-U detects the inserted non-inlined and inlined leaf nodes.

Inlined Leaf Nodes

For each inlined leaf node $\ell_y \in \mathbb{I}_\ell(U)$, the *changeDetector* algorithm defines four parameters (lines 18–20, Figure 4.49). Next, the algorithm invokes the *detectInsertedLeafNodes* function (line 21, Figure 4.49) for detecting inserted inlined leaf nodes.

The *detectInsertedLeafNodes* function executes two SQL queries as depicted in Figures 4.53(a) and 4.53(b) after slight modifications. The SQL query in Figure 4.53(a) is

used to detect the inlined inserted leaf nodes whose parent nodes are inserted internal nodes. The inlined inserted leaf nodes that are in the best matching subtrees are detected by using SQL in Figure 4.53(b). The modification of SQL query in Figure 4.53(a) is as follows. We remove “NULL AS L0,” in line 3. We modify SQL query in Figure 4.53(b) as following. The “NULL AS L0,” in line 3 is deleted. The inserted leaf nodes are stored in the `INS_LEAF` table. The semantics of its attributes are depicted in Figures 4.52(a) and 4.52(b). The schema of the `INS_LEAF` table in Figure 4.52(a) is without the `L0` attribute. Note that the updated leaf nodes are also detected in this phase.

Non-inlined Leaf Nodes

For each non-inlined leaf node $\ell_x \in \mathcal{N}_\ell(U)$, the *changeDetector* algorithm defines three parameters (lines 26–27, Figure 4.49). Next, the *changeDetector* algorithm invokes the *detectInsertedLeafNodes* (line 28, Figure 4.49) for detecting the inserted non-inlined leaf nodes.

The *detectInsertedLeafNodes* function executes two SQL queries as depicted in Figures 4.53(c) and 4.53(d) after slight modifications. We remove “A.LOCALORDER,” in line 2 (Figure 4.53(c)) and “A.LOCALORDER AS L0,” in line 3 (Figure 4.53(d)). Note that the SQL queries in Figures 4.53(c) and 4.53(d) are used to detect the non-inlined inserted leaf nodes that are in the inserted subtrees and in the best matching subtrees, respectively. The inserted leaf nodes are stored in the `INS_LEAF` table and the updated leaf nodes are also detected in this phase.

Consider two versions of an XML document that are shredded in relational database as depicted in Figure 6.2. Figure 6.17(c) depicts the inserted leaf nodes detected by the SQL queries depicted in Figures 4.53(a)–4.53(d). The highlighted tuples are the corresponding tuples of updated leaf nodes detected as inserted leaf nodes.

6.5.4 Deletion of Leaf Nodes

In this section, we present the details on how HELIOS-U detects the deleted non-inlined and inlined leaf nodes.

Inlined Leaf Nodes

The inlined deleted leaf nodes are detected by two SQL queries as depicted in Figures 4.53(a) and 4.53(b) after slight modifications. The SQL query in Figure 4.53(a) is modified as follows. We remove “NULL AS L0,” in line 3 and replace “INS_LEAF” in line 1 with “DEL_LEAF”.

The “IS NULL” in line 8 and “IS NOT NULL” in line 9 are replaced by “IS NOT NULL” and “IS NULL”, respectively. The “I1” replaces “I2” in lines 3 and 4. The modifications of the SQL query in Figure 4.53(b) are as following. We replace “INS_LEAF” in line 1 with “DEL_LEAF” and “INS_INT” in line 5 with “DEL_INT”. The “NULL AS LO,” in line 3 is removed. The “*doc_id2*” in line 10 is replaced by “*doc_id1*”.

The DEL_LEAF table stores the detected deleted inlined leaf nodes. The semantics of its attributes are depicted in Figures 4.52(a) and 4.52(b). The schema of the DEL_LEAF table in Figure 4.52(a) is without the LO attribute. Note that the updated leaf nodes are also detected in this phase.

Non-inlined Leaf Nodes

The SQL queries in Figures 4.53(c) and 4.53(d) are used to detect the non-inlined deleted leaf nodes after slight modifications. We modify SQL query in Figure 4.53(c) as following. We replace “INS_LEAF” in line 1 with “DEL_LEAF”. The “A.LOCALORDER,” in line 2 is removed. We replace “*doc_id2*” in lines 7 and 16 with “*doc_id1*”. The “*doc_id1*” in line 13 is replaced by “*doc_id2*”. The “PID1” replaces “PID2” in lines 7 and 16. We replace “PID1” in line 13 with “PID2”. The modifications of SQL query in Figure 4.53(d) are as following. We replace “INS_LEAF” in line 1 with “DEL_LEAF” and “INS_INT” in line 5 with “DEL_INT”. The “A.LOCALORDER AS LO,” in line 3 is removed. The “*doc_id2*” in line 6 is replaced by “*doc_id1*”. The detected deleted inlined leaf nodes are also stored in the DEL_LEAF table. Note that the updated inlined leaf nodes are also detected in this phase.

Consider two versions of an XML document that are shredded in relational database as depicted in Figure 6.2. Figure 6.17(d) depicts the deleted leaf nodes. The highlighted tuples are the corresponding tuples of updated leaf nodes detected as deleted leaf nodes.

6.5.5 Content Updates of Leaf Nodes

The updated leaf nodes are detected after the inserted and deleted leaf nodes are detected. In this section, we shall discuss how HELIOS-U detects the updated non-inlined and inlined leaf nodes.

Inlined Leaf Nodes

To detect the updated inlined leaf nodes, the *detectUpdateLeafNodes* function executes the SQL query as shown in Figure 4.54(a) after we remove line 5. Observe that the updated nodes can be detected from the DEL_LEAF and INS_LEAF tables as they can be decomposed

as pairs of deleted and inserted leaf nodes. The updated inlined leaf nodes are stored in the `UPD_LEAF` table. The semantics of its attributes are depicted in Figures 4.52(a) and 4.52(b). The schema of the `UPD_LEAF` table in Figure 4.52(a) is without the `L01` and `L02` attributes. Note that after detecting the updated leaf nodes, we need to delete the corresponding tuples of the updated leaf nodes in the `DEL_LEAF` and `INS_LEAF` tables.

Non-inlined Leaf Nodes

The non-inlined updated leaf nodes in unordered XML documents are not classified into absolute and relative updated leaf nodes. This is because the position among siblings of a node in unordered XML documents is not important. The non-inlined updated leaf nodes in unordered XML documents can be detected by using SQL query depicted in Figure 4.56 after slight modifications. We remove line 5 and the “AND I.LO = D.LO” in line 17 is also removed. We noticed that the result of this SQL query may be *incorrect* in some conditions as discussed in Chapter 4. Hence, we use the *refineUpdatedLeafNodes* algorithm to correct the result of this SQL query. The intuition behind the *refineUpdatedLeafNodes* algorithm in HELIOS-U is similar to the one in HELIOS-O. The `UPD_LEAF` table is also used to store the updated non-inlined leaf nodes. Similarly, we also need to delete the corresponding tuples of the updated leaf nodes in the `DEL_LEAF` and `INS_LEAF` tables.

Consider two versions of an XML document that are shredded in relational database as depicted in Figure 6.2. Figure 6.17(e) depicts the updated leaf nodes. The highlighted tuples in the `DEL_LEAF` and `INS_LEAF` tables are deleted.

6.6 Summary

In this chapter, we have described two approaches for detecting the changes to *unordered* XML documents, namely, XANDY-U and HELIOS-U. We have presented the extensions of SUCXENT schema and *Shared Inlining* schema that are the underlying relational schemas of XANDY-U and HELIOS-U, respectively. We have presented the algorithms as well as the SQL queries used in XANDY-U and HELIOS-U to determine the best matching subtrees (Phase 1). We have elaborated on how XANDY-U detects the changes (Phase 2). Finally, we have discussed the algorithms and the SQL queries for detecting the changes (Phase 2) in HELIOS-U. In Chapter 7, we shall study the performances of XANDY-U and HELIOS-U compared to the state-of-the-arts approach. We shall also analyze the quality of XDeltas detected by XANDY-U and HELIOS-U.

Chapter 7

Performance Study of Xandy–U and Helios–U

In the previous chapter, we have elaborated on our approaches for detecting the changes to *unordered* XML documents, namely, XANDY–U [LBM05] and HELIOS–U [LB05a]. In this chapter, we shall present their performance study.

This chapter is organized as follows. In Section 7.1, we shall present the experimental settings and data sets used in the experiments. Section 7.2 shall discuss the performance of XANDY–U and HELIOS–U for different number of nodes. We shall elaborate on the execution time proportions for Phase 1 and Phase 2 of our approaches in Section 7.3. In Section 7.4, we shall discuss the effects of file size to the execution time of XANDY–U and HELIOS–U. Furthermore, we shall present the effects of percentage of changes to the performance time of XANDY–U and HELIOS–U in Section 7.5. Section 7.6 shall elaborate on the execution time proportions of each sub phases in Phase 2 of our approaches. In Section 7.7, we shall present the experiments to see the effects of percentage of changes to the performance of XANDY–U and HELIOS–U when there is only a type of changes. We shall also discuss the effects of depth of XML documents to the performance of XANDY–U and HELIOS–U in Section 7.8. In Section 7.9, we shall show the performance as well as the result quality of XANDY–U and HELIOS–U when we vary the value of similarity score threshold θ . The study on the result quality of XANDY–U and HELIOS–U compared to X-Diff for different percentage of changes shall be presented in Section 7.10. Finally, we shall summarize our discussion in Section 7.11.

7.1 Experimental Settings and Data Sets

Our approaches, namely, XANDY-U and HELIOS-U are implemented in Java. We used a commercial relational database system¹ (refers to $\mathcal{RDB}_{\mathcal{B}}$ in Chapter 5) for storing XML documents. We also used the same machine as discussed in Chapter 5. Two synthetic data sets based on the SIGMOD Record DTD [ACM] (SIGMOD Data sets) and Oxford English Dictionary [Dic] (TCSD Data sets) [YÖK04] are also used (Figures 5.1(a) and 5.1(b)). We compared the performance of our approaches to the Java version of X-Diff [WDyC03] (downloaded from <http://www.cs.wisc.edu/~yuanwang/xdiff.html>).

7.2 Numbers of Nodes

In this section, we compare the performances of our approaches for different numbers of nodes in XML documents. We used two data sets: SIGMOD data sets (Figure 5.1(a)) and TCSD data sets (Figure 5.1(b)). We set the similarity score threshold θ to “0.00”. Note that when $\theta = 0.0$, we get the upper-bound of the execution times of our approaches.

7.2.1 Data-centric Data Set

Figures 7.1(a) and 7.1(d) depict the performance of Phase 1 in our approaches when the percentages of changes are set to 3% and 9%, respectively. We notice that HELIOS-U is up to 62.02 times faster than XANDY-U (on average, 16.37 times faster). Similar to XANDY-O, XANDY-U did not return results after 100,000 seconds for data set larger than “SIGMOD-12”. Hence, we stopped the process. This is why we do not plot the results of XANDY-U. Figures 7.1(b) and 7.1(e) depict the performance of Phase 2 in our approaches when the percentages of changes are set to 3% and 9%, respectively. We observe that for the first three data sets XANDY-U is slightly faster than HELIOS-U; however, the difference of execution time between HELIOS-U and XANDY-U is not significant. For the larger data sets, HELIOS-U is up to 197 times faster than XANDY-U. On average, HELIOS-U is 42.74 times faster than XANDY-U.

Figures 7.1(c) and 7.1(f) depict the overall performance of our approaches when the percentages of changes are set to 3% and 9%, respectively. First, we compare HELIOS-U and XANDY-U to X-Diff. X-Diff is up to 3.67 times faster than HELIOS-U for the first two data sets. Thereafter, HELIOS-U is up to 100.70 times faster than X-Diff. Compared to

¹Due to the license agreement, we cannot mention their name.

CHAPTER 7. PERFORMANCE STUDY OF XANDY-U AND HELIOS-U

Figure 7.1: DCSD: Execution Time vs Number of Nodes (Log scale).

XANDY-U; X-Diff is up to 7.73 times faster than XANDY-U for the first four data sets (3% changes). When the percentage of changes is set to 9%, X-Diff is up to 4.34 times faster than XANDY-U for the first three data sets. For all larger data sets, XANDY-U is faster than X-Diff (up to 17 times). We observe that our relational-based approaches are more scalable than X-Diff. This is because of the similar reason as discussed in Chapter 5. Compared to XANDY-U; HELIOS-U is up to 63 times faster.

In the next set of experiments, we increase the main memory to 1 GB. We set the percentage of changes to 3% and the similarity score threshold θ is set to “0.00”. Figure 7.2(a) depicts the performance of Phase 1 in XANDY-U and HELIOS-U. We notice that HELIOS-U is 2.85–63 times faster than XANDY-U. Figure 7.2(b) depicts the performance of Phase 2 in XANDY-U and HELIOS-U. For the first four data sets, XANDY-U is faster than HELIOS-U;

CHAPTER 7. PERFORMANCE STUDY OF XANDY-U AND HELIOS-U

Figure 7.2: DCSD: Execution Time vs Number of Nodes (Log scale, 1GB RAM).

however, the differences are not significant (less than 440 msec). For the larger data sets, HELIOS-U is up to 156 times faster than XANDY-U. Figure 7.2(c) depicts the overall performance of all approaches. HELIOS-U is 1.6–60 times faster than XANDY-U. We observe that X-Diff is now able to detect the changes up to SIGMOD-07. This shows that X-Diff depends on the size of main memory. Note that X-Diff is still much less scalable than our approaches. For the first three data sets, X-Diff is 2.1–6.4 times than XANDY-U. We notice that, for the last three data sets, XANDY-U is up to 65 times faster than X-Diff. Compared to HELIOS-U; X-Diff is 3–3.5 times faster for the first two data sets. Afterwards, HELIOS-U is up to 563 times faster than X-Diff. This set of experiments concludes that increasing the size of main memory is not *a good solution* to have a more scalable memory-based change detection system. This is because there is still a limit of space in main memory to store both XML trees. Therefore, using relational database systems for detecting the changes to XML document is a better and more scalable solution.

Now, we investigate the performance gain of the approaches when we increase the size of the main memory. The ratio is defined as $\frac{a}{b}$, where a is the performance of an approach when the size of memory is 512MB and b is the performance of an approach when the size of memory is 1GB. Recall that the execution time of X-Diff is improved (up to 1.21 times) when the size of memory becomes 1GB (see page 125). Figure 7.3 depicts the performance gain of

Figure 7.3: Performance Gain.

XANDY-U and HELIOS-U. We observe that the performances of XANDY-U and HELIOS-U are affected. For HELIOS-U, it is up to 1.23 times faster (on average, 1.04 times) when we increase the size of the main memory. XANDY-U is up to 1.59 times faster (on average, 1.2 times). These results show that the size of main memory will affect the performance of all approaches.

7.2.2 Text-centric Data Set

Figure 7.4(a) depicts the performance of Phase 1 in our approaches when the percentage of changes is set to 3%. We notice that XANDY-U is up to 1.25 times faster than HELIOS-U for the first two data sets. This is because of the same reason as presented in Chapter 5. As the number of nodes increases, HELIOS-U becomes up to 7.83 times faster than XANDY-U. This is because of the same reasons as presented in Chapter 5. Figure 7.4(b) depicts the performance of Phase 2 in our approaches when the percentage of changes is set to 3%. We observe that HELIOS-U is up to 8.87 times faster than XANDY-U. On average, HELIOS-U is 4.31 times faster than XANDY-U.

Figure 7.4(c) depicts the overall performance of our approaches when the percentage of changes is set to 3%. We observe that XANDY-U is slightly faster than HELIOS-U for the first data set; however, the difference is not significant. For larger data sets, HELIOS-U is up to 6.96 times faster than XANDY-U (on average, 4.23 times faster). Let us now compare our approaches to X-Diff. We observe that X-Diff is up to 2.15 times and 2.10 times faster than HELIOS-U and XANDY-U, respectively, for the first two data sets. This is due to the I/O cost needed by the relational-based approaches. For larger data sets, HELIOS-U and XANDY-U are up to 8.62 times and 3.06 times faster than X-Diff, respectively. Observe that X-Diff becomes less scalable due to the same reason as discussed in Chapter 5.

CHAPTER 7. PERFORMANCE STUDY OF XANDY-U AND HELIOS-U

Figure 7.4: TCS D: Execution Time vs Number of Nodes (Log scale).

7.3 Phase 1 vs Phase 2

In this section, we compare the performance of Phase 1 (“Finding the best matching subtrees” phase) to the one of Phase 2 (“Detecting the Changes” phase). We use the first ten SIGMOD data sets.

Figures 7.5(a) and 7.5(b) depict the comparisons between Phase 1 and Phase 2 in HELIOS-U and XANDY-U, respectively. We observe that the performance of Phase 1 in HELIOS-U takes up to 97.10% of the overall performance (“SIGMOD-10”, Figure 7.5(a)). On average, Phase 1 contributes 71.50% of the overall performance (with $\sigma = 0.162$). Now, let us analyze the comparison between Phase 1 and Phase 2 in XANDY-U. Up to 95.20% of the overall performance of XANDY-U is contributed by the execution time of Phase 1 (“SIGMOD-10”). Phase 1 takes, on average, 84.00% of the overall performance (with $\sigma = 0.076$).

Furthermore, we find out that the bottom-up matching process contributes significantly to the performance of Phase 1 in HELIOS-U. In Figure 7.5(c), we observe that the execution times of finding matching *article* nodes and matching *issue* nodes are significant in Phase 1. Note that they are sub phases in the bottom-up matching process. We observe that up to 66.84% and 24.20% of the total execution time of Phase 1 in HELIOS-U are from the

CHAPTER 7. PERFORMANCE STUDY OF XANDY-U AND HELIOS-U

Figure 7.5: HELIOS-U and XANDY-U: Phase 1 vs Phase 2.

execution time of finding matching *article* nodes and matching *issue* nodes. Hence, the bottom-up matching process contributes up to 91.04% of the total execution time of Phase 1 in HELIOS-U. The execution time of finding matching *author* leaf nodes takes only up to 19.43% of the total execution time of Phase 1 in HELIOS-U (“SIGMOD-01”). Similarly, the bottom-up matching process in XANDY-U also contributes significantly to the performance of Phase 1 (Figure 7.5(d)). The execution time of bottom-up matching process takes up to 98.17%. This is much higher compared to the execution time of finding matching leaf node groups that only contributes up to 32.86% (“SIGMOD-04”). The execution time proportion of sub phases in Phase 2 will be presented in Section 7.6.

7.4 File Size vs Execution Time

In this section, we shall analyze the effect of file size to the performance of our approaches. We use “SIGMOD-06” data set. We fixed the number of nodes and increased the file size by increasing the length of each leaf node value. For each leaf node, we multiply its content length by a factor Ω . Hence, the file size of the XML documents will be changed (from 180KB up to 1,013KB). Figure 5.12(a) depicts the correlation between factor Ω and file size. We set the percentage of changes and similarity threshold θ to “3%” and “0.0”, respectively. Note

Figure 7.6: File Size vs Execution Time.

that we use same first version of XML documents as used in Chapter 5; however, the second versions are generated in different ways as we do not have move operations in *unordered* XML documents. This set of experiments also shows us the effects of the increment of content length of leaf nodes to the performance of our approaches.

Figure 7.6(a) depicts the performance of Phase 1 in our approaches for different file sizes. When the file size is increased up to almost 6 times, the execution times of HELIOS-U and XANDY-U are only increased by 26.77% (around 1.182 seconds) and 3.05% (around 0.686 second) , respectively. Figure 7.6(b) depicts the performance of Phase 2 in our approaches for different file sizes. For the similar file size increment, the execution times of Phase 2 of HELIOS-U and XANDY-U are only increased by 35.50% (around 0.480 second) and 5% (around 0.379 second), respectively. The overall performance of our approaches for different file size is depicted in Figure 7.6(c). The execution times of HELIOS-U and XANDY-U are only increased by 28.81% (around 1.662 seconds) and 3.54% (around 1.066 seconds), respectively, when the file size is increased up to almost 6 times. From this set of experiments, we conclude that the performances of HELIOS-U and XANDY-U are more influenced by the number of nodes compared to the file size. This is because when the number of node increases, we have more tuples in the relations.

CHAPTER 7. PERFORMANCE STUDY OF XANDY-U AND HELIOS-U

Figure 7.7: Percentage of Changes vs Execution Time.

7.5 Percentage of Changes

In this section, we shall present the performance of HELIOS-U and XANDY-U when we vary the percentages of changes. “SIGMOD-05” data set is used. The percentage of changes is varied from 3% to 30%. We distributed the percentage changes equally for each type of changes. The similarity score threshold θ is set to “0.00”.

Figure 7.7(a) depicts the performance of Phase 1 in our approaches. HELIOS-U and XANDY-U are slightly influenced by the changes to the percentage of changes. The differences between the minimum and maximum execution times of XANDY-U and HELIOS-U are around 8.9% and 9%, respectively. Figure 7.7(b) depicts the performance of Phase 2 in our approaches. We observe that both approaches are affected by the changes to the percentage of changes. As the percentage of changes increases, the execution time of our approaches also increases. When the percentage of changes is increased by 1%, the performances of HELIOS-U and XANDY-U become, on average, 2.95% and 1.23% slower, respectively. Figure 7.7(c) depicts the overall performance of our approaches. The performances of HELIOS-U and XANDY-U are affected by the changes to the percentage of changes. We

Figure 7.8: HELIOS-U: Execution Time Proportion for Different Types of Changes.

shall see the influences of the changes to the percentage of changes to the performances of HELIOS-U and XANDY-U when there is only one type of changes in Section 7.7.

7.6 Execution Time Proportion of Different Types of Changes

In this set of experiments, we shall see the execution time proportion of each type of changes. We used first eight data sets and set the percentages of changes to 3% and 9%.

7.6.1 HELIOS-U

Figures 7.8(a) and 7.8(b) depict the proportion of execution times of detecting insertions, deletions, and updates when the percentages of changes are set to 3% and 9%, respectively. The execution times of detecting insertions and deletions contribute up to 47.52% and 43.18% of the overall performance of Phase 2, respectively. On average, detecting insertions and deletions take 40.68% and 34.17% of the overall execution time of Phase 2, respectively. We observe that up to 48.10% (on average, 25.15%) of the overall execution time of Phase 2 is taken by the execution time of detecting updates.

CHAPTER 7. PERFORMANCE STUDY OF XANDY-U AND HELIOS-U

Figure 7.9: XANDY-U: Execution Time Proportion for Different Types of Changes.

Furthermore, Figures 7.8(c) and 7.8(d) depict the proportion of execution times of detecting insertions/deletions internal and leaf nodes when the percentages of changes are set to 3% and 9%, respectively. We observe that the execution times of detecting the inserted/deletion leaf nodes significantly influence the performance of performance of Phase 2. This is because of the same reason as discussed in Section 5.7.

7.6.2 XANDY-U

Figures 7.9(a) and 7.9(b) depict the proportion of execution times of detecting insertions, deletions, and updates when the percentages of changes are set to 3% and 9%, respectively. When the percentage of changes is set to 3%, we observe that the execution times of detecting insertions and deletions contribute up to 47.52% (on average, 44.19%) and 43.18% (on average, 35.55%) of the total execution time of Phase 2, respectively. On the other hand, when the percentage of changes is set to 9%, up to 47.61% and 45.93% of the total execution time of Phase 2 are taken for detecting the insertions and deletions, respectively. Detecting updates contributes up to 29.71% (on average, 17.49%) to the overall performance of Phase 2.

Now, we elaborate further on the finer granularity of performance of detecting insertions and deletions. Figures 7.9(c) and 7.9(d) depict the proportion of execution times of detecting

CHAPTER 7. PERFORMANCE STUDY OF XANDY-U AND HELIOS-U

Figure 7.10: HELIOS-U: Different Types of Changes.

insertions/deletions internal and leaf nodes when the percentages of changes are set to 3% and 9%, respectively. We observe that detecting the insertions/deletions of leaf nodes has significant impact to the overall performance of Phase 2. The execution time of detecting insertion of leaf nodes contributes up to 46.55% of the overall performance of Phase 2. Up to 45.21% of the total execution time of Phase 2 is the execution time of detecting deleted leaf nodes. This happens because of the same reasons as discuss in Chapter 5. Note that the SQL queries for detecting inserted/deleted leaf nodes in XANDY-U and XANDY-O are almost similar. The differences between these SQL queries have been discussed in Chapter 6.

7.7 Different Types of Changes

In this section, we shall observe the effects of percentage of different types of changes to the performances of XANDY-U and HELIOS-U. We use “Sigmod-05” data set as the first version and set similarity score threshold θ to “0.00”. The percentages of changes are between “1%” to “9%”. We generated four second versions by using our changes generator such that the changes are content updates only (denoted by “Update”), deletions only (denoted by “Deletion”), and insertions only. The insertions can occur in two following ways. First, the insertion may occur in any parts of the XML documents (denoted by “Insertion”). Second, the insertion may occur only at the end of the XML documents (denoted by “Append”).

Figures 7.10(a) and 7.10(d) depict the performance of HELIOS-U when there is only a particular type of changes. We observe that the execution time of Phase 1 of HELIOS-U is not significantly affected. The difference between the maximum and minimum execution time is less than 1 second. The performance of Phase 2 of HELIOS-U is also not

CHAPTER 7. PERFORMANCE STUDY OF XANDY-U AND HELIOS-U

Figure 7.11: XANDY-U: Different Types of Changes.

significantly influenced by the changes to percentage of changes when there is only insertion/append/deletion. Observe that the difference between the maximum and minimum execution time is less than 0.1 second. When there is only update operation, the performance of Phase 2 of HELIOS-U is affected. We analyzed that the execution time of detecting updated leaf nodes is increased as the percentage of updated operation increases (Figure 7.10(e)). Recall that we may get incorrect result when we detect the updated non-inlined leaf nodes (Chapter 6). When the percentage of update operation is increased, the possibility of having incorrect result will also be increased. That is, the execution time of the *refineUpdatedLeafNodes* algorithm will be increased if we have more *incorrect* tuples.

Figures 7.11(a) and 7.11(d) depict the performance of XANDY-U when there is only a particular type of changes. Similar to HELIOS-U, Phase 1 of XANDY-U is slightly affected. The difference between the maximum and minimum execution time is less than 0.7 second. Phase 2 of XANDY-U is also not significantly influenced by the changes to percentage of changes when there is only insertion/append/deletion. The execution time of Phase 2 of XANDY-U is affected by the percentage of updated leaf nodes (Figure 7.11(d)). This is because of the above reason. Figure 7.11(e) depicts the execution time details of Phase 2 of XANDY-U.

7.8 Depths of XML Documents

In this section, we analyze the effects of the depth of XML documents to the performance of our approaches.

A set of synthetic XML documents whose maximum node levels are between 5 and 14 is generated. The number of nodes of the first versions is fixed to 2860. The percentage

Max Depth	Phase1			Phase 2	Overall
	SP1	SP2	Total		
5	0.225	4.665	4.911	0.632	5.543
8	0.194	6.350	6.544	0.655	7.199
14	0.203	11.191	11.394	0.608	12.002

SP1: Finding matching leaf node groups

SP2: Bottom-up matching

(a) XANDY-U

(b) HELIOS-U

Figure 7.12: Depth vs Execution Time.

of changes and the similarity score threshold θ are set to “3%” and “0.00”, respectively. Figure 7.12(a) depicts the performance of XANDY-U for different maximum node levels. Phase 1 is significantly influenced by the depth of XML documents. The performance of XANDY-U becomes 33.33% slower when the depth of XML documents is changed from 5 to 9. When we increase the depth of XML documents form 9 to 14, we observe that the execution time of XANDY-U is increased by 74.1%. Similar to XANDY-O, we notice that the performance of XANDY-U is affected by the bottom-up matching process (“SP2”, Figure 7.12(a)). This is because of the same reasons as discussed in Chapter 5.

Similar to HELIOS-O, we analyze the performance of HELIOS-U by changing the numbers of *non-inlined* nodes. We increase the number of *non-inlined* leaf nodes up to three additional *non-inlined* leaf nodes while the numbers of nodes in the first versions of XML documents are fixed to 2345 nodes. Figure 7.12(b) depicts the performance of HELIOS-U for different numbers of non-inlined leaf nodes. We observed that HELIOS-U is influenced by number of *non-inlined* leaf nodes. The performance of HELIOS-U becomes 34.76% faster when the number of non-inlined leaf node is increased from 1 to 3. This is because of the similar reason as presented in Chapter 5.

7.9 Similarity Score Threshold

In this section, we shall analyze the effects of similarity score threshold θ to our approaches. First, we shall analyze the influences to the execution time. Then, we shall observe the result quality for different values of similarity score threshold θ .

7.9.1 Similarity Score Threshold vs Execution Time

In this set of experiments, we set the value of similarity score threshold θ between “0.00” and “0.60”. We use “Sigmod-05” data set with “3%” and “9%” changes.

CHAPTER 7. PERFORMANCE STUDY OF XANDY-U AND HELIOS-U

Figure 7.13: Similarity Score Threshold θ vs Execution Time.

Figures 7.13(a) and 7.13(d) depict the performance of XANDY-U and HELIOS-U when we vary the value of similarity score threshold θ . Similar to XANDY-O and HELIOS-O, as the value of similarity score threshold θ becomes larger, the performance of Phase 1 of XANDY-U and HELIOS-U becomes slightly faster. The performances of XANDY-U and HELIOS-U for various similarity score threshold θ are depicted in Figures 7.13(b) and 7.13(e). We observe that the execution times of XANDY-U and HELIOS-U increase as the value of similarity score threshold θ increases. For overall performance, we notice that the performance of Phase 1 of XANDY-U and HELIOS-U becomes faster when we have higher value of similarity score threshold θ (Figures 7.13(c) and 7.13(f)). Note that the reasons behind the results of this set of experiments are similar to the one in Chapter 5.

Figure 7.14: Result Quality vs Similarity Score Threshold.

7.9.2 Similarity Score Threshold vs Result Quality

In this section, we shall present the effect of similarity score threshold θ to the result quality. We set similarity score threshold θ between “0.00” to “0.60”. We use “Sigmod-02” data set with “6%” and “18%” changes. Note that the definition of result quality is similar to the one in Chapter 5.

Figure 7.14(a) depicts the result quality of XANDY-U and HELIOS-U for different values of similarity score threshold θ when we set the percentage of changes to 6%. XANDY-U is able to detect the optimal delta until the similarity score threshold θ is set to 0.40. However, when the similarity score threshold θ is greater than 0.40, the result quality of XANDY-U is reduced. We also notice that HELIOS-U is able to detect the optimal deltas regardless of different values of similarity score threshold θ . Figure 7.14(b) depicts the result quality of XANDY-U and HELIOS-U for different values of similarity score threshold θ when we set the percentage of changes to 9%. We observe that the result quality of HELIOS-U is reduced when the similarity score threshold θ is greater than 0.50. On the other hand, the result quality of XANDY-U is reduced when the similarity score threshold θ is greater than 0.40. From this set of experiments, we conclude that the result quality of our approaches depends on not only the similarity score threshold θ , but also the percentages of changes. We also observe that the result quality is also affected by the distribution of changes. Note that we have discussed this issue in Chapter 5.

7.10 Result Quality

In this section, we analyze the result quality of HELIOS-U and XANDY-U. We also compare the result quality of HELIOS-U and XANDY-U to the one of X-Diff [WDyC03]. We used

Figure 7.15: Result Quality.

data set “SIGMOD-02”, and constructed the second versions with various percentages of changes. Note that the definition of result quality is similar to the one in Chapter 5.

Figure 7.15(a) depicts the result quality comparisons between HELIOS-U, XANDY-U and X-Diff. We observe that the result quality of HELIOS-U is almost similar to the one of XANDY-U. When the percentage of changes is set to 30%, the result quality of HELIOS-U is slightly better. We analyzed the deltas produced by HELIOS-U and XANDY-U. We find out that there is a pair of deleted and inserted leaf nodes that is detected as an updated leaf node by HELIOS-U. XANDY-U detects as a deleted leaf node and an inserted one. Furthermore, when the percentage of changes is set to 30%, HELIOS-U and XANDY-U cannot find the optimal delta. Let us elaborate on this issue by using an example. Consider T_1 and T_2 as depicted in Figure 7.15(b). In this example, we notice that node *authors* is not deleted and there are two updated leaf nodes. However, HELIOS-U and XANDY-U detect as a deletion of a subtree rooted at *authors* and an insertion of a subtree rooted at *authors*. Obviously, this reduces the quality of deltas detected by HELIOS-U and XANDY-U. Note that the similarity score between subtree rooted at node *authors* in T_1 and T_2 is equal to “0”. In addition, we observe that HELIOS-U and XANDY-U are able to produce better

result quality than X-Diff when the percentages of changes are greater than 15%. This is because of the similar reasons as presented in Chapter 5.

7.11 Summary

We have presented our performance study of our approaches compared to X-Diff. We summarize our performance study as follows.

- The experimental results show that the relational-based approaches indeed have better scalability than the memory-based approaches. We observe that X-Diff is not able to detect the changes to the data sets larger than SIGMOD-06/TCSD-04 due to lack of memory.
- For DCSD data sets, XANDY-U is up to 17 times faster than X-Diff for large data sets. The performance of HELIOS-U is up to 100 times faster than X-Diff. We also notice that HELIOS-U outperforms XANDY-U (up to 63 times faster).
- When we use TCSD data sets, we observe that XANDY-U and HELIOS-U are up to 3 times and 9 times faster than X-Diff, respectively. HELIOS-U also outperforms XANDY-U (up to 7 times faster).
- HELIOS-U is more scalable than XANDY-U as XANDY-U does not return results for data sets larger than SIGMOD-12 after 100,000 seconds.
- The number of nodes has more effects to the performance of our approaches compared to the file size. When the file size is increased up to more than 5 times, the performance of our approaches becomes up to 28% slower.
- Phase 1 of our approaches is the bottle neck. It can take up to more than 97% of the overall performance.
- The percentages of changes indeed have influences to the performance of our approaches. When the percentage of changes is increased by 1%, the performance of our approaches becomes up to 8% slower.
- The depth of XML documents is also affected our approaches, especially the performance of Phase 1. This is because we determine the best matching subtrees in bottom-up fashion.

- We find out that the performance of our approaches is slightly affected by the changes to similarity score threshold θ . We notice that the result quality of our approaches is also affected by similarity score threshold θ . In addition, we also show that the result quality is also influenced by the percentages of changes.
- Our approaches have superior result quality compared to X-Diff.