SUPPORTING POLYMORPHISM IN XML DATA

By

SHUOHAO ZHANG

A dissertation submitted in partial fulfillment of
the requirements for the degree of
DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science
AUGUST 2006
To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of SHUOHAO ZHANG find it satisfactory and recommend that it be accepted.

___________________________________
Chair

___________________________________
ACKNOWLEDGMENT

First of all, I would like to thank my advisor Dr. Curtis Dyreson for his guidance throughout my graduate study. I have learned a great deal and had a lot of fun in the past six years. I would like to thank my other committee members, Dr. Zhe Dang and Dr. Carl Hauser, for their important advice. Thanks are also due to many people who have given me valuable insight and help over the years – Dr. Anneliese Andrews, Dr. David Bakken, Dr. David Benson, Dr. Zhe Dang, Dr. Murali Medidi and Dr. Richard Snodgrass.

I am grateful to the School of Electrical Engineering and Computer Science for its great faculty and for the financial support it has offered me. Especially, my thanks go to chair of graduate studies Dr. Ali Saberi and graduate advisor Ruby Young. I have been a teaching assistant since I came to Washington State University. This is an interesting and nourishing experience, and I thank all the instructors and students that have supported my work. I would also like to thank the School of Electrical Engineering and Computer Science and the Graduate School for awarding me travel grants to attend an academic conference.

Finally, thanks are due to my family. Most importantly, I am indebted to my lovely wife Jingqun. Her constant patience, understanding and encouragement have been so important to me. Also, my accomplishment wouldn’t have been possible without the support from my parents Chen Youzhao (陈又钊) and Zhang Liren (张立人), and my wife’s parents Yu Xiulan (余秀兰) and Xu Delin (许德林). They are always there for us, with such great love.
The Extensible Markup Language (XML) is rapidly becoming the standard language for web data representation and exchange. XML data is structured as a hierarchy. Different users and organizations may adopt different hierarchies for their data. It often happens that the same information can be represented in different hierarchies in XML. We use the term polymorphism to characterize this property of the XML format.

Polymorphism is very common in the practical use of XML data, yet there has been relatively little research on it. This dissertation examines issues important to the support for polymorphism in XML data. First, we examine the semantics of XML hierarchy. We formulate a semantics that captures the data modeling functionality of XML hierarchy. We then look at several applications in which polymorphism in XML data is of great significance and vast presence. These include XML compaction, XML restructuring, symmetric exploitation of XML data, and schema-less, semantics-based change detection for XML. Studying these issues is beneficial to the advancement of XML data management in general and to the understanding of XML data polymorphism in particular.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENT</td>
<td>iii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>iv</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>ix</td>
</tr>
<tr>
<td>1. OVERVIEW</td>
<td>1</td>
</tr>
<tr>
<td>2. COMPACTING XML DATA</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Motivating Example</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Formal Framework</td>
<td>11</td>
</tr>
<tr>
<td>2.3.1 Graph, Tree and Forest</td>
<td>11</td>
</tr>
<tr>
<td>2.3.2 Forest Signature</td>
<td>13</td>
</tr>
<tr>
<td>2.3.3 Complete Signatures</td>
<td>15</td>
</tr>
<tr>
<td>2.3.4 Identifier, Type, and Type Dependency</td>
<td>16</td>
</tr>
<tr>
<td>2.4 A Translative Semantics for XML</td>
<td>20</td>
</tr>
<tr>
<td>2.4.1 An Example</td>
<td>20</td>
</tr>
<tr>
<td>2.4.2 Description of the Translative Semantics</td>
<td>26</td>
</tr>
<tr>
<td>2.5 Compaction</td>
<td>28</td>
</tr>
<tr>
<td>2.5.1 Restructuring</td>
<td>28</td>
</tr>
<tr>
<td>2.5.2 Grouping</td>
<td>32</td>
</tr>
<tr>
<td>2.5.3 Summary Restructuring and Grouping</td>
<td>36</td>
</tr>
<tr>
<td>2.5.4 Compacting DBLP data</td>
<td>37</td>
</tr>
<tr>
<td>2.6 Related Work</td>
<td>39</td>
</tr>
<tr>
<td>2.7 Conclusion</td>
<td>40</td>
</tr>
<tr>
<td>3. POLYMORPHIC XML RESTRUCTURING</td>
<td>42</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>42</td>
</tr>
<tr>
<td>3.2 Related Work</td>
<td>49</td>
</tr>
<tr>
<td>3.2.1 LCA-based Querying Techniques</td>
<td>49</td>
</tr>
<tr>
<td>3.2.2 Restructuring</td>
<td>51</td>
</tr>
<tr>
<td>3.2.3 Other Related Work</td>
<td>52</td>
</tr>
<tr>
<td>3.3 Preliminaries</td>
<td>53</td>
</tr>
<tr>
<td>3.3.1 Tree, Forest and Node Type</td>
<td>53</td>
</tr>
<tr>
<td>3.3.2 Signature</td>
<td>54</td>
</tr>
<tr>
<td>3.3.3 Forest Restructuring</td>
<td>55</td>
</tr>
<tr>
<td>3.3.4 Closest Relation</td>
<td>57</td>
</tr>
<tr>
<td>3.3.5 Preserving Closeness</td>
<td>59</td>
</tr>
<tr>
<td>3.4 The Poly-transform</td>
<td>61</td>
</tr>
<tr>
<td>3.4.1 The Poly-transform Algorithm</td>
<td>62</td>
</tr>
<tr>
<td>3.4.2 Implementation Complexity</td>
<td>64</td>
</tr>
<tr>
<td>3.4.3 Reversibility</td>
<td>67</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1 author.xml........................................................................................................10
Figure 2 pub.xml...........................................................................................................10
Figure 3 Are these two trees the same semantically?.....................................................21
Figure 4 Gluing subtrees...............................................................................................21
Figure 5 Resulting graphs after subtree gluing..............................................................22
Figure 6 Connecting related nodes................................................................................22
Figure 7 Result of node connecting................................................................................23
Figure 8 The final graph...............................................................................................23
Figure 9 Node gluing.....................................................................................................26
Figure 10 Node connecting............................................................................................27
Figure 11 Grouping.......................................................................................................33
Figure 12 A possibly illegitimate grouping....................................................................34
Figure 13 An example of the book data in the DBLP collection ....................................37
Figure 14 Querying the same information represented in ............................................44
Figure 15 A document with same-label nodes at different levels ................................50
Figure 16 Nodes closest to node 3 in books.xml............................................................59
Figure 17 The poly-transform algorithm.......................................................................61
Figure 18 Restructure books.xml with target signature
   bibliography#publisher#book#(title,author,editor).....................................................63
Figure 19 Merging lists of nodes to find closest nodes...................................................65
Figure 20 Placing titles below publishers in the target..................................................66
Figure 21 Illustration for proof of Theorem 1.................................................................71
Figure 22 Experiment one measures three signatures....................................................81
Figure 23 Relative cost of initialization vs. poly-transform............................................81
Figure 24 Poly-transform enabled XQuery vs. XQuery..................................................82
Figure 25 A fragment of cs_authors.xml.....................................................................89
Figure 26 A fragment of cs_books.xml........................................................................90
Figure 27 EBNF grammar for the extended XPath.......................................................93
Figure 28 Nodes closest to the first title.......................................................................97
Figure 29 The search is restricted by the type information............................................98
Figure 30 Signature of cs_authors.xml.......................................................................102
Figure 31 An algorithm for computing node type.......................................................104
Figure 32 A naïve, in-memory algorithm for evaluating the closest axis.......................106
Figure 33 Numbering the data tree of cs_authors.xml...............................................109
Figure 34 The LCA-join of publisher and title...............................................................110
Figure 35 The original version of a bibliography document.........................................121
Figure 36 A new version of the bibliography document..............................................121
Figure 37 An alternative new version of the bibliography document............................122
Figure 38 Structurally identical nodes........................................................................129
LIST OF TABLES

Table 1 Cardinality ratios for the example graph ..............................................................31
Table 2 Reversibility for sample signatures ......................................................................76
Table 3 Element identities for bib1.xml .....................................................................140
Table 4 Identities for bib2.xml ...................................................................................147
Table 5 Values of identities for book.............................................................................147
1. OVERVIEW

The Extensible Markup Language (XML) [57] is a general-purpose markup language for data representation and exchange. XML employs markups (also called tags) to annotate data and to describe the structure of data as well. Accompanied by the proliferation of the Internet, XML has rapidly becoming the standard language for data publishing and exchange on the web.

XML data is structured as a hierarchy. An XML parser parses an XML document and produces a tree data model. Since different users and organizations may adopt different hierarchy structures for their data, it often happens that the same information can be represented in different hierarchies in XML. We use the term polymorphism to characterize this property of the XML format.

Polymorphism is a commonly used term in computer science, especially in object-oriented design. However, polymorphism of XML data is different and is novel in the XML literature. The word “polymorphism” is literally composed of “poly-”, meaning “many”, “-morph-”, meaning “form”, and “-ism”, meaning “a distinctive system or theory”. In this dissertation, “polymorphism in XML Data” describes a theory about XML’s capability to let data assume many forms. Polymorphism is very common in the practical use of XML data, yet there has been relatively little research on it. This dissertation presents a formal framework that facilitates the understanding of XML polymorphism, and examines several important problems pertinent to XML
Polymorphism. In this overview, we outline the key questions this dissertation raises, with a brief explanation of how each is approached.

Polymorphism refers to the phenomenon that the same information can be represented in different hierarchies. This naturally raises the following question:

*How do we determine the “sameness” of two pieces of XML data, when their structures are different?*

Basically, we need to define a *semantics* for XML data that makes semantic comparison possible. Since XML as a format does not carry semantics *per se*, what such a semantics does is to capture XML’s practical usage. We propose that the semantics of XML data be represented by a graph. Based on this semantics, each XML data collection, modeled as a forest (a forest is more general than a tree and can model a fragment of XML data), can be uniquely mapped to a graph. Thus, comparison of the data translates to a graph isomorphism problem. To translate an XML data collection to a graph, we *glue* nodes of an XML forest that represent the same real-world entity, and *connect* (add an edge between) each pair of nodes that are “related”. Two issues are crucial to this process. The node gluing is only possible with a set of specifications of the dependencies among the various kinds of nodes in a forest, called *identifiers*; the node connecting depends on the observation that “closeness” translates to “relatedness” in a hierarchy. We formalize these intuitive notions and a formally defined semantics of XML data follows. This semantics for XML data is central to the reasoning of and hence support for polymorphism in XML data. Chapter 2 explains this semantics in detail.
With an understanding of what constitutes semantic equivalence for structurally different data, we then study some problems of great practical importance in which polymorphism is present. The first problem is the compaction of XML data, described as follows.

*How do we produce an XML forest smaller than an input XML forest, without losing any data?*

Since XML is used to exchange data, if the data size can be reduced, the cost of exchanging data can also be reduced. The process of reducing the size of data without loss of information is called compression. Compaction is a special kind of compression. Compression usually encodes the input in a different language or format. In compaction, however, we keep the output in the same language as the input. The formal framework developed earlier in Chapter 2 provides a basis for the discussion of XML data compaction. Chapter 2 then presents two compaction techniques, *restructuring* and *grouping*. Restructuring compacts XML data by changing its hierarchy structure; grouping groups siblings, keeping the hierarchy structure unchanged.

In general, restructuring itself is a very useful operation for manipulating XML data. It allows users to rearrange the information to conform to a given hierarchy structure. Restructuring is one of the most important functionalities of XML query languages; this is especially true for XSLT [62], whose primary duty is structural transformation for XML data. However, the restructuring mechanism in the current XML query languages has major inconveniences. To restructure data, users have to first know the specific
structure of the input data, and then use a query language that is difficult to master, and complicated and error-prone in use. These limitations give rise to the following question.

*Can we restructure without having to know the structure of the input data, and can we restructure without having to use an involved query language?*

Chapter 3 proposes *polymorphic restructuring* as a solution to the above problem. Polymorphic restructuring requires little or limited knowledge of the structure of the input data, and uses a simple, declarative specification of the target, called a *signature*. It is polymorphic in the sense that one restructuring operation works on inputs of different structures. Such polymorphism is both convenient for a naive user and necessary when an expert queries a heterogeneous or evolving data collection. An important issue in polymorphic restructuring is *reversibility*, the property that the restructured target data is lossless with respect to the source data. Some target signatures result in reversible restructuring while some do not. We examine this issue in detail. Finally, this chapter also sketches a join technique that can be used to efficiently restructure XML data.

Compaction and restructuring are two direct applications of XML polymorphism. In each application, we transform source data to a target hierarchy while keeping the data semantics intact. In a more general setting of polymorphism, a user may not know or care how data is organized in a hierarchy, i.e., any hierarchy is possible. In such a scenario, the question becomes:

*Can a user effectively exploit data with an unknown hierarchy structure?*
This is a somewhat open-ended question, as there is no specific direction such as with compaction (finding a smaller XML representation for the same data). Chapter 4 proposes symmetric exploitation of XML data, as opposed to the popular, asymmetric path expressions which currently is the principal means of locating data in XML. Path expressions are brittle because they often depend on the structure of data and break if the data is structured differently. The structure of data could be unfamiliar to a user, may differ within a data collection, or may change over time as the schema evolves. We propose a symmetric construct that locates related nodes in an XML forest, independent of a specific structure. It can augment many XPath expressions and can be seamlessly incorporated in XQuery or XSLT. This leads to a querying mechanism that adeptly accommodates polymorphism in XML data.

Under the graph-based semantics for XML discussed in Chapter 2, it is straightforward to determine whether two pieces of XML data are different. However, as mentioned before, this semantics is dependent on a given set of node identifiers. Sometimes such identifiers are not available, thus the assumption of the graph semantics does not hold.

Usually identifiers are represented within a schema such as an XML Schema [60] or DTD [57] file. Such schema specifications also specify hierarchy structure. We use schema-less to describe the circumstance where neither identifiers nor structure knowledge is available. Chapter 5 examines this situation in the context of change detection, which aims to find out where data differs.
Can we define and detect the change between data, with no knowledge of its schema (hierarchy structure and identifiers)?

We call this application schema-less, semantics-based change detection. It is schema-less because no prior knowledge of the data’s schema is assumed; it is semantics-based since it does not assume structural similarity between the data.

Change detection is very useful, for example, to reduce space in a historical data collection and to support temporal queries. Change detection has been extensively researched, but only for data of similar hierarchy structure. Most previous techniques break down when the structural change is significant. In fact, it has not been defined what constitutes a “change” when data of different structures can be considered the same.

Chapter 5 develops a change detection algorithm based on the semantics, rather than on the structure, of the data. It is important to note that, here “semantics” has a different meaning from in Chapter 2, where identifiers are given in advance and remain unchanged across various data collections. Instead, the algorithm itself computes node identities. Identities are intrinsically different from identifiers, because each node in an XML tree has a unique identity, and nodes in different data collections that are considered the same do not necessarily have the same identity. The algorithm then uses identities to associate nodes in different data collections. Changes in the data can be discovered by looking at these associations. In this application, we accommodate polymorphism in XML data from a new angle, as we have a different assumption.
2. COMPACTING XML DATA

Compression aims to reduce the size of data without loss of information. Compaction is a special kind of compression in which the output is in the same language as the input. Compaction of an XML data forest produces a smaller XML forest, without losing any data. This chapter develops a graph semantics for XML data, which serves as a basis for the discussion of XML data compaction. Two compaction techniques, restructuring and grouping, are then presented.

2.1 Introduction

*Compression* aims to reduce the size of data without loss of information. It is useful because smaller data can save storage space and also network bandwidth when data is transmitted. A *compressor* generates a file smaller in size than the original; feeding this file to a *de-compressor* can recover the original.

XML is rapidly becoming a dominant media for data exchange over the Internet. Because XML data is usually quite verbose, compression is an important issue for XML. Several tools are already available for XML compression. One can use either a general purpose compressor such as *gzip*, or an XML-specific compressor (such as XMill [38]) to compress XML data.

This chapter addresses a special kind of compression, called compaction, where the compressed output remains as XML. Existing compression techniques do not compact
the data because they all produce a compressed file in a non-XML format, which only a special-purpose de-compressor can understand. For instance, a gzipped XML file is not XML. By keeping the compressed data in XML syntax, we have the benefit to be able to evaluate the quality of a compression in terms of the number of nodes in the data model, which is not available in other compression techniques. Another benefit of compaction is that it is orthogonal to other compression techniques, so an XML file can be compacted and then compressed.

The general idea behind compaction is that the same data can be represented in XML in (several) different structures. For instance data about books and authors could have either <book> elements enclosed within <author> elements or vice-versa. A <book> that is written by several <author>s will be duplicated for each <author> if <book> elements are enclosed within <author> elements. But a different structure may duplicate <author> elements. The different structures usually will have different sizes. For compaction, the structure with the smallest size is the most desirable.

Compaction only works if the different representations all have the “same data”. Since compaction may change the order of siblings or change “white-space” text nodes, compaction is only possible for XML in data-centric scenarios where order and white-space are unimportant. We formalize the semantics of data-centric XML by introducing a graph-based representation that allows data in different structures to be semantically compared. Under this semantics, two XML data collections are considered equal if and only if they are mapped to isomorphic graphs. Compaction can be achieved as follows.
First, XML data is translated to its graph semantics. This graph is then transformed into different forests, which correspond to different representations of the same data. These forests are of different sizes; those with a size smaller than the original data are all more compact.

This chapter is organized as follows. Section 2.2 gives a motivating example. Section 2.3 presents a framework in which XML compaction can be formally discussed. Section 2.4 introduces a semantics for XML that translates an XML data collection to a graph. Section 2.5 presents two compaction techniques that transform a forest to another without affecting the semantics; it then discusses how the most compact forest among these forests could be found. As a concrete example, in Section 2.5.4 we show an experiment in which compaction significantly reduces the size of DBLP data – 18% in terms of file size and 25% in terms of number of elements. Section 2.6 presents related work, and Section 2.7 concludes the chapter.

### 2.2 Motivating Example

Consider two XML data documents, `author.xml` and `pub.xml`, shown respectively in Figure 1 and Figure 2. The data is simple enough that we can rely on readers of the data to agree on its intended semantics. In `author.xml` author n1 writes a book t1, published by p1; author n2 is a co-author on book t1 and also independently writes a book t2, published by p2. `pub.xml` contains exactly the same information except that the structure is different. At the top level the data is grouped by publishers while in
author.xml it is grouped by authors. Author elements enclose book elements in author.xml but are nested within book elements in pub.xml.

Intuitively, author.xml and pub.xml have the same data, though they differ in how that data is represented. Each document has data about the same two authors, the same two publishers, and the same two books. Each document similarly relates each book, author, and publisher, e.g., in both documents book t2 is authored by author n2 and published by publisher p2. Section 2.4 develops a formal framework that allows the implicit meaning or semantics of an XML data collection to be determined and compared. For compaction, what interests us is the fact that author.xml and
pub.xml have the same data but are of different sizes. Excluding text data, there are 14 elements in author.xml but only 13 elements in pub.xml. This suggests that author.xml could be compacted to at least the size of pub.xml. Of course, other compression techniques could potentially reduce the physical size of author.xml much further; however, only compacting produces output in XML.

Compaction is concerned with logical redundancy as much as physical redundancy. Note that in compaction we can measure the size of XML data by the number of elements. This differs from common compression tasks in which the size of a compressed file can only be measured by the disk space it occupies. While file size reflects the physical redundancy in a file, the number of duplicate nodes gives a better measure for redundancy on the logical level. By preserving the XML syntax in the output, compaction rearranges the original data to a new form with fewer places that are subject to update anomaly. Certainly, compacting an XML file may potentially (and usually does) compress the data at the same time. Though fewer elements do not guarantee a physically smaller file in general, it is usually so in practice.

2.3 Formal Framework

This section defines graph, tree and forest. It then develops a semantics for XML.

2.3.1 Graph, Tree and Forest

We start with definitions of graph, tree and forest.
**Definition 1** [graph] A graph is a five-tuple \((V, E, \Sigma, L, C)\), where

- \(V\) is a *node* set,
- \(E: V \times V\) is an *edge* set,
- \(\Sigma\) is an *alphabet* of labels and text values,
- \(L: V \rightarrow \Sigma\) is a *label function* that maps a node to its label, and
- \(C: V \rightarrow \Sigma \cup \{\varepsilon\}\) is a *value function* that maps a node to its *value* (a node \(v\) has an empty value if \(C(v) = \varepsilon\)). \[\square\]

This definition of graph adds label and value on top of the usual notion of graph. This is convenient in defining an XML document as a tree in which each node has a label and a value.

**Definition 2** [tree] A tree \(T = (V, E, \Sigma, L, C)\) is a special kind of graph, where

- \(V\) is the *node* set. \(r \in V\) is a special node called the *root* of \(T\),
- \(E: V \times V\) is the *edge* set such that there is a path between \(r\) and every node (other than \(r\)), and there is no cycle among the edges in \(E\); if \((u, v) \in E\) and \(u\) is closer to \(r\) than \(v\), then \(u\) is the *parent* of \(v\) and \(v\) is a child of \(u\),
- \(\Sigma\) is an *alphabet* of labels and text values,
- \(L: V \rightarrow \Sigma\) is a *label function* that maps a node to its label, and
- \(C: V \rightarrow \Sigma \cup \{\varepsilon\}\) is a *value function* that maps a node to its *value* (a node \(v\) has an empty value if \(C(v) = \varepsilon\)). \[\square\]
Trees defined above are what we call XML trees. Every XML document can be modeled as a tree. Note that this tree data model differs from the DOM data model [56]; it ignores sibling order. And it does not model attributes, comments, or processing instructions. But the simpler model is sufficient for our purposes.

We often need to deal with an XML data collection, which is a group of XML documents or parts of XML documents. Since a collection could have several roots it can be modeled as a forest.

**Definition 3** [forest] A forest consists of $n$ ($n > 0$) trees. It is a five-tuple $(V, E, \Sigma, L, C)$, where $V$, $E$ and $\Sigma$ are the unions of the node sets, edge sets and label sets of the $n$ trees, respectively, and $L$ and $C$ are functions that consists of all the node-to-label and node-to-value mappings in the $n$ trees, respectively. 

2.3.2 Forest Signature

Compaction rearranges the structure of XML data to achieve compactness. This subsection introduces a specification for describing the structure of a forest.

**Definition 4** [signature] Denoted $\text{sig}(F)$, the signature of a forest $F$ is a forest such that

- $C(v) = \varepsilon$ for every node $v$ in $\text{sig}(F)$;
- if $F$ is a tree that consists of a root $r$ and forest $S$ (the root of each tree in $S$ is connected to $r$ by an edge), then $\text{sig}(F)$ is a tree that consists of a root node labeled
\(L(r)\) and the forest \(\text{sig}(S)\) (if \(F\) consists of a single node \(r\), then \(\text{sig}(F)\) contains a single node as well);

- if \(F\) is a forest that consists of \(n\) trees \(H_1, \ldots, H_n\) \((n > 0)\), then \(\text{sig}(F)\) consists of a set of trees: \(\text{sig}(H^1), \ldots, \text{sig}(H^k)\), where \(H^i \in \{H_1, \ldots, H_n\}\) \((1 \leq i \leq k)\) and \(\{\text{sig}(H^i) \mid 1 \leq i \leq k\} = \{\text{sig}(H_i) \mid 1 \leq i \leq n\}\); tree equivalence is defined in terms of isomorphism between labeled trees. □

A signature summarizes the structure of a forest but is not concerned with the values of the nodes. (The value function \(C\) of a signature forest is empty.) In a signature, sibling trees are always of different structures. A signature is similar to a Data Guide. It is unlike a schema specification such as a DTD because any forest has exactly one signature, but potentially could conform to many schemas.

Signature will be frequently used in the rest of this dissertation. Depicting forests is rather space-consuming, hence we use the following notations to textually represent a signature:

- trees in a forest are enclosed in a pair of parentheses, separated by commas, and
- the symbol # separates a root node and the forest rooted at this node.

For example, the document \texttt{author.xml} in Figure 1 has the following signature:

\[
\text{bib}\#\text{author}\#(\text{name},\text{book}\#(\text{title, publisher})).
\]
2.3.3 *Complete Signatures*

Trees in a database forest typically exhibit a high degree of structural regularity. In this section we identify a special class of forests characterized by the fact that labels in their signatures are *complete*.

**Definition 5** [complete signature and forest] A signature \( s \) is *complete* if and only if any label in \( s \) appears exactly once. Furthermore, if \( s \) is the signature of a forest \( F \), then \( F \) is *complete*. □

The forests in Figure 1 and Figure 2, for example, are both complete since every label appears once in each forest’s signature.

Complete signatures are an important class of signatures since there is no ambiguity in specifying the structure of the target data. For instance, consider the ambiguity in restructuring a data instance where \(<\text{name}>\) is a subelement of both \(<\text{author}>\) and \(<\text{publisher}>\). This data has a signature \((\text{author#name},\text{publisher#name})\) and hence is *not complete*. If the target signature in a restructuring is \(\text{name#author} \), it is ambiguous as to which \(\text{name}>\) in the source data should be used in the target. To resolve this, we need to disambiguate an author name and a publisher name. For example, we relabel the source so that its signature becomes \((\text{author#aname},\text{publisher#pname})\) and change the target signature accordingly to \(\text{aname#author}\).
In a data-centric scenario, a forest is either complete, or can be easily relabeled to be so. Hence, we make an important assumption in this chapter that compacting is applied only to complete XML data.

2.3.4 Identifier, Type, and Type Dependency

We now present some concepts that are important in the discussion of XML semantics.

**Definition 6** [identifier] Given a set $S$, if a function $f$ is an *identifier* of $S$, then for any $x \in S$ and $y \in S$, $x$ and $y$ are called *equivalent* if and only if $f(x) = f(y)$. $f(x)$ is called the *identity* of $x$. □

In this definition, the fact that $x$ and $y$ are equivalent does not necessarily mean that $x$ and $y$ are the same item in $S$. This is intrinsically different from the concept of *key*, which prohibits the coexistence of two items with the same (evaluated) key. Note that the above definition imposes the weakest constraint on an identifier $f$. Only $f$'s domain, $S$, is known. The range of $f$, the specific mapping from the domain to the range, and what constitutes equality in the range are all to be further defined. This allows a broad spectrum of definitions for identifier.

Many different identifiers are possible for the node set $V$ of a forest $(V, E, \Sigma, L, C)$. A simple (but not quite interesting) identifier $f : V \rightarrow \mathbb{Z}$ maps each node to a distinct integer by, for example, a depth first traversal of the tree. This identifier evaluates to a distinct value for each node and hence is a key for $V$. 
Labels can be used to partially identify nodes in a forest, but not to distinguish nodes of the same label. To further identify nodes of the same label, we need another characterization that identifies nodes of the same label. One such characterization is a type identifier. Here we define nodes to be of the same type if they have the same label; type identifier is defined to be an identifier that identifies nodes of the same type.¹ Such type identifiers observe the dependency among nodes of different types in a forest. For example, we may have the following dependencies in author.xml and pub.xml:

- an author depends on its corresponding name,
- a book depends on its corresponding title, and
- a name, title or publisher each depends on its value.

In each of these dependencies, one type is dependent on some other types or its own value. In a specific dependency, we call a node of the depending type a depending node; a depending node is identified by nodes or value corresponding to the deciding types, which we call the identifying information of the depending node. In general, we shall allow identifying information to be a combination of both nodes and values.

Usually, a node’s identifying information is its immediate children (nodes or values). We further observe that, regardless of the relative position of the depending types and the deciding types, the identifying information is always “closest” to a dependent node. For

¹ Note that the term “type” is commonly used in the XML database literature but with varying meanings in different research areas. With the assumption of complete forest in this chapter, the type of a node is simply its label.
example, if a book is identified by its title, then in the forest that title is closer to the book it identifies than it is to other books.

More precisely, suppose \( v \) is a dependent node and \( u \) is a type \( t \) identifying node of \( v \), then \( u \) is closest in distance to \( v \) among all type \( t \) nodes. This observation suggests that we can employ this notion of *closeness* to locate the identifying information.

**Definition 7** [related nodes] Let \( v \) be a node of type \( x \). Then \( related(v, t) = \{ x \mid x \text{ is a node of type } t \text{ and from among all the nodes of type } t, x \text{ is closest in distance to } v \} \). The distance between a pair of nodes is measured by the length of the path that connects the nodes. □

Using the notion of closest, related nodes, we formalize a type identifier as follows.

**Definition 8** [type identifier] A type identifier \( I \) of a type \( t \) is a two-tuple \((I_{\text{Type}}, I_{\text{Text}})\), where \( I_{\text{Type}} = \{ x_1, \ldots, x_m \} \) and \( I_{\text{Text}} = \{ y_1, \ldots, y_n \} \) are each a set of types, \( m \) and \( n \) are non-negative integers and they are not both zero, and \( t \in I_{\text{Type}} \). Two type \( t \) nodes \( u \) and \( v \) are *identical*, denoted \( u \equiv v \), if and only if the following holds.

- When \( m > 0 \), for each \( q \) in \( related(v, x_i) \), \( 1 \leq i \leq m \), there exists a node \( p \) in \( related(u, x_i) \) such that \( p \equiv q \); for each \( p \) in \( related(u, x_i) \), there exists a node \( q \) in \( related(v, x_i) \) such that \( p \equiv q \).
When \( n > 0 \), for each \( q \) in \( \text{related}(v, y_i) \), \( 1 \leq i \leq n \), there exists a node \( p \) in \( \text{related}(u, y_i) \) such that \( C(p) = C(q) \); for each \( q \) in \( \text{related}(u, y_i) \), there exists a node \( p \) in \( \text{related}(v, y_i) \) such that \( C(p) = C(q) \).

The following notation represents the dependency of type \( t \) on the other types:

\[
t \leftarrow x_1, \ldots, x_m; y_1, \ldots, y_n
\]

where the delimiter symbol “;” is required, even if \( m \) or \( n \) is zero. □

The above definition recursively describes how a depending node is identified by a combination of nodes of other types and some values. The base case in the recursive definition is when the set \( I_{\text{Type}} \) of type \( t \) is empty. In this case, whether two type \( t \) nodes \( u \) and \( v \) are identical is decided by comparing some values. If \( I_{\text{Type}} \) is non-empty, then whether \( u \) and \( v \) are identical is recursively determined by whether nodes of some other types are identical. As a special case, \( u \) and \( v \) are identical if they are the same node.

Using the type identifier notation, the dependencies in the motivating example are represented as follows.

- author ← name;
- book ← title;
- name ← ; name
- title ← ; title
- publisher ← ; publisher
2.4 A Translative Semantics for XML

We now present a semantics for XML. The dictionary meaning of the word “semantics” that best describes its use in our context is “the meaning or relationship of meanings of a sign or set of signs”. The signs studied here are XML data modeled as forests.

Definition 9 [translative semantics] A translative semantics of a set $S$ is a function $f : S \rightarrow T$. $S$ and $T$ are called the domain and the range of the translative semantics. $f(s)$ is called the semantics of $s$ under $f$. Two distinct elements in $S$ are semantically equal under (a translative semantics) $f$ if and only if $f$ maps them to the same item in $T$. □

A translative semantics translates each item in the domain to its semantic image in the range. We propose a translative semantics for XML that maps a forest to a graph. Using the motivating example, we first illustrate this semantics by example. A generalized description of the semantics then follows.

2.4.1 An Example

Continuing with the motivating example introduced in Section 2.2, we analyze why author.xml and pub.xml are semantically equivalent. This analysis is informal in nature yet helps us understand and formalize this semantics for XML. In the following, we show a series of operations that reduces each tree to a graph. Figure 3 shows the two trees that are created when the documents are parsed.
Figure 3 Are these two trees the same semantically?

Figure 4 Gluing subtrees
Figure 5 Resulting graphs after subtree gluing

Figure 6 Connecting related nodes
Figure 7 Result of node connecting

Figure 8 The final graph
First, we identify duplicate information, i.e., data that represents the same real-world entity. Duplicates are identified by the type identifiers. In author.xml, since the first and third book have the same title, and we have identifiers “book ← title;” and “title ← ; title”, the first and third book elements are duplicates. Similarly, in pub.xml, the second and third author elements represent the same author, because of “author ← name;” and “name ← ; name”. The duplicate information is removed through a process called node gluing. Gluing removes a duplicate, leaving only a single copy of the data. An example of gluing is shown in Figure 4. In the topmost tree an author element is duplicated. So one copy is removed. Note that its removal moves the edge from the book element to the remaining copy of the author element. Node gluing changes the tree to a partial order. The result is shown in Figure 5.

The next step is to add edges between “related” nodes. In author.xml, authors are related to the books they wrote, and also to the publishers that publish those books. A tree can only (directly) capture relationships between parent and child nodes. The proposed semantics represents every such relationship with an edge, hence creating a graph that will usually contain cycles. We call the process of relating nodes as node connecting.

Figure 6 illustrates node connecting with dashed lines. In the figure, only author, name and publisher nodes are connected. To reduce clutter in the depicted graphs we have chosen not to represent some of the relationships. A connection between the n2 name node and the t2 title node, for example, is not shown. This connection can be inferred because there is a one-to-one correspondence between book and title (as well as author
and name). Hence, any node connected to a name node is also connected to the corresponding (its parent) author node. How do we decide which types of nodes to connect and which not to? It depends on the possible parent-child relationships in forests to be semantically compared. For example, since author is always a parent of a name in any possible forest that we compare, we only connect author, but not name, with other types of nodes. On the other hand, since an author node can be either a parent or a child of a book node, we need to make connection between them in the graph. Not adding these edges keeps Figure 6 less cluttered. More importantly, it saves a certain amount of cost (depending on the property of the data) not to physically materialize these edges. Logically, however, these edges are present in the graph.

Note that the root node bib is, as we would infer, equally related to all nodes in each document. (Imagine that we could produce another semantically equivalent document that has two book nodes as the children of a root node bib, because the book nodes are both related to bib too.) For ease and clarity of presentation, we choose to remove this bib node in both graphs. The final graphs resulting from node connecting are shown in Figure 7.

The two graphs in Figure 7 are isomorphic. To illustrate this more clearly, a graph that is equal to both is depicted in Figure 8. It is semantically equivalent to the two initial trees because neither subtree gluing nor node connecting changes the “meaning” of the data. It is also a “minimal” form of the original trees in the sense that duplicate data has been eliminated. The graph in Figure 8 (as well as the graphs in Figure 7) is a canonical
representation of the two initial trees, because it is semantically equivalent to the original
data, yet syntactically minimal. The next task is to formalize the notion of a canonical
graph and how to construct it.

2.4.2 Description of the Translative Semantics

We have illustrated a translative semantics for XML by example. This section formalizes
this semantics for complete forests.

2.4.2.1 Node Gluing

Two nodes are glued together if and only if they are identical. That is, they are of the
same type and their identifiers evaluate to the same value. The gluing process is shown in
Figure 9.

**Input:** forest $F$

**Gluing:**

$G (V, E, \Sigma, L, C) := F$

for every pair of nodes $u, v \in V$, such that $u \neq v$

for every edge $(v, y) \in E$,

$E := E \setminus \{(v, y)\}$  // remove $(v, y)$ from $E$

$E := E \cup \{(u, y)\}$    // add $(u, y)$ to $E$

$V := V \setminus \{v\}$

update the functions $L$ and $C$ accordingly

**Output:** graph $G (V, E, \Sigma, L, C)$

**Figure 9 Node gluing**

The idea is that, if $u$ and $v$ are identical, then it is only necessary to keep one copy. We
can replace every edge $(v, y)$ with $(u, y)$ and then remove $v$. Adding new edges to the
forest may result in cycles. Thus gluing produces a graph in general. When nodes are glued in this process, the size of $V$ decreases, and the size of $E$ does not increase (and may decrease).

As we can see, semantically comparing two forests is only possible when given the set of identifiers for all types of nodes. Identifiers carry the information about how nodes are related, and are crucial to reason about data semantics.

2.4.2.2 Node Connecting

Related nodes, as we have defined earlier, are connected. Using the notations from above, the connecting process is described in Figure 10.

The idea is that every pair of related nodes is now explicitly identified by an edge that connects them. Before node connecting, a pair of related nodes may or may not be adjacent, while semantically whether they are connected or not should not make any difference.

**Input**: $F$ and $G (V, E, \Sigma, L, C)$

**Connecting**:

for every pair of nodes $u, v \in V$, such that

$u \in related(v, L(u)) \text{ in } F$ \hspace{1cm} // Note, in $F$, not $G$

if $(u, v) \notin E$

$E := E \cup \{ (u, v) \}$ \hspace{1cm} // add $(u, v)$ to $E$

**Output**: graph $G (V, E, \Sigma, L, C)$

**Figure 10 Node connecting**
Connecting effectively changed a tree to a graph by adding edges. The number of edges in $E$ may either increase or decrease depending on the specific situation. There is no change to $V$, $\Sigma$, $L$, or $C$.

### 2.5 Compaction

With a semantics for XML data defined, a formal discussion on compaction is now possible. Compaction aims to transform a forest to a smaller forest. The two forests will have the same translative semantics, i.e., the forest-to-graph translation maps them to the same canonical graph. The compacted forest may or may not have the same structure as the original data forest. *Grouping* is a compaction technique that does not change the structure, as discussed further in Section 2.5.2, while *restructuring* changes the structure, as discussed next.

#### 2.5.1 Restructuring

Compaction can be achieved by changing the structure of a forest. A *restructuring* is a transformation that changes the structure of the forest but keeps its semantics intact. A forest can be restructured by mapping it to its canonical graph, and then creating a semantically equivalent forest with a different structure. Ideally, the restructuring will yield a forest that is *smaller* in size than the original.

Consider the document, *author.xml*, shown in Figure 1. Observe that in the document each book contains a publisher element. But a publisher usually publishes many books.
This means that the information in the publisher element is *duplicated* in several book elements. If the document were restructured to that shown in Figure 2, with publisher elements above book elements, then the publisher information is *not* duplicated. Achieving compactness through restructuring exploits the idea that restructuring can remove duplicates.

A restructuring algorithm that transforms a canonical graph to a forest is shown in Appendix A. This algorithm takes a canonical graph \( G \) and a complete target signature as input, and outputs a new forest \( F \) that conforms to the signature. The algorithm builds the output forest \( F \) in a top-down fashion, guided by the target signature.

Essentially, this restructuring algorithm is an inverse of the combination of node gluing and node connecting. In changing the canonical graph to a forest, the restructuring algorithm *disconnects and unglues* nodes. It disconnects since the output must be acyclic, and it unglues (makes duplicates) since the semantics encoded in every edge in the canonical graph must be faithfully preserved. It is interesting to see that mapping a canonical graph back to a forest is substantially more complicated than computing the canonical graph from a forest, even though the graph and the forest entails the same semantics under a given set of identifiers. Also note that the third block may produce more clones of *parent* than necessary. That is, it ensures semantic equivalence between the output forest and the canonical graph at the cost of introducing redundancy in the forest. Fortunately, this can be taken care of by grouping – our second compaction technique presented in the next subsection.
In restructuring, different target signatures will yield forests of different sizes. To find the most compact forest among them, we could simply enumerate all the possible target forests. However, this is computational intractable. The number of different target signatures is more than exponential.²

While in general the problem is hard, there is a simple technique to generate a compact forest for some forests. The idea is to take advantage of the cardinality ratio. The ratio characterizes the relationship between pairs of element types as one of the following: one-to-one, one-to-many, or many-to-many. For example, the relationship between publisher and book is one-to-many: a book is published by exactly one publisher but a publisher publishes many books. On the other hand, the relationship between book and author is many-to-many.

Cardinality ratio may come with the data as a predefined constraint; if not, it can be quickly determined by traversing the canonical graph. (In contrast, we do not infer type identifiers and assume they must be given.) Table 1 shows the cardinality ratios for the example graph. The relationship between author and name is one-to-one (recall that authors are glued by name, hence each author is associated with a single name, and vice-versa). Author to book is many-to-many since an author can write many books, and a book can have many authors. (Note that exact, average ratios could be computed, e.g., 4.2 to 2.7.)

---

² Given a label set of size \( n \), suppose the number of distinct unordered trees is \( t(n) \) and the number of distinct unordered forests is \( f(n) \). We have,

\[
\begin{align*}
  t(1) &= 1, \quad t(n) = (2n-2) \cdot t(n-1) = (2n-4) \cdot (2n-2) \cdot \ldots \cdot 2 \cdot t(1) = (2n-2) \cdot (2n-4) \cdot \ldots \cdot 2, \\
  f(1) &= 1, \quad f(n) = (2n-1) \cdot f(n-1) = (2n-3) \cdot (2n-1) \cdot f(n-2) \cdot \ldots \cdot f(1) = (2n-1) \cdot (2n-3) \cdot \ldots \cdot 3.
\end{align*}
\]
The key to achieving compactness is to focus on types that are related in one-to-many relationships. Specifically, assume types \( X \) and \( Y \) are in a one-to-many relationship. Then a target signature that has \( X \) above \( Y \) leads to a forest that is more compact than a forest with \( Y \) above \( X \). Consider the example of publisher and book, which have a one-to-many relationship. If publishers are above books in the target signature, then in the target forest there are no duplicate publishers (or duplicate books). Every book is placed under the publisher to which it belongs. If books are above publishers in the target signature, then the same publisher may be duplicated several times. Such a forest is less compact and hence need not to be considered.

<table>
<thead>
<tr>
<th></th>
<th>pub</th>
<th>title</th>
<th>book</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>author</td>
<td>n-m</td>
<td>n-m</td>
<td>n-m</td>
<td>1-1</td>
</tr>
<tr>
<td>name</td>
<td>n-m</td>
<td>n-m</td>
<td>n-m</td>
<td></td>
</tr>
<tr>
<td>book</td>
<td>n-1</td>
<td></td>
<td>1-1</td>
<td></td>
</tr>
<tr>
<td>title</td>
<td>n-1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The technique for generating a target signature for a compact forest begins by considering one-to-one relationships. One side of the relationship is made a child of the other side. If one side is involved in gluing the other, then it should be made the child, otherwise either side can be made the child. Consider book and title in the example
graph. Their relationship is one-to-one. Furthermore, book is glued using title. Hence book should be a parent of title in the potentially compact output. The target signature after this step is book#title and author#name. Next, one-to-many relationships are processed by making the one side the parent. In the example, after considering the one-to-many relationships, the target is pub#book#title and author#name. Finally, only many-to-many relationships remain. The remaining types are placed as high as possible in the forest. In the example graph, this means that author is made a child of book resulting in the signature pub#book#(title,author#name).

Once the target signature has been generated, the original forest is restructured using the target signature. The restructured forest may or may not be smaller, i.e., the technique does not generate the \textit{most compact} forest. Finding the signature that leads to the most compact restructuring is theoretically intractable. However, we expect the technique outlined above to lead to a “reasonable” target signature in practice. The technique can be further refined by utilizing the average cardinality ratio of many-to-many relationships, e.g., if the ratio is thirty-to-two, then the two side of the relationship should be made the parent. But such refinements are beyond the scope of this chapter.

2.5.2 \textit{Grouping}

A second compaction technique is grouping. Grouping does not change the structure of a forest (i.e., the signature remains unchanged). Rather, it reduces the size of a forest by gluing information that is duplicated among siblings.
An example of grouping is shown in Figure 11. Suppose a store sells the same book at two different prices (a discount is given to book club members). When the book data is input into the store’s database, the data is entered as two separate book elements as shown in the left side of the figure. Yet, the data is about the same book, given the identifier “book ← title;”. Grouping merges the duplicate information, yielding a more compact data collection with the same translatable semantics, that is, the canonical graph for the data collection is the same before and after grouping. The book data can be grouped by title, yielding a single book element sold at two different prices as shown in the right side of the figure.

Figure 11 Grouping

More generally grouping can be described as follows. Assume $u$ and $v$ are siblings of type $t$. Let the forests rooted at the children of $u$ and $v$ be denoted as $F_u$ and $F_v$, respectively. $F_u$ and $F_v$ each consists of a collection of trees. Suppose $T$ is a forest consisting of trees commonly shared by $F_u$ and $F_v$ ($T \subseteq F_u \cap F_v$). Also suppose that $U = F_u - T$ and $V = F_v - T$. Then the grouped forest consists of either $u$ or $v$, one of the duplicated
identical forests $T$, and both $U$ and $V$. Hence, the grouping removes a copy of $T$ without semantic loss.

**Definition 10** [grouping] A grouping transformation is called a grouping of type $t$ nodes $u$ and $v$ by $\text{sig}(T)$. $\text{sig}(T)$ is a signature that is called the grouping pattern. □

There are some constraints on grouping. First, we caution that it is important that all identifying information for both $u$ and $v$ is contained in $T$. Otherwise, $u$ and $v$ would not be identical, and thus could not be glued at all. This is substantially different from the “group by” construct in SQL and XQuery, where “semantically different” records and trees, respectively, are grouped, and hence no “gluing” takes place. Because grouping in compaction effectively glues nodes, it has to ensure their semantic equality.

A second, more subtle condition concerns $U$ and $V$. Consider the scenario where $U$ consists of $x$ and $y$, and $V$ consists of $x'$ and $y'$, as shown in Figure 12. $x$ and $x'$ are of type $x$; $y$ and $y'$ are of type $y$.

![Figure 12 A possibly illegitimate grouping](image)

Without grouping, $x$ and $y'$ are not related. The reason is that these two nodes have a distance of at least four, while the minimum distance between a type $x$ node and a type $y$
node in the forest is two. According to the gluing criterion, they will not be connected in
the canonical graph. If we grouped \( u \) and \( v \), however, then \( x \) and \( y' \) would have a distance
of two. In this case they would be related nodes and hence a new edge between these two
nodes would be introduced in the canonical graph. The canonical graphs of the two
forests with and without the grouping would at least differ at this edge. Hence, grouping
would change the data semantics in this example. In general, the second criterion can be
described as follows. If trees in \( U \) are all of the same structure, grouping does not
introduce new edges in the canonical graph. (Since \( U \) and \( V \) have the same structure, trees
in \( V \) have the same structure too.) Such a grouping that does not affect semantics is called
a legitimate grouping. When \( U \) (\( V \)) contains trees with different structures, grouping
would change data semantics and is hence illegitimate. We refer to legitimate grouping
whenever using the term “grouping” alone.

A grouping always results in a forest equal to or smaller than the original forest. Hence
grouping is always “good” in terms of compactness. In addition, we can potentially do
grouping multiple times with different grouping patterns, on the same type. For the
sample data in Figure 12, for instance, there are two possible grouping patterns:

\((\text{sig}(T), x)\) and \((\text{sig}(T), y)\). (The pattern \((\text{sig}(T), x, y)\) is uninteresting, and the
pattern \(\text{sig}(T)\) is illegitimate, as we have shown.) There are two different orders in which
we can apply the two kinds of groupings. Interestingly, there are examples where doing
so may result in two different trees of different sizes, each of which cannot be further
compacted by grouping!
Again, determining the most compact grouping is intractable. As we have shown in the above, when there are $N$ possible grouping patterns for a type, each permutation of these $N$ patterns corresponds to a result. Each individual result may be different in size from the others. In general, finding the most compact grouping is NP-hard since there are $N!$ permutations.

Even though finding the most compact grouping is a hard problem, there is a simple technique to find a “good” grouping that will lead to a more compact forest. Just as in restructuring, the technique utilizes the cardinality of the relationships in the graph. Given siblings of $N$ types to group, we focus only on one-to-many relationships to generate a compact grouping pattern. For one-to-many relationships, we group first by the “one” side of the relationship and then by the “many” side. Consider if a book is sold at a single price, but different books can be sold at the same price. Then we should group by price, not title. For one-to-one and many-to-many relationships we group in any order; there is no a priori advantage to choosing a particular side.

2.5.3 Summary Restructuring and Grouping

Ideally, compaction will generate the smallest forest that represents the same information as a given forest $F$. We call such a forest the most compact forest of $F$. But, generating the most compact forest is intractable. We can however reason about the characteristics of the canonical graph to quickly find a more compact forest (if it exists) using two orthogonal techniques: restructuring and grouping. Restructuring changes the structure of
a forest to produce a smaller forest, while grouping retains the structure. Since the two
techniques are orthogonal, they could be combined to compact a forest, i.e., a forest could
be restructured and then grouped.

2.5.4 Compacting DBLP data

The effectiveness of compaction depends on the data collection. Some data collections
are compactable while others are not. To gauge how well compaction performs on real-
world data, we chose to compact DBLP data. DBLP provides bibliographic data
formatted in XML. The data has entries for conference papers, journal articles, book
chapters, books, technical reports, and theses. We extracted book data from the collection
for testing purposes, yielding a 309KB data collection that contains 7312 elements. All
the book entries have the same structure. Thus the data is complete. An example book
entry is shown in Figure 13. The signature of the data is

book#(author,editor,title,publisher,year,isbn).

<book mdate="2002-01-03" key="books/bc/MaierW88">
  <author>David Maier</author>
  <author>David Scott Warren</author>
  <title>Computing with Logic: Logic Programming with Prolog</title>
  <publisher>Benjamin/Cummings</publisher>
  <year>1988</year>
  <isbn>0-8053-6681-4</isbn>
</book>

**Figure 13 An example of the book data in the DBLP collection**
We decided to try compacting the DBLP book data by restructuring. It turns out that there are several one-to-many relationships in the data. For instance, a publisher can publish several books, but each book is published by a single publisher. There is also a one-to-many relationship between book and year. All other relationships are one-to-one, except that author is in a many-to-many relationship with every other type and publisher to year is many-to-many. Using the restructuring technique outlined in Section 2.5.1, the following “compact” signature would be generated

\[ \text{publisher#year#book#(author,editor,title,isbn)} \].

Restructuring the data using this signature yields a 252KB data collection that contains 5441 elements. The compacted data has an 18% reduction in file size and a 25% reduction in number of elements. The compaction reduced space because the original data replicated publisher information among all the books that were published by that publisher. Year data was also duplicated. In the restructured data, each publisher is represented only once and has among its descendents only those books that it publishes. (Note that we can recover the original data by restructuring the compact data using the original signature, that is, there is no data loss, though the specific ordering in the original data cannot be recovered.) Grouping would yield a similar reduction by grouping first by publisher, then by year (the others do not matter). So though the size of the reduction depends on the data collection, some real-world collections can be compacted.
2.6 Related Work

Compaction for XML is similar to XML compression in the sense that they both aim to describe the same information with shorter representation. However, compaction differs from usual compression since the output has to retain XML syntax. To the best our knowledge, we are not aware of any work directly related to the concept of compaction. In this section, we briefly review general and XML-specific compression techniques, and relate them to compaction when pertinent.

Most modern data compression techniques have their genesis in the Huffman algorithm [33] or the LZ77 algorithm [67]. Huffman coding is statistical; it assigns shorter codes to more frequent characters and longer codes to less frequent ones. Popular data compression tools such as gzip and pkzip are based on LZ77. A later version of LZ77, the LZW algorithm [55], is more suitable for practical implementation. The essence of the LZ77 family of compression techniques is to store repetitive sequences just once. Any repetition of a sequence that previously occurred is replaced by a pointer to that sequence. Such techniques are called pattern-based.

Specialized compressors take advantage of the specific properties of the data to be compressed. For XML data in particular, several compression techniques have been proposed. The earliest such work is XMill [38]. Incorporating existing compressors, XMill compresses XML structures and values separately, uses type specific compressors for different types of data, and allows user-defined compressors for domain specific data-
types. Data compressed by XMill cannot be directly queried; doing so would entail a complete decompression. XGrind [52] and XPRESS [45] are both compressors that support direct query evaluations on compressed XML data. XGrind uses a compression scheme based on Huffman coding, while XPRESS adopts an encoding method called reverse arithmetic encoding. It is worthwhile to note that both compression techniques are homomorphic because the structure of the original XML data is preserved in the compressed XML data. In contrast, an important compaction technique proposed in this chapter, restructuring, always changes the structure of the original data. Homomorphism is important for the compressed data to be efficiently queried. Compaction, on the other hand, is useful to ascertain the semantic redundancy in the data. Compacted XML data is not supposed to be queried directly by the query intended for the original data. Among the three compressors, XGrind is the only one that tries to utilize schema information such as a DTD to enhance the compression ratio. In comparison, finding an appropriate schema (target signature in our situation) is the goal of compaction. To enhance compaction ratio, knowledge of identifiers in the original data is required, and knowledge of cardinality ratio is helpful.

2.7 Conclusion

XML compaction aims to produce a smaller, compact XML forest, without losing information. This chapter develops a formal framework for the compaction of XML data. It first formalizes XML data by introducing a forest data model and defining types and identifiers. A translative semantics for XML is then presented. This semantics translates
an XML data collection to a canonical graph, depending on a given set of identifiers. Data collections that translate to the same canonical graph are deemed to have the same semantics. Based on this formalization, two compaction techniques, restructuring and grouping, are discussed. Though finding the most compact forest is computationally prohibitive in general, we developed simple techniques to find a more compact forest at low cost using restructuring or grouping.

In future we plan to explore the relationship between compaction and compression. General compression techniques are not confined to produce the same file format as the input. Hence, it is reasonable to expect that they can achieve a better compression than compaction. However, a file can be first compacted and then compressed. Does combining the compaction with compression produce better performance than compression alone? An interesting work is to examine this problem on both the theoretical and experimental grounds.
3. POLYMORPHIC XML RESTRUCTURING

It has become increasingly important to be able to query and restructure XML with limited knowledge of its original structure. Such a polymorphic capability is both convenient for a naive user and necessary when an expert queries a heterogeneous or evolving data collection. The least common ancestor (LCA) has been previously shown to be useful to achieving polymorphism. The LCA can be used to find related nodes in an instance of an XML data model, independent of its structure. This chapter refines the use of the LCA to restructure XML data in such a way that information is neither lost nor created during the restructuring. Our restructuring operation uses a simple, declarative specification of the target, called a signature. This chapter also sketches a join technique that can be used to efficiently restructure XML data.

3.1 Introduction

Queries over XML data usually depend on detailed knowledge of the hierarchy structure of the data. Users commonly know what kind of data is stored in an XML data collection, but have less precise knowledge of its structure. A polymorphic operation is an operation that does not depend on the structure of the data. The benefits of polymorphism are manifold. First, queries that depend on specific structure often need to change when the structure varies. In contrast, a polymorphic operator can accommodate a wide variety of different structures. Second, heterogeneous data is common in XML, especially when data is acquired from diverse sources. A polymorphic operator works over all the differently structured sources. Third, since a polymorphic operator does not require prior
knowledge of the structure, it is easier to use for a user with limited expertise in a query language such as XQuery or XSLT.

It is the use of path expressions that ties an XQuery query or XSLT program to a specific structure. Each path expression locates data in an instance of an XML data model, and may break if the same data were structured differently. Therefore the central issue for the realization of polymorphism is a proper substitute for path expressions. This substitute needs to be (1) free of structure-dependent expressions, and (2) capable of conveying common query semantics.

The least common ancestor (LCA) has been shown by various studies to be a good candidate (e.g., [16] [37]). It is semantically expressive in identifying related information in hierarchical data, independent of the specific structure of the hierarchy. The basic assumption behind the use of LCA is that the semantic relationships among a set of nodes can be captured by their LCA.

Let’s look at an example that sketches an LCA-based approach to querying data, which modifies the example given in [37]. Figure 14 shows two different structures for bibliographic data. In books.xml, the data is structured as a list of books, where each book contains the information related to that book. In contrast, publishers.xml organizes the data differently. Each book is listed below its publisher. Though both documents have the same data, they have very different structures. The figure also lists two queries on the bibliographic data. Query 1 retrieves the publishers of books authored by “Codd”.

43
An LCA-based strategy can be used to answer the query as follows. First, find each “Codd” author node. Second, pair the node with every publisher node. Third, find the LCA of each pair. Fourth, and finally, only keep those publisher nodes that have an LCA which is not an ancestor of any other LCA (of the “Codd” author node and a publisher node). In books.xml, for example, node 5 is paired with publisher nodes 4, 8, 12, and 16. The LCAs for the four pairs are respectively 2, 1, 1, and 1; but only node 2 is not an ancestor of some other LCA. Hence the query result contains only node 4, which is the publisher Addison Wesley. This strategy is independent of the specific structure of books.xml.
For publishers.xml, the LCA-strategy yields the “same” result. First, the “Codd”
author node, node 5, is found. Next, it is paired with publisher nodes 2, 9, and 13. The
LCA of each pair is 2, 1, and 1, respectively. Only the first pair is kept, yielding Addison
Wesley, which is the sole publisher of a Codd book.

The advantage of the LCA-based approach is that it is independent of the structure: the
same operation will work on both structures. This structural independence is a property
that we call polymorphism. In contrast, consider how the desired nodes would be located
using a traditional approach, e.g., XPath. Since the relative positions of an author and its
corresponding publisher differ in the two documents, a different path expression is
needed for each structure: “//author[text()="Codd"]/../publisher”, and
“//author[text()="Codd"]/..” for books.xml and publishers.xml, respectively.

The disadvantage of the LCA technique is that it sometimes incorrectly relates nodes.
Consider Query 2, which asks for the author of each title. For books.xml, the LCA-
strategy relates author node 5 to title node 3, author node 13 to title node 11, and author node
17 to title node 15. However, it would also incorrectly relate author nodes 5, 13 and 17 to
title node 7! More correctly, no author should be related to title node 7. Similarly, the LCA-
strategy fails on publishers.xml as well. It gives a correct result for title nodes 4, 11
and 15, but for title node 7 it incorrectly returns author node 5, which should only be
related to title node 4.
This example representatively illustrates the potential pros and cons of an LCA-based approach. While the approach works for many queries (such as Query 1) it is incorrect for others. The problem is that current LCA-based techniques tend to be too inclusive as illustrated by Query 2 on the example data.

Problems with correctness are a serious drawback to LCA-based approaches gaining wider acceptance. Traditionally in database applications, one would not trade correctness of the query semantics for simplicity and convenience in query formulation. One workaround is to use LCA-based techniques in ranking query results. For instance, if one could differentiate that in an answer to Query 2 on publishers.xml, author node 5 is more related to title node 4 than title node 7 then the results could be ranked by relevance. Query ranking systems are important in information retrieval and knowledge discovery applications.

This chapter makes the following contributions. First, we develop a refined LCA-based operator, called the Closest relation, that locates related nodes. We show that this operator is useful for querying XML data in general and restructuring in particular. We also show how to integrate it into an XML query language such as XQuery.

A second contribution is that we examine XML restructuring in detail. A common XML querying task, restructuring transforms a data collection to a new structure. Programming a restructuring transformation in XQuery or XSLT is currently the most widely-used solution in the database community. Using the proposed Closest operator, we take a different approach to restructuring. Namely, we propose a polymorphic XML
restructuring algorithm called the *poly-transform*. Unlike queries in XQuery or XSLT, the poly-transform is independent of the source structure; all that it needs is a *declarative* specification of the target. The specification is called a *signature* (introduced Section 2.3.2). For example, the structural signature for publishers.xml is

\[
\text{bibliography}\#\text{publisher}\#\text{book}\#(title,author,editor)
\]

With such a structural signature, the poly-transform restructures books.xml to publishers.xml with the following poly-transform enabled XQuery query.

```xml
for $root in //bibliography
  return poly-transform($root, bibliography#publisher#book#(title,author,editor))
```

Alternatively, a query can be turned into a polymorphic query by first transforming the input data to the desired structure.

```xml
for $root in poly-transform(doc("foo.xml")/*,
  bibliography#publisher#book#(title,author,editor))
  ---rest of query using structure of publisher.xml---
```

Note that foo.xml could be any kind of bibliographic database that uses the same ontological terms as publisher.xml (mapping of ontological terms is considered an orthogonal issue, our concern is only with the structure). Since the full transformation is potentially expensive for a large data collection the technique can be folded into a path expression as follows.

```xml
for $codd in title[#author="Codd"]
  ---rest of query ---
```
The “#” step finds the author nodes that are related to a title.

In the example above, the data in books.xml can also be recovered by applying the poly-transform to publishers.xml with bibliography#book#:title,publisher,author,editor as the target structural signature. The fact that either data collection can be recovered from the other suggests that the mapping is reversible. Intuitively, reversibility is what constitutes a “good” restructuring. We define reversibility as being (1) inclusive, and (2) non-additive. This formalization establishes the correctness of the poly-transform semantics. This is the third contribution.

The fourth and final contribution is that prove that the proposed poly-transform is inclusive. By examining the structural signatures of the source and target, we can further determine whether it is possible for a restructuring to be non-additive. Therefore, we can always tell whether a poly-transform produces reversible mappings.

Before we conclude this section, we emphasize that the polymorphic paradigm is not meant to replace XSLT or XQuery. Rather, our aim is to extend these languages with an additional capability. Any restructuring task that can be done by the poly-transform can be done in XQuery (though different XQuery queries are usually needed for each different source structure). It is in such “niche market” that polymorphic querying in general and polymorphic restructuring in particular becomes handy. While a user can employ a polymorphic query construct for simplicity and reusability, she still has the

---

3 The data in the recovered version of books.xml is the “same” modulo ordering and duplicates. The data may be in a different order than the original, and may contain duplicates unless duplicate-elimination is turned on in the poly-transform.
option to use XQuery or XSLT for data whose structure is known and is not expected to change, or in situations where the expressiveness of XQuery or XSLT is necessary for the task.

This chapter is organized as follows. Section 3.2 discusses related work. Section 3.3 presents definitions and notations. The details of the poly-transform are given in Section 3.4. The section also describes some interesting properties of the poly-transform and the duplicate-eliminating poly-transform, and its integration into XQuery. We have implemented the poly-transform in Java. Section 3.5 describes the result of experiments to measure the performance of the implementation. Section 3.6 concludes the chapter with a summary and discussion of future work.

3.2 Related Work

We discuss work in several different areas related to this chapter.

3.2.1 LCA-based Querying Techniques

The LCA has been shown by various researchers to be effective at identifying related information in hierarchical data. Previous work falls into two categories: keyword-based search (e.g., [16] [29] [63]) and structured declarative querying. (e.g., [37] [48]). In keyword-based search, users describe the data of interest by supplying keywords. LCA can be employed to rank results by relevance. In structured declarative querying, a full query language such as XQuery is augmented with an LCA-based query construct. A structured declarative query language has precise and more involved query semantics.
Existing LCA-based techniques have the drawback of being too inclusive – query answers often contain invalid results alongside correct ones. We discuss three most relevant papers, with each improves on the preceding (less inclined to produce incorrect results).

Schmidt et al. propose a *Meet* operator [48], which is equivalent to LCA. The Meet operator suffers from the problem shown by Query 2.

Cohen et al. propose an *Interconnection* relationship that more appropriately decides whether nodes are semantically related [16]. Two nodes are interconnected if and only if the path between them does not contain two distinct nodes with the same tag (label). The Interconnection relationship avoids the problem that *Meet* has with Query 2. In both books.xml and publishers.xml, a path between a title with no author and any author (which belong to a different title) includes two book nodes. Thus the interconnection relationship correctly identifies that such a title is not related to any author.

![Figure 15 A document with same-label nodes at different levels](image-url)
Li et al. [37] give an example in which the Interconnection relationship produces incorrect result. The example (Figure 4 in [37]) is reproduced here in Figure 15. For Query 2, both author nodes would be returned for either title node according to the Interconnection semantics. To avoid this, a Meaningful Lowest Common Ancestor Structure (MLCAS) operator is introduced to facilitate what is called “schema-free XQuery”. However, MLCAS still produces incorrect result. For example, if title node 10 is removed from Figure 15, then both author nodes would be related to title node 2 by MLCAS. As another example, MLCAS would incorrectly treat author node 5 and editor node 8 in publishers.xml as meaningfully related, despite the fact that they belong to two distinct books. Finally, although the MLCAS operator itself is polymorphic, queries equipped with MLCAS may not be so.

3.2.2 Restructuring

Restructuring is a part of every general-purpose XML query language. Indeed, any restructuring can be specified in XSLT and XQuery. However, beyond expressiveness, there are issues such as simplicity and reusability. Correctly programming a restructuring has been recognized as a non-trivial task [23] [35] [47]. Pankowski [47] and Krishanmurthi et al. [35] proposed descriptive languages specifically for specifying XML data transformations. These special-purpose languages hide from users much of the procedural specification necessary in XQuery and XSLT, just like the poly-transform proposed in this chapter. Unlike the poly-transform these languages are not polymorphic and hence not reusable. Instead the transformations depend on the source structure and
must be rewritten for different sources. Thus, programmer effort generally grows in proportion to both the number of sources and the number of targets.

Restructuring is not limited to XML or hierarchical data. In a broader sense, restructuring can be viewed as a problem related to data integration. YAT [12] and SilkRoute [24] translate relational data into XML. Abiteboul et al. [1] and McBrien and Poulouvassilis [44] propose to use a general framework that allows translation between more heterogeneous data models. Both define a set of primitive operations in a common data model that can express restructuring transformations in higher-level data models. The common data model is an ordered labeled tree in [1] and a hypergraph-based data model in [44]. The former describes a correspondence between an OODB and SGML as an application; the latter focuses on restructuring transformations between structured data and XML, and ER and XML representations.

3.2.3 Other Related Work

Erwig [23] discusses the notion of information preservation, but only limited to basic transformation operations such as renaming, deletion and grouping. Information preservation is formally defined by Bohannon et al. [5] as invertibility and query preservation, in the context of schema embedding. Invertibility is similar to our notion of reversibility; but they consider query preservation in the context of schema embedding, in which an incomplete schema is embedded in a more complete schema (a limited form of polymorphism).
Tree pattern relaxation (e.g., [2] [3]) is also related to polymorphic XML querying. Query relaxation aims to provide flexible XML querying by combining both tree structure and keyword search. Such a query uses a “template” structure to specify the context for a keyword search. Relaxed queries and polymorphic queries are useful complements. A relaxed pattern query allows polymorphism on the insignificant structural aspect yet at the same time prescribes what is essential to the structure of query result.

3.3 Preliminaries

This section defines preliminary concepts that are used in this chapter.

3.3.1 Tree, Forest and Node Type

We model XML documents as labeled trees. Each tree node represents an element and is labeled with its tag; each tree edge represents a parent-child relationship between nodes. An XML data collection consists of multiple documents or fragments of documents. It is not necessarily a tree, and thus modeled as a forest of labeled trees. In a forest, two nodes are connected if and only if they are located in the same tree. We use $V$ in this chapter to denote the set of nodes in a forest, $E$ to denote the set of edges, and $L$ to denote the function that maps each node to its label. Thus, a forest $F$ is represented as a three-tuple $(V, E, L)$.

**Definition 11** [node type] The root-to-node path type (node type in short) of a node $v$ is the list of node labels on the path from the root to $v$ (inclusive). Denote the type of $v$ to be
\( T(v) = x_1/\cdots/x_k, k \geq 1, \) where \( x_1, \ldots, x_k \) are the labels on the path from the root to \( v \) (i.e., \( x_1 \) is the label of the document root, \( \ldots \), and \( x_k \) is the label of \( v \)). \( T \) is called the type function. □

The node type specifies a type for each node, differentiating among nodes with the same label by taking into account the labels of their ancestors. The node type can be computed even when a schema is absent. If a schema is present, the type of each node could instead be taken from the schema. The type induced by the node type would often coincide with the typing designated by a schema in the form of DTD or XML Schema.\(^4\)

It happens that in both books.xml and publishers.xml in Figure 14, nodes with the same label are of the same type. For example, the type of all author nodes in books.xml is bibliography/book/author. For brevity, we will use the label to denote the type whenever there is no confusion. So in books.xml, the type of all author nodes is author. Otherwise, we have to use the node type to distinguish nodes with the same labels. In Figure 15, for example, both node 2 and node 10 have the label title; but their node types are different: article/title for the former and article/content/reference/ref/title for the latter.

3.3.2 Signature

Signatures, define earlier by Definition 4 on page 13, will be used to specify the structure of a forest. We go over the concept of signature using books.xml as an example. In

---

\(^4\) A most notable exception is data with a recursive schema. However, as we do not assume knowledge of schema, one is not able to tell from the data instance whether it conforms to a recursive schema or a non-recursive schema. Thus we do not consider recursion in this chapter.
books.xml, there are four book subtrees with three kinds of structures. The first and third book subtrees have the same signature $s_1 = \text{book#}(\text{author}, \text{publisher}, \text{title})$; the second and forth book subtrees have distinct structures, whose signatures are $s_2 = \text{book#}(\text{author}, \text{publisher}, \text{title})$ and $s_3 = \text{book#}(\text{editor}, \text{publisher}, \text{title})$, respectively. So the signature of books.xml is bibliography#($s_1, s_2, s_3$).

We further define a signature to be composed of terms. A label is called a simple term, while a list of signatures is called a complex term. The signature of books.xml has two terms. The first is a simple term: bibliography, and the second is a complex term that contains three signatures $s_1$, $s_2$, and $s_3$.

3.3.3 Forest Restructuring

With forest and signature defined, we now introduce forest restructuring.

**Definition 12** [forest transform] A forest restructuring transform (transform in short) is a function $\text{trans}(F, s) = (F_s, A)$, where the input is a forest, $F$, and a target signature, $s$, and the output contains a forest, $F_s = (V_s, E_s, L_s)$ whose signature is $s$, and a relation $A: V \times V_s$, called the association relation. □

**Definition 13** [forest restructuring] Following the notations in the previous definition, a specific execution of a restructuring transform, $\text{trans}(F, s)$, is called a forest restructuring (restructuring in short). □
A transform is a function while a restructuring is a specific execution that depends on a source forest and a target signature. Note that a restructuring generates not only a target, but also an association relation $A$ that associates nodes in the source to those in the target. Associations are an important part of a restructuring. Repeated restructurings create a *composed association*.

**Definition 14** [composed association] A *composed association* $C$ of associations $A: V_1 \times V_2$ and $B: V_2 \times V_3$ is the composition of relations $A$ and $B$. Specifically, $(v_1, v_3) \in C$ if and only if there exists $v_2 \in V_2$ such that $(v_1, v_2) \in A$ and $(v_2, v_3) \in B$. □

Composed association is employed in determining whether a restructuring has desirable properties. Suppose $s$ is the signature of forest $F$ and $x$ is a signature that has only the labels in $s$. Then consider the following transformation: $\text{trans} (\text{trans}(F, x), s) = F_s$ which restructures $F$ to some forest with the signature $x$, and then to a forest, $F_s$, that has $F'$'s signature. Since nodes in the source and target forests are related through an association relation, we can easily determine whether $F$ and $F_s$ are the “same graph” (i.e., isomorphic). Therefore, it is easy to decide whether the restructuring adds or loses data.

**Definition 15** [forest inclusion] Let $F = (V, E, L)$ and $F' = (V', E', L')$ be two forests with association $A: V \times V'$. $F'$ is said to include $F$ under association $A$, denoted $F \subseteq_A F'$, if and only if $\forall (u, v) \in E$, $\exists (u', v') \in E'$ such that $L(u) = L'(u')$, $L(v) = L'(v')$, and $(u, u'), (v, v') \in A$. □
Definition 16 [forest equivalence] Let $F$ and $F'$ be two forests with association $A$. Then $F'$ is said to be equivalent to $F$ under association $A$, denoted $F =_A F'$, if and only if $F \subseteq A F'$ and $F' \subseteq A F$. 

With inclusion and equivalence defined, we can formally discuss the desirable properties of a transform function.

Definition 17 [transform properties] Suppose that $\text{trans}(F, x) = (F_x, A)$ and $\text{trans}(F_x, s) = (F_s, B)$, where $s$ is the signature of $F$. Also suppose that $C$ is the composed association of $A$ and $B$. Then $\text{trans}(F, x)$ is

- *non-additive* if $F_s \subseteq C F$, and *additive* otherwise,
- *inclusive* if $F \subseteq C F_x$, and *non-inclusive* otherwise, or
- *reversible* if it is both inclusive and non-additive (i.e., $F =_C F_x$), and *irreversible* otherwise. 

A *reversible* restructuring neither manufactures nor loses data. It formalizes the notion of what constitutes a “good” restructuring. After the poly-transform is presented, we will examine if it possesses the above properties.

3.3.4 Closest Relation

Recall that in Query 1 in the motivating example (Figure 14), we need to find the publisher for the author “Codd”. Though the two sample documents have the same information, they have different structures. However, we can observe that in both
structures the same relationship holds between a result node (an “Addison Wesley” publisher node) and a context node (a “Codd” author node): they have the least possible distance of any pair of author and publisher nodes. Such pairs are called closest nodes.

As an example, in books.xml the publisher node 4 is closest to the “Codd” author node 5, while in publishers.xml the publisher node 2 is closest to the “Codd” author node 5.

The semantics of Query 1 is to find the publisher that is related to a specific author; the above observation suggests that we can translate related-ness to closeness.

**Definition 18** [distance] The distance between two nodes $u$ and $v$, denoted $\text{dist}(u, v)$, is the number of edges on the path between $u$ and $v$ if they are connected, or infinity otherwise. $\square$

**Definition 19** [Closest relation] Suppose $V$ is the set of nodes of forest $F$ and $a, b$ are node types. Denote $\text{minTD}(a, b)$ to be the minimum type distance of $a$ and $b$:

$$
\text{minTD}(a, b) = \min\{\text{dist}(u, v) \mid T(u) = a \land T(v) = b, u, v \in V\}.
$$

Then

$$
\text{Closest} = \{(u, v) \mid u, v \in V \land \text{dist}(u, v) = \text{minTD}(T(u), T(v))\}. \quad \square
$$

$u$ and $v$ are closest if no two other nodes with the same types are closer than $u$ and $v$ are. For example, Figure 16 diagrams the all the nodes closest to the node 3 in books.xml.
(only the first two book subtrees are shown for brevity). There are four closest nodes to node 3; each is connected to it with a dashed arrow. Node 3 is not connected to node 9, because they have a distance of four while $\min_{TD}(\text{title}, \text{editor})$ is two. Similarly, using the $\text{Closest}$ relation can correctly answer Query 2; it returns no author for title node 7.

![Figure 16 Nodes closest to node 3 in books.xml](image)

(only the first two book subtrees are shown)

### 3.3.5 Preserving Closeness

We have now formalized the notion of “closest” and axiomatized its relation to common query semantics. Next, we examine its particular implication in restructuring.

The $\text{Closest}$ relation records which pairs of nodes are closest in books.xml and publishers.xml. We observe that whenever two nodes are closest in books.xml so are their counterparts in publishers.xml. This can be verified for every pair of closest nodes. In books.xml, for example, “Database” title node 3 and “Addison Wesley”
publisher node 4 have a distance of two which is equal to $\min TD(\text{title}, \text{publisher})$. Their counterparts in $\text{publishers.xml}$ are respectively node 4 and node 2, which have a distance equal to $\min TD(\text{title}, \text{publisher})$ in $\text{publishers.xml}$. So though the relative positions between a title and its related publisher differ in the two documents the Closest relation captures which pairs of nodes are related in each document.

If the closeness of every pair of nodes in the source forest is preserved in the target forest after a transformation, the transformation is considered to have included all information in the source. Consider in $\text{books.xml}$ the “Database” title node 3 and the “Addison Wesley” publisher node 4, which are closest to each other. If during a transformation that title become lost or closest to a different publisher, then this particular title to publisher correspondence is lost in the target, and possibly a non-existent correspondence is manufactured.

We showed that a closeness preserving restructuring can be reversible. Here we first formalize the notion of preserving closeness as follows.

**Definition 20** [closeness preserving] Suppose $V$ and $V'$ are node sets of two forests, and $A : V \times V'$ is a set of node associations. $A$ is said to preserve closeness if, $\forall p, q \in V$ such that $\text{Closest}(p, q), \exists p', q' \in V'$, where $(p, p'), (q, q') \in A$, and $\text{Closest}(p', q')$. □

For convenience, we will sometimes say that a restructuring, rather than the association relation it generates, preserves closeness.
3.4 The Poly-transform

We now present a forest restructuring transform that we call the poly-transform. The poly-transform is simply defined as a closeness preserving transform. This section discusses it in detail. Section 3.4.1 presents an algorithm that implements the poly-transform and Section 3.4.2 evaluates the implementation complexity. Section 3.4.3 then examines the reversibility-related properties of the poly-transform. Section 3.4.4 discusses its integration into XQuery.

**poly-transform:**

Input: (1) a forest $F=(V, E, L)$
(2) a signature $\text{sig}$
(3) $\text{roots} = \{ r \mid r \in V \text{ and } L(r) = \text{rootLabel} \}$

Transformation: $\text{polyTRANSFORM}(\text{roots}, \text{suffix(\text{sig})})$

Output: a forest $F'=(V', E', L')$ and a set of associations $A$

$
\text{clone}(c):$
\text{return a node that has the same label and text content as } c
$

\text{polyTRANSFORM}(\text{parents, sig}):$
\text{return if sig is empty}
\text{term := prefix(sig)}
\text{if term is a just a label}
\text{for each node } p \text{ in parents}
\text{p' := clone(p), add (p, p') to A, add p' to V'}$

// Find closest nodes of label term
\text{children := [clone(c) | c \in V, L(c) = term and Closest(c, p)]}
\text{for each node } c' \text{ in children}
\text{add (p', c') to E', add (c, c') to A, add c' to V', update } L'$

// Do the next label
\text{polyTRANSFORM(children, suffix(sig))}
\text{else}

// term is a complex term
\text{for each } s \text{ in term}
\text{polyTRANSFORM(parents, s)}$

Figure 17 The poly-transform algorithm
3.4.1 The Poly-transform Algorithm

Figure 17 depicts an algorithm that implements the poly-transform. Given a forest $F$, it produces a forest $F'$ that conforms to the given signature $\text{sig}$. The forest $F'$ is built in a top-down fashion. The algorithm also outputs the association relation $A$. Using $\text{polytrans}()$ to denote the poly-transform function, we have $\text{polytrans}(F, \text{sig}) = (F', A)$.

As an example, suppose we transform $\text{books.xml}$ using

\[
\text{bibliography}\#\text{publisher}\#\text{book}#(\text{title,author,editor})
\]

as the target signature. $\text{polyTRANSFORM}$ is called with the first term in the signature, bibliography, as the root label. A list of one bibliography node is created. The next term in the signature is publisher. A list of four publisher nodes is created. Since they are all closest to the bibliography node in $\text{books.xml}$, each is added as a child of the bibliography node. The next term is book. For each publisher node in $\text{books.xml}$ we find the closest book node(s), which is each publisher’s parent. So in the target each publisher will have one book child. The last term (title,author,editor) is processed by recursively invoking the transformation for each kind of child with book as the parent label. Each recursive invocation fleshes out a book element with additional children. The closest title to each book is added as a child of the book, then the closest author (if present) and finally the closest editor (if present). Some books do not have a closest author or editor. The transformation ends when the last term is processed. The association relation $A$ is updated.
each time a new node is created. Figure 18 shows the final result, where the dashed lines indicate that no such closest node is found.

Duplicate elimination is an additional step in the algorithm in the creation of the \textit{children} list. Only a child with a different value (in the source) can be added to the list. Furthermore, when a duplicate is detected, the original inherits all of the closest relationships of the duplicate. So in the poly-transform with duplicate elimination, the first step will create three publisher nodes, rather than four. But the publisher node representing Addison Wesley will be closest to all of the nodes represented by both Addison Wesley publishers in the source. Duplicate elimination permutes the order of the data, so the poly-transform with duplicate elimination cannot ensure that the ordering of the source data is maintained in the target.
3.4.2 Implementation Complexity

The most important and potentially costly step in the poly-transform algorithm is to find closest pairs of nodes of particular labels. However, it turns out that this step can be implemented efficiently as a single join operation, which we call an LCA-join. First, each node in the source forest is numbered according to the lexical ordering of the elements; the two trees in Figure 14 are numbered in such a way. Each node also keeps the number of its maximum descendent. So for example node 2 in books.xml keeps a pair of numbers (2, 5). Ancestor/descendent relationships can be determined by reasoning about the number pairs: all nodes with a number larger than the number of a node \( v \) and no larger than its maximum descendent’s (the descendent of \( v \) last reached in the preorder traversal) are descendents of \( v \). For example, nodes 3, 4 and 5 are all the descendents of node 2 in books.xml. The node numbering can be done with one preorder traversal of the source forest.

Next, a list of nodes is created for each type in the signature. Then closest pairs can be computed in a single LCA-join by simply merging three lists as depicted in Figure 19. In the figure, there are three lists of nodes: parents, children, and least common ancestors (lca in short). The parents list is nodes with the previous type in the target signature. The children list is nodes for the current type in the target signature (so children of the current type are being added to parents of the previous type). The lca list is the list corresponding to the type that is the least common ancestor of the child and parent types in the source signature. For instance, if title children are being added to publisher parents from the
source in books.xml, then the lca type is book. The lists are merged in the direction of a lexical ordering of the data (from left to right in the figure). A parent is closest to a child if both are descendent of the same lca. If a parent is not a descendent of the current lca, then either the current lca is before the current parent (child), in which case the current lca pointer is advanced, or the current parent (child) is before the current lca, in which case the current parent (child) pointer is advanced. Typically only two lists are merged instead of three since the parent or child is the lca.

As an example consider Figure 20. It illustrates the task of placing <title> elements below <publisher> elements in the restructuring of books.xml to the signature publisher#title.

The lca is book. The merging process starts with the pointers at the start of each list. The first publisher and title are both descendents (within the range) of the first book, so the publisher is a parent of the title in the output. The next publisher however is not within the range of the current lca hence it is not closest to the first title. The merging process continues by advancing the lca and child pointers to find the next Closest pair.
Since a single label may correspond to several types, an LCA-join is needed between each pair of types corresponding to the closest types of the source and target labels. For example, suppose that the signature publisher#title is used for a transformation. Suppose further than there are two types of publisher nodes (e.g., bibliography/publisher and bibliography/book/publisher), and three types of title nodes. Then each type of publisher node joins with each type of title node that is closest, that is, that has the minimum type dist (more than one type may be equally close).

The LCA-join operation to find closest nodes is of special importance in database management systems. If an XML DBMS can iterate through elements of a particular type, then the poly-transform can be supported with little overhead. Such indexes are commonly available in native XML DBMSs (e.g., Xindice, eXist, and BerkeleyDB-XML provide element type indexes.)

One list merge is needed for every label in the signature. Since the time complexity of a single list merge is \( O(n) \), where \( n \) is the number of nodes, the time complexity of the
poly-transform is \( O(ns) \) where \( s \) is the number of labels in the signature and \( n \) is the number of nodes.

3.4.3 Reversibility

Reversibility is considered the characterization of a “good” restructuring. A restructuring is reversible if and only if it is inclusive and non-additive. In this subsection we examine whether our implementation of the poly-transform (Figure 17) possesses these properties. We first discuss an assumption that simplifies our presentation. We then prove that the poly-transform is always inclusive. However, the poly-transform is not always non-additive; we give the condition under which it is non-additive (and hence reversible because it is already inclusive).

3.4.3.1 Complete Signature and Forest

For simplification of presentation, we assume that the forests and signatures involved in this subsection are all complete, as defined in Definition 5 on page 15. Recall that a signature is complete if it does not contain two occurrences of the same type; a complete forest is a forest whose signature is complete.

For example, books.xml is not complete as its signature is shown in Section 3.3.2 to be bibliography\((s_1, s_2, s_3)\), in which \( s_1, s_2 \) and \( s_3 \) each contains an occurrence of the book type. On the other hand, the signature used to restructure books.xml in Figure 18 is

\[
\text{bibliography}\#\text{publisher}\#\text{book}\#(\text{title,author,editor})
\]
is complete, because each type in it only occurs once.

We can assume that the source forest and target signature are complete because of the following two facts.

(1) Incomplete forests and signatures can be modified so that they are complete. First, if the source is not complete, we can “pad out” the forest to make it complete. For example, Figure 18 shows an incomplete forest; but if we realize the dashed edges and manufacture the three non-existing nodes, the forest becomes complete. Second, suppose we have an incomplete target signature, say, bibliography#(s₁, s₂, s₃) as shown in Section 3.3.2. We can easily merge the three terms s₁, s₂ and s₃ so that the signature becomes complete:

\[
\text{bibliography}\#\text{book}\#(\text{title,publisher,author,editor})
\]

(2) Such modifications do not affect our reasoning on a restructuring. The example in Section 3.4.1 uses books.xml as the source (Figure 18), which is not complete. We have seen that whenever a closest author or editor node is not found for a book node in the source, the corresponding book node in the target would just have no such author or editor child. Similarly, the target signature used in Figure 18 can be regarded as the result of modifying the signature of the resulting forest shown in Figure 18.

3.4.3.2 Reversibility-related Properties

We assume that the inputs to a poly-transform, a source forest and a target signature, are both complete. We make this assumption because in a complete forest, every node has a
minimum label distance node of any given label. The following lemma formally spells out this property.

**Lemma 1** Suppose $F = (V, E, L)$ is a complete forest, $u \in V$, and $L(u) = x$. Then for a label $y, y \neq x, \exists v \in V$, such that $\text{Closest}(u, v)$ and $L(v) = y$. □

From how the poly-transform is implemented, we also have Lemma 2.

**Lemma 2** Following the above notations, if $p_2$ is an ancestor of $q_2$ in $T_2$, then $p_1$ and $q_1$ are a closest $p$-$q$ pair in $T_1$. □

With the help of these two lemmas, we can show that the poly-transform preserves closeness.

**Theorem 1** Given a complete forest and a complete signature, the poly-transform preserves closeness. □

Suppose $F_1 = (V_1, E_1, L_1)$ and $F_2 = (V_2, E_2, L_2)$ are the source and target in a poly-transform restructuring, respectively. $F_1$ and $F_2$ are both complete. $A$ is the association relation. $(p_1, p_2), (q_1, q_2) \in A$, where $p_1 \neq q_1, L_1(p_1) = L_2(p_2) = p$ and $L_1(q_1) = L_2(q_2) = q$. To show $A$ preserves closeness, we need to prove that when $(p_1, q_1)$ is a closest pair, so is $(p_2, q_2)$.

**Proof** [Theorem 1] Suppose $lca(u, v)$ is a function that returns the lca of two vertices $u$ and $v$. In $F_2$, consider $lca(p_2, q_2)$. There are two possibilities:
1. $lca(p_2, q_2) \in \{p_2, q_2\}$. Then either $q_2$ is an ancestor of $p_2$ or $p_2$ is an ancestor of $q_2$. Note that $F_2$ is complete, so if $q_2$ is an ancestor of $p_2$, then $q_2$ is the only $q$ vertex closest to $p_2$; if $p_2$ is an ancestor of $q_2$, then $q_2$ is one of the (possibly more than one) $q$ vertices closest to $p_2$. Therefore $dist(p_2, q_2)$ is $\min(p, q)$ in $T_2$.

2. $lca(p_2, q_2)$ is neither $p_2$ nor $q_2$. We prove by contradiction and thus assume $q_2$ is not a closest $q$ vertex to $p_2$ in $F_2$. Suppose $q_2'$ is a closest $q$ vertex to $p_2$, and $x_2 = lca(p_2, q_2')$. $L_2(x_2) = x$. According to Lemma 1, there exists a $p$ vertex $p_2'$ that is closest to $q_2$, because $F_2$ is complete. Also suppose $x_2' = lca(p_2', q_2)$. Again, $L_2(x_2') = x$ since $F_2$ is complete. Assume these newly-introduced vertices’ counterparts in $F_2$ are described by the following associations: $(x_1, x_2), (x_1', x_2'), (p_1, p_2), (p_1', p_2') \in A$, as illustrated by arrowed lines in Figure 21. According to Lemma 2, $(x_1, q_1), (x_1', q_1), (x_1, p_1)$ and $(x_1', p_1')$ are all closest pairs in $F_1$, as represented by curved lines in Figure 21. However, if $x_1$ and $x_1'$ are distinct vertices, then $(p_1, q_1)$ is not a closest pair. This can be checked in a way similar to the proof of Lemma 2. This contradicts with the assumption that $(p_1, q_1)$ is a closest pair.

A subtlety arises when $x_1$ and $x_1'$ are identical. In this case, $(p_1, q_1)$ is a closest pair, but $(p_2, q_2)$ is not. This could be remedied by short-circuiting $x_2$ and $x_2'$. It is reasonable to do this since $x_2$ and $x_2'$ come from the same source vertex and hence they should have a distance of zero. By short-circuiting vertices that are cloned from the same vertex in the source, $(p_2, q_2)$ becomes a closest pair as well. □
Theorem 1 shows that the poly-transform does preserve closeness. Because of that, we can prove that a poly-transform restructuring is always inclusive on a complete source and with a complete target signature.

**Theorem 2** Given a complete forest and a complete signature, the poly-transform is inclusive. □

**Proof** [Theorem 2] Let polytrans\((F, x) = (F_x, A)\) be a poly-transform restructuring; also suppose polytrans\((F_x, s) = (F', B)\), and \(C\) is the composed association of \(A\) and \(B\). To show that polytrans\((F, x)\) is inclusive, we need to show \(F \subseteq C F'\), i.e., \(\forall (u, v) \in E, \exists (u', v') \in E'\) such that \((u, u'), (v, v') \in C (E\) and \(E'\) are respectively the edge sets of \(F\) and \(F')\). Since \(u\) and \(v\) have a distance of one, they form a closest pair in \(F\). By Theorem 1, for any pair \((u_x, v_x)\) of nodes in \(E_x\) such that \((u, u_x), (v, v_x) \in A, (u_x, v_x)\) is a closest pair (not

![Diagram of closest pair and not a closest pair with labels p, q, l, and dotted lines indicating association and ancestor-descendant relationships.](image-url)
necessarily an edge though). \(u_x\) and \(v_x\) exist because a poly-transform restructuring is total. Similarly, fixing a pair \((u_x, v_x)\), for any pair \((u', v')\) such that \((u_x, u'), (v_x, v') \in B\), \((u', v')\) is a closest pair. Furthermore, this pair necessarily forms an edge. This is because (1) \(F\) and \(F'\) have the same signature \(s\), (2) \(L(u) = L(u')\), and (3) \(L(v) = L(v')\); hence a closest pair \((u', v')\) in \(F'\) has to have a distance of one as well. For any edge \((u, v) \in E\), we have found an edge of \(F'\), \((u', v') \in E'\) such that \((u, u'), (v, v') \in C\). Therefore, \(\text{polytrans}(F, x)\) is inclusive. \(\Box\)

Theorem 2 states that all poly-transform restructurings are inclusive. Hence, for any edge \((u, v)\) in \(F\), there is an edge \((u', v')\) in \(F'\) such that \(u\) and \(u'\), and \(v\) and \(v'\) are associated by the composed node association \(C\). What about the opposite case? We first need to introduce the opposite notion of closeness preservation as follows.

**Definition 21** [inverse closeness preservation] Suppose \(F\) and \(F'\) are source and target of a transformation and \(A\) is the association relation. This transformation is *inverse closeness preserving* if, for any \((p, p'), (q, q') \in A\), \((p', q')\) is a closest pair implies that \((p, q)\) is a closest pair. \(\Box\)

Inverse closeness preservation is a desirable property for a transformation because of the following.

**Theorem 3** A poly-transform restructuring is non-additive if it is inverse closeness preserving. \(\Box\)
Proof [Theorem 3] The proof is similar to that of Theorem 2, only going in the opposite direction. □

Recall that a transformation is reversible if and only if it is both inclusive and non-additive. Theorem 2 ensures that a poly-transform restructuring is always inclusive (when the source and target are both complete). To decide if a restructuring is reversible, we need to tell whether it is non-additive as well. By Theorem 3, this reduces to deciding whether it is inverse closeness preserving.

Theorem 4 below shows how to determine inverse closeness preservation (and hence reversibility) by examining the source and target signatures. We first define a term that will be used in the theorem.

Definition 22 [bonding label] Suppose x and y are labels in a signature, and lca is the least common ancestor of x and y. A label is a bonding label of x and y if it is lca or a descendent of lca. □

For example, consider a simple signature p#q#(r#x,y). Since the least common ancestor of x and y is q in this signature, all labels except p are bonding labels of x and y.

A certain b node can help decide whether two nodes with labels x and y are closest, hence “b bonds x and y”. This is formally described by the following lemma.

Lemma 3 Suppose label b is a bonding label of labels x and y. If x is an x node, y is a y node, and b is a b node, then
• \( \text{Closest}(x, b) \land \text{Closest}(y, b) \Rightarrow \text{Closest}(x, y) \), and

• \( \text{Closest}(x, b) \land \lnot \text{Closest}(y, b) \Rightarrow \lnot \text{Closest}(x, y) \). \hfill \square

Lemma 3 shows how exactly a bonding label \( b \) bonds \( x \) and \( y \). As an example, let’s continue with the signature \( p\#q\#(\#x,y) \), in which all labels except \( p \) are bonding labels of \( x \) and \( y \). Consider bonding label \( r \) for example. The first bullet says that if a pair of \( x \) and \( y \) nodes, \( x \) and \( y \), are closest to the same \( r \) node, then \( x \) and \( y \) are closest. The closeness is guaranteed by their being in the same subtree rooted at a certain \( q \) node, in which \( r \) also resides. Similarly, if \( r \) is closest to \( x \) but not \( y \), then \( x \) and \( y \) are not closest, which is what the second bullet says. On the other hand, consider label \( p \) which is not a bonding label of \( x \) and \( y \). If a \( p \) node \( p \) is closest to both \( x \) and \( y \), they may or may not be closest depending on whether they are in the same subtree rooted at a certain \( q \) node. Similarly, if \( p \) is closest to \( x \) but not \( y \), they still may or may not be closest. So \( p \) does not help in determine the closeness of \( x \) and \( y \).

Lemma 3 is useful in proving the following theorem which determines whether a poly-transform restructuring is inverse closeness preserving, and hence reversible.

**Theorem 4** A poly-transform restructuring is inverse closeness preserving if and only if for each pair of labels \( x \) and \( y \) (both are present in both source and target), at least one of the followings holds:

• \( x \) and \( y \) are ancestor-descendent in the target signature;
in the target signature $x$ and $y$ each has an ancestor no higher than their lca, which is also a bonding label of $x$ and $y$ in the source signature. □

Theorem 4 says that inverse closeness preservation (and hence reversibility) is equivalent to the satisfaction of at least one of the above two conditions, for each pair of labels $x$ and $y$. The first condition only concerns the target signature. If $x$ and $y$ are ancestor-descendent in the target signature, then we do not need to worry about the source. Otherwise, $x$ and $y$ each needs to have an ancestor that is (1) no higher than their lca, and (2) a bonding label of $x$ and $y$ in the source signature. A common special case is that the lca of $x$ and $y$ in the target is a bonding label of $x$ and $y$ in the source, which satisfy both (1) and (2).

Let’s consider three sample signatures: $a#b#c#d$, $a#b#(c,d)$, and $a#(b#c,d)$. Table 2 shows whether each specific pairing of the signatures result in an inverse closeness preserving restructuring. First, consider the column where the target is $a#b#c#d$. The first bullet in Theorem 4 is satisfied for every pair of labels, so the restructuring is always inverse closeness preserving. Now consider the column where the target is $a#b#(c,d)$. The first bullet is satisfied for all pairs except $c$ and $d$. In the target, the only no-higher-than-lca ancestor for $c$ and $d$ is $b$; we then examine whether $b$ is a bonding label of $c$ and $d$ in the source. The restructuring is not inverse closeness preserving only for the source signature $a#b#c#d$, because $b$ is not a bonding label in $a#b#c#d$ only. The last column can be similarly derived. Note that any cell on the upper-left to lower-right diagonal contains a
“yes”, which means that a restructuring is always reversible when the source and target signatures are the same.

<table>
<thead>
<tr>
<th>inverse closeness preservation (hence reversible)?</th>
<th>target signature</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>source signature</td>
<td>a#b#c#d</td>
<td>yes</td>
</tr>
<tr>
<td>a#b#(c,d)</td>
<td>a#b#(c,d)</td>
<td>yes</td>
</tr>
<tr>
<td>a#(b#c,d)</td>
<td>a#(b#c,d)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Proving Theorem 4 is more involved, as shown below.

**Proof** Suppose nodes \( p, q \) are in the source, nodes \( p', q' \) are in the target, \( L(p) = L(p') = p, L(q) = L(q') = q \), \( (p, p'), (q, q') \in A \), and \( (p', q') \) is a closest pair in the target.

(if) We need to show that if the condition is true, then \((p, q)\) is a closest pair in the source.

There are two possibilities:

Labels \( p \) and \( q \) are ancestor-descendant in the target. In this case, obviously \((p, q)\) is a closest pair.

Labels \( p \) and \( q \) are not ancestor-descendant in the target. Suppose in the source \( a_p \) and \( a_q \) are both bonding labels of \( p \) and \( q \). Also suppose in the target \( lca \) is the least common ancestor of \( p \) and \( q \), \( a_p \) is an ancestor of \( p \), \( a_q \) is an ancestor of \( q \), and both \( a_p \) and \( a_q \) are no higher than \( lca \). Proving by contradiction, suppose \((p, q)\) is not a closest pair in the source. Then there exists an \( a_p \) node \( ap \), where \((ap, p)\) is a closest pair (because \( a_p \) is an ancestor
of $p$), but $(ap, q)$ is not (because otherwise the path from $p$ via $ap$ to $q$ is a closest path).

Since $ap$ is an ancestor label of $p$ in the target, $ap$’s counterpart in the target, $ap'$, form a closest pair with $p'$ (according to how poly-transform appends nodes in the target). Also, since $ap$ is no higher than $lca$ in the target, $q'$ is not a closest $q$ to $ap'$. Hence, $q'$ and $p'$ do not form a closest pair in the target (Lemma 3, since $ap$ is a bonding label of $p$ and $q$). This contradicts with the assumption that $(p', q')$ is a closest pair.

**(only if)** We need to show that if the condition is not true, then $q$ and $p$ are not necessarily a closest pair in the source. Suppose (i) in the target $p$ and $q$ are not ancestor-descendant, and (ii) in the source no label between $p$ (exclusively) and $lca$ (inclusively) is a bonding label. These two conditions guarantee that, by application of Lemma 3, all $p$ nodes are closest to each $q$ node. Therefore, regardless of $q$ and $p$, $q'$ and $p'$ form a closest pair. 

#### 3.4.4 Poly-Transform Enabled XQuery

There are two complications with restructuring using XQuery. First, dependency on information of the source structure reduces the flexibility and reusability of an XQuery program. Second, programming an XQuery query is non-trivial as well as error-prone, since for instance, duplicates may have to be eliminated by the restructuring. Embedding the poly-transform in XQuery would make transformations simple. For example, the following query would produce `publishers.xml` from `books.xml`.

```xml
for $root in //bibliography
  return polyTRANSFORMDupElim($root, "bibliography#publisher#book(title,author,editor")
```

77
The poly-transform enabled XQuery program does not assume any knowledge of the source structure, except for finding the root elements in the target document, in this case a bibliography node. So this query could be issued against any XML source. It grabs whatever bibliographical information is contained in the source and has labels in the target signature. The second benefit is that it is a simple extension. The only syntactic addition is a new function. In general, if the target signature is $\text{sig}$, the poly-transform enabled XQuery is

$$\text{for } \$root \text{ in } //\text{prefix} \\
\quad \text{return polyTRANSFORM($\$root$, $\text{sig}$)}$$

Alternatively, the source could be transformed as follows to the required structure as the first step in the query.

$$\text{for } \$p \text{ in } \{\text{for } \$root \text{ in } //\text{prefix} \text{ return polyTRANSFORM($\$root$, $\text{sig}$)}\} \\
\quad \text{---rest of query---}$$

Since transforming the source is potentially expensive, a lightweight alternative is to add a new axis to XPath: the closest axis (see Section 4). The closest axis finds all the nodes that are closest to the context node. Let $\#$ be abbreviated syntax for the axis. Then, the following query can be used to find the titles of books published by the author Codd regardless of whether the structure is $\text{books.xml}$ or $\text{publishers.xml}$.

$$\text{for } \$a \text{ in } //\text{title}[\#\text{author}="Codd"] \\
\quad \text{return } \$\text{title}$$

Finally, it is important to note that in poly-transform enabled XQuery, all of the XQuery functionality to utilize path expressions to find data and filter the selected data through
the conditions in a where clause is still available. The poly-transform only transforms the
data selected in a FLWR expression (which can be input to yet other FLWR expressions); it extends rather than replaces XQuery.

3.5 Experiments

This section describes two experiments that we performed with a prototype Java implementation of the poly-transform. Both experiments were performed on a typical PC: hyperthreaded 2.8GHZ CPU, 2GB SDRAM, Windows XP, and Java (jdk 1.4.2). Although a Java implementation will not be as fast as a C implementation, we chose Java because it has libraries for XML parsing and DOM building that enable rapid, robust construction of applications that process XML. The code is available from our website\(^5\).

Experiment one evaluates the basic cost of the poly-transform. This experiment tests the speed of transforming an XML document, which is a list of \(<book>\) elements, using three different signatures. Each \(<book>\) has an ISBN attribute and contains four subelements: a unique \(<title>\), an \(<author>\) (an author may have written up to seven books), and a \(<publisher>\) (there are twenty, different publishers). Each author has two subelements: \(<first>\) and \(<last>\). We chose to test three signatures to evaluate whether the form of the signature had any effect. The three signatures are

- a *deep* signature: last#publisher#title#first#author,
- a *flat* signature: publisher#(last,title,first,author), and

\(^5\) [www.eecs.wsu.edu/~cdyreson/pub/polytransform](http://www.eecs.wsu.edu/~cdyreson/pub/polytransform)
• a mixed signature: publisher#title#author#(last,first).

The deep signature transforms the tree into a chain. The flat signature flattens the tree structure. And the mixed signature combines aspects of the flat and deep signatures. We also wanted to observe the poly-transform on documents of increasing size, so we performed the experiment on documents with from 150,000 to 1,500,000 elements (3MB to 32MB in size, respectively). Finally, we wanted to determine the cost of duplicate elimination, so we tested the transformation with and without duplicate elimination.

Figure 22 shows the result of experiment one. The bars from left to right for each document size plot the cost of transforming to each of the three signatures, followed by the cost with duplicate elimination. The graph shows that the cost of the poly-transform without duplicate elimination is roughly linear in the size of the document and that there is very little difference between the various signatures. Recall that the time complexity of the poly-transform is $O(ns)$, where $n$ is the number of nodes and $s$ is the number of labels in the signature. Duplicate elimination adds an additional 20%-40% for these document sizes, and the differences in the signatures become apparent, though they remain slight. As the document gets larger, duplicate elimination costs increase since more elements are being eliminated. The complexity of duplicate elimination is $O(nk)$, where $k$ is the cost of determining a duplicate. To better understand the transformation we measured the cost of separate steps in the process. The transformation consists of two major phases: 1) an initialization phase that parses the document and constructs internal data structures (the lists of elements), and 2) a transformation phase, which is the poly-transform algorithm.
Figure 23 compares the cost of each phase for each document size across all the experiments. Initialization is roughly 40% of the total cost. In an XML DBMS the initialization phase would be absent, assuming the DBMS has previously parsed the data and built indexes to iterate through nodes of each type.

Figure 22 Experiment one measures three signatures

Figure 23 Relative cost of initialization vs. poly-transform
Figure 24 Poly-transform enabled XQuery vs. XQuery

Experiment two is a “real-world” comparison of XQuery with poly-transform enabled XQuery. We chose an open source XML DBMS called eXist as the test XQuery engine. For the experiment we chose the first XML Query Use Case [59]. Among all the queries in the use case, only four “restructure” the document: Q2, Q3, Q4, and Q11. Q2 produces a list of titles and authors, while Q3 eliminates duplicate authors for each title (i.e., it groups authors). Q4 lists titles by author, and eliminates duplicates. Finally Q11 is a list of books with authors, followed by a list of books with editors. For data in the experiment we chose <inproceedings> elements from the DBLP data set (i.e., real-world data), converting them to <book> elements to match the Use Case schema. We randomly
selected elements to create two documents: one with 1,000 books and one with 10,000 books. The experiment tested the queries against the “equivalent” poly-transform enabled-XQuery query by evaluating the “root” portion of the query in eXist, and then applying the poly-transform on the result (as discussed in Section 3.4.4). So the poly-transform is a post-processing layer on the DBMS. We counted the cost of the poly-transform enabled XQuery query as the sum of the cost of evaluating the query in eXist, and evaluating phase 2 in the poly-transform. That is, we calculated what it would cost if the poly-transform were supported directly by eXist, and we did not count the cost of eXist constructing a text document, nor the cost of the poly-transform parsing the text document.

Figure 24 shows the result of experiment two. For each query, the leftmost bar is the poly-transform on the small dataset. To its right is eXist on the small dataset, followed by the large experiment tests. Though eXist is very efficient in many queries, a complex restructuring and grouping query in eXist is inefficient, even on relatively small data collections primarily because several passes are required over the source data. Additionally, duplicate elimination is an expensive operation. Since there is no grouping operator in XQuery, XQuery compilers have a difficult time recognizing when a query groups data to eliminate duplicates, and consequently generate (very) inefficient query execution plans for grouping. Overall, the results suggest that the poly-transform can be integrated into an XQuery engine without incurring significant overhead.
3.6 Conclusion

This chapter makes several contributions. First, it articulates the importance of polymorphic operators that work independently of a particular structure in an instance of an XML data model. Second, we focus on restructuring. The chapter formally defines three properties of a data transformation: reversible, non-additive, and inclusive. We argue that “good” restructurings are reversible or at least inclusive so that they do not manufacture relationships among data items. Third, we introduce the concept of closeness preservation as an important semantic constraint in developing good restructurings. Closeness preservation ensures that when data is restructured elements that are close in the source will remain close in the target. Fourth the chapter presents the poly-transform, which is a transformation function that preserves closeness. We presented an algorithm that implements the poly-transform. The poly-transform has a very simple syntax, making it intuitive for naïve users and easy to integrate into a query language for hierarchical data. Fifth, we described a new kind of join that can efficiently implement the poly-transform. Finally, the chapter describes two experiments that show that the poly-transform is efficient with a cost proportional to the size of the input.

The poly-transform is not the only possible application of the closeness preserving property. The problem of restructuring data is similar to the problem of change detection which concerns identifying elements in different versions of a document that represent the same real-world entity. Traditional change detection technique for XML do poorly when the structure of data changes dramatically, for instance, they would overestimate the “difference” between books.xml and publishers.xml even though the two
documents contain the same data but are structured very differently. By transforming one structure to the other using the poly-transform and comparing identical structures, it should be possible to improve the accuracy of change detection. So the poly-transform has potential applications in version management and temporal queries. We would also like to extend the poly-transform to include a more expressive grouping mechanism, whereby groups can be created on a subset of values relative to a node. Other extensions of the technique include changing the metadata during restructuring, for instance, restructuring to a different namespace or transforming between an attribute and a subelement (which would be rather trivial with the poly-transform).
4. SYMMETRICALLY EXPLOITING XML

Path expressions are the principal means of locating data in a hierarchical model. But path expressions are brittle because they often depend on the structure of data and break if the data is structured differently. The structure of data could be unfamiliar to a user, may differ within a data collection, or may change over time as the schema evolves. This chapter proposes a novel construct that locates related nodes in an XML forest, independent of a specific structure. It can augment many XPath expressions and can be seamlessly incorporated in XQuery or XSLT.

4.1 Introduction

In 1970, E. F. Codd proposed a relational model of data to replace the popular (at that time) hierarchical model [15]. Codd critiqued the hierarchical model because it did not support the symmetric exploitation of data. The hierarchical model used asymmetric path expressions to locate data. A path expression is a specification of a path (or a set of paths) in a hierarchy. Path expressions are asymmetric because they depend on how the data in the hierarchy is structured, and the same data can be organized in different hierarchies. Codd presented five reasonable hierarchies for a simple part/supplier data collection and demonstrated that, in general, a path expression formulated with respect to one hierarchy, would fail on some other hierarchy.

---

6 Codd’s critique included other arguments such as the critical concept of (physical) data independence that are not germane to this chapter.
It is generally accepted that Codd won the debate with the hierarchical model as evidenced by the current industrial dominance of relational database management systems. But thirty-five years later a hierarchical data model has resurfaced with the advent of XML. XML data models are tree-like, hierarchical models. As a consequence, asymmetric path expressions have reappeared in XML query languages. The core of most XML query languages is a path language to navigate to various places in a hierarchy. For XQuery the path language is (a subset of) XPath. Asymmetric path expressions make queries brittle in the sense that a query might fail to produce the desired result if the structure changes or if it is executed on the same data organized in a different hierarchy.

There are four scenarios where symmetric exploitation of hierarchical data is particularly attractive: lack of schema knowledge, heterogeneous data, irregular data, and schema evolution. First, detailed knowledge of the data’s structure or schema is often needed in order to correctly formulate a path expression. Many data collections lack a schema, and even when a schema is present, it may be complex and difficult to decipher for some users. The ability to query data without knowing its specific structure would be useful for both expert and casual users. Second, there is the increasing need for data integration. It is becoming common to pull data from different sources into a single data collection. Each source could organize similar data in a different hierarchy. If queries are not symmetrically exploitable, then a single logical query over the heterogeneous hierarchies would potentially require different path expressions for each structure. Third, the decentralized nature of the web has facilitated a growth in the generation and exchange of data authored by casual users. More often than not, data provided by these users does not
conform to a strict schema; rather the data in a single collection is *irregularly structured*. Last, even in a centralized database with a single, simple, well-defined schema, shifts in business strategy and corporate environments sometimes engender evolution in how data is organized. Legacy path expressions that depend on a particular hierarchy may no longer work when a *schema evolves*.

A common theme underlying the various scenarios above is that the asymmetric nature of path expressions makes them brittle. This chapter proposes a novel extension to XPath to support the symmetric exploitation of XML data. We extend XPath with a *symmetric locator*. This extension allows a user to query XML data without knowing its exact structure in most situations. That is, a user simply needs to know the names of relevant elements and attributes and their possible relationships to properly formulate a query. The extension is simple in syntax and semantics. Specifically, we introduce a new axis: the *closest* axis, which locates nodes that are closest to a context node. In abbreviated syntax the closest axis is represented by an operator “$>”, so the expression “$>t” locates all nodes of type t “closest to” the node bound by $n. Remarkably, this simple operator can replace asymmetric steps in path expressions in most XQuery queries written for daily tasks.

This chapter is organized as follows. A motivating example is presented in Section 4.2. Section 4.3 and Section 4.4 defines the syntax and semantics of the closest axis respectively. We consider both in-memory and persistent implementation in Section 4.5. Section 4.6 illustrates the use of the closest axis by rewriting some of the queries in the
XML Query Use Cases [59]. Section 4.7 discusses related work and Section 4.8 concludes the chapter.

4.2 Motivation

Consider how a collection of bibliographic information such as authors, books and publishers may be represented in a hierarchy. One of (but not limited to) the following two hierarchies may be used: 1) the hierarchy contains a list of authors, each of which contains a list of books by that author, or 2) the hierarchy contains a list of books, each of which contains a list of authors of that book. As an example, consider some bibliographic data about the author E. F. Codd that involves two books and two publishers. The two different representations are captured by two XML document fragments, cs_authors.xml and cs_books.xml, shown respectively in Figure 25 and Figure 26. Both documents contain the same data but they have different structures.

```
...<author>
  <name>E. F. Codd</name>
  <book>
    <title>The Relational Model for Database Management</title>
    <publisher>Addison Wesley</publisher>
    <price>$46.95</price>
  </book>
  <book>
    <title>Cellular Automata</title>
    <publisher>Academic Press</publisher>
    <price>$9.95</price>
  </book>
</author>
... 
```

Figure 25 A fragment of cs_authors.xml
For some queries, different path expressions are needed to query each hierarchy. For example, consider a query to retrieve books by E. F. Codd. The XQuery query for `cs_authors.xml` is given below.

```
return doc("cs_authors.xml")//author[name='E. F. Codd']/book
```

This query uses a path expression that navigates from the root to the proper author elements and then finds the desired book. But this query does not work for `cs_books.xml` since it has a different structure. To query `cs_books.xml` a different query has to be formulated.

```
return doc("cs_books.xml")//book[author/name='E. F. Codd']
```

The path expression in each query differs. Moreover, no single path expression suffices to locate the desired data in both hierarchies.
While asymmetric path expressions are wedded to a particular hierarchy, the key to developing a symmetric locator is to identify what is invariant across the same data organized in different hierarchies. Observe that in both Figure 25 and Figure 26, book titles by an author are closest to that author. Here “closeness” is the distance on the path between nodes in the hierarchical model of an XML document. In both hierarchies the author E. F. Codd is the closest author to each of the titles. This is not something specific to &lt;author&gt;s and &lt;title&gt;s only. In fact, whenever two nodes are closest in Figure 25, so are their counterparts in Figure 26.

This invariant property of different hierarchies can be exploited with a symmetric locator that locates related information based on closeness rather than a specific path. The symmetric locator is a closest axis. In abbreviated syntax the axis is denoted by an operator “->”. Semantically, the axis locates all nodes that are closest to the context node. So “$v->b” returns a sequence of all b nodes closest to the node bound by $v. With this operator, the query posed at the beginning of this section can be expressed as follows:

```
return doc("any.xml")-&gt;author-&gt;name='E. F. Codd]-'&gt;book
```

where “any.xml” could be either “cs_authors.xml” or “cs_books.xml”. It can also be applied to a hierarchy that, for example, contains a list of publishers, each of which contains a list of books. Furthermore, the query works for data with a heterogeneous structure. Suppose we mix bibliographic data from multiple sources (say, from the hierarchy of Figure 25 and that of Figure 26), the same query would work without any change. In contrast, it would be cumbersome to formulate a query using asymmetric path expression to query
heterogeneous data. The user first has to know what structures are present in the data, and then potentially needs to write a different path expression to cater to each distinct structure in the data. Complicated as it is, such a query could only handle those hierarchies taken into account, and may need to change whenever there is some schema change. In summary, the closest axis is more convenient to formulate and more robust against structural changes than asymmetric path expressions.

Although the closest axis is intended to replace asymmetric XPath axes in many practical uses, it cannot replace all of them. Without the sophisticated navigational functionalities provided by asymmetric path expressions, a query language may be less than Turing-complete [34]. When path expressions are inevitable for a task, one still needs to resort to XPath. (See Section 4.6 for more discussion on this.) So it is important to remember that the closest axis extends but does not replace path expressions.

4.3 Syntax

The closest axis has a very simple syntax that can be seamlessly integrated into XPath. Figure 27 shows the EBNF grammar for the axis, where the newly-introduced symbols are underlined.

This grammar extends the current XPath grammar defined in the W3C candidate recommendation “XML Path Language (XPath) 2.0” [58]. (It can be found in “Appendix A: XPath Grammar”.) There are 73 rules in the current XPath grammar, among which only rule [29] has to be modified. The modified rule, annotated [29'], introduces a new
step, ClosestStep. ClosestStep is further defined by the new rule \([n1]\). A ClosestStep may or may not be abbreviated. New rules \([n2]\) and \([n3]\) define the unabbreviated and abbreviated syntax, respectively. An unabbreviated step is in the form of closest::NodeTest, and an abbreviated step is ->NodeTest. The new XPath grammar has a total of 76 rules, with one of the original rules modified and three added.

\[
[29'] \text{AxisStep} ::= (\text{ForwardStep} \mid \text{ReverseStep} \mid \text{ClosestStep}) \text{ PredicateList}
\]

\[
[n1] \text{ClosestStep} ::= \text{ClosestAxis} \text{ NodeTest} \mid \text{AbbrevClosestStep}
\]

\[
[n2] \text{ClosestAxis} ::= <"closest" ":">\text{NodeTest}
\]

\[
[n3] \text{AbbrevClosestStep} ::= "->" \text{NodeTest}
\]

Figure 27 EBNF grammar for the extended XPath

### 4.4 Semantics

This section presents the formal semantics of the closest axis. We first present a data model for XML documents, and then define the closest axis. The specific semantics of the axis depends on the important concept of node “type”. We also discuss a technique that computes node type in the absence of data schema.

#### 4.4.1 Tree Data Model

XML documents are commonly modeled as ordered, labeled trees. We first define such an XML data model in Definition 23. Note that Definition 23 bears similarity with
Definition 2 on page 12, but it is different in a few aspects. We will adopt the following definition in the discussion of this chapter.

**Definition 23** [tree] A tree is a tuple \((V, E, \Sigma, L, C, T)\), where

- \(V\) is the **node set**. \(r \in V\) is a special node called the **root** of \(T\),
- \(E \subseteq V \times V\) is the **edge set** such that there is a path between every pair of nodes, there is no cycle among the edges, and edges that share a common node – called the **parent** of the other node (the **child**) in each such edge – are ordered,
- \(\Sigma\) is an **alphabet** of labels and text values,
- \(L : V \rightarrow \Sigma\) is a **label function** that maps each node to its label,
- \(C : V \rightarrow \Sigma \cup \{\varepsilon\}\) is a **value function** that maps a node to its **value**, in which \(C(v) = \varepsilon\) if node \(v\) has an empty value, and
- \(T : V \rightarrow S\) is a **type function** that maps each node to a **type**, which is a value in the **type set** \(S\).

This tree data model is a stripped-down version of the Document Object Model (DOM) [56]. Though the model is simple it is sufficient for our purposes in this chapter. Elements are represented by nodes in our tree data model. Other kinds of nodes in the DOM such as attributes and comments are ignored.\(^7\)

The label function maps each node to its label, that is, its element tag. So a \(<\text{book}>\) node would map to the label `book`.

\(^7\) Due to this simplification, the closest axis will locate elements only. However, the closest axis can be defined for a complete data model as well; in that situation all kinds of nodes can be located.
The type function refines the label function by possibly partitioning the nodes associated with the same label into different types. The type function specifies the type of each node. There are several ways that the type could be computed. If a schema is available, then the type of a node might be given by an element type definition in the schema. For instance a <name> element type definition might provide the type for author names, while publisher names resolve to a different type even though they share a common label. When a schema is unavailable, the simplest case is to make the type and label functions the same. That is, nodes are typed by their labels. Because the type plays an important role in the definition of the closest axis, Section 4.4.3 and Section 4.4.4 give two more refined methods of computing the type in the absence of a schema.

Ordering of elements is important for XML. We adopt the common document order which orders nodes in a tree data model based on the first appearance of the corresponding text in the document. In cs_authors.xml, for example, text fragment <name> appears earlier than both occurrences of <book>. Hence, the name node precedes the book nodes in the tree data model.

4.4.2 The Closest Axis

As introduced in the motivating example in Section 4.2, the closest axis is used to locate “the closest nodes” to the context node. The result of evaluating a closest axis is a sequence of nodes that are closest to the context node.
**Definition 24** [the closest axis] Suppose \((V, E, \Sigma, L, C, T)\) is a tree and \(c \in V\) is the context node. The closest axis evaluates to a sequence of nodes \([d_1, \ldots, d_n]\), where

- \(d_1, \ldots, d_n \in V\),
- \(\forall i, 1 \leq i \leq n, \forall x, y \in V, L(x) = L(d_i) \land T(y) = T(c) \Rightarrow dist(c, d_i) \leq dist(y, x); \quad (i)\)
- \(\forall i, j, 1 \leq i < j \leq n, d_i \text{ precedes } d_j \text{ in document order.}\) \(\quad (ii)\)

The closest axis is defined as a function that takes a context node \(c\) and returns a node sequence. The function has two primary conditions. First, the **node ordering condition** \((ii)\) states that the result preserves the original document order. Second, the **node selection condition** \((i)\) constrains the nodes that appear in the result. The condition stipulates that a node is in the result if it is the closest node of a particular label to the context node, but that the distance to the closest node is within the minimal distance of a node of the same type as the context node to a node of the same label as a node in the result. The intuition is that the closest axis seeks out all the nodes of each different label that are closest to the context node, but restricts the search based on the minimal possible distance between a given type and a given label.

To better understand the meaning of the node selection condition, let’s consider an example. Figure 28 shows the hierarchy for the XML given in Figure 25. Assume that the type of each node is its label. Let the context node be the leftmost **title** node. The nodes closest to the **title** node are pointed to by dashed arrows. The five closest nodes are...
ordered in the closest axis in document order. In this example there is only one node of each label that is closest, but in general there could be several nodes with the same label that are closest. Note that none of the nodes in the other book subtree is closest to this title.

![Diagram of nodes closest to the first title]

**Figure 28 Nodes closest to the first title**

In the above example, the node selection condition in the axis definition could in fact be simplified to the following

$$\forall i, 1 \leq i \leq n, \forall x \in V, L(x) = L(d_i) \Rightarrow \text{dist}(c, d_i) \leq \text{dist}(c, x) \quad (i')$$

in which the types are absent. This simplification is possible because the tree is of a regular structure. We say that the tree’s structure is *regular* if the following holds

$$\forall c \in V, \forall l \in \Sigma, \exists v \in V \text{ such that}$$

$$L(v) = l \land \text{dist}(c, v) = \min\{ \text{dist}(y, x) \mid L(x) = l \land T(y) = T(c) \}. $$
That is, for any node $c$ and any label $l$ in a regular tree, we can always find a node $v$ labeled $l$ such that $c$ and $v$ are of the minimal distance of nodes typed $T(c)$ and nodes labeled $l$. The fact that every node is guaranteed to have a minimal distance node of any given label reduces $(i)$ to $(i')$.

For hierarchies that are not regular, the node selection condition $(i)$ is more appropriate than the simplified version $(i')$. For example, suppose that the first book does not have a price child, as shown in Figure 29. Here the label and type are still the same; but the tree is irregular in the sense that the context title node could not find a label price node within the minimal distance of all possible pairings of type title node and label price node, which is two. The closest axis using condition $(i')$ (just the labels) would locate the price child of the second book since it is the closest price. But this price should not belong to the closest axis of the first book because it is closer to the second book than the first.

*Figure 29 The search is restricted by the type information*
In the trees in Figure 28 and Figure 29, the label and type functions are the same. But there are scenarios where nodes of the same label should be differentiated for the results to be intuitively appropriate. For example, the price of a book and the price of a car need to have different types. Suppose the closest node labeled name to car price is four while the closest node labeled name to book price is three. If all price nodes were of the same type, then no car price would have any name node in its closest axis. By distinguishing the two types of price nodes, a car price node can include in its closest axis all name nodes at a distance of four. Note that the book price’s axis contains author name, while the car price’s axis probably contains owner or dealer name, a different type of name. But the node selection condition (i) is concerned with type of the context node only, but not the type of the nodes in the axis.

Our use of the term “closest” to describe the new axis is evocative but imprecise. For some data collections, the results may be counterintuitive. For example if an author name node is equi-distant from both a book price type and a car price type (an unlikely scenario, but possible) then the closest axis of this node includes nodes of both type book price type and car price. So while the closest axis can be used to symmetrically exploit data, the meaning of the axis depends on the data.

The closest axis is similar to the current XPath axes insofar as it returns a node sequence relative to a context node. A node test and predicates can be further applied to filter the sequence. Unlike all other axes, the closest axis is a non-directional axis. That is, it does not locate nodes in a particular direction (up, down, left, right) in the hierarchy. Instead it
utilizes node and type information to find nodes that are close to the context node in any direction. Only non-directional axes can symmetrically exploit data.

An interesting consideration is whether nodes connected by ID/IDREF relationships can be considered as closest nodes. In a tree, there would be no edge between two such nodes, so the two nodes would not be closest. But we could easily add a “virtual” edge to connect such nodes and compute distances in the resulting graph. However, in the interest of simplicity we do not consider such virtual edges.

4.4.3 Root-to-Node Path Type

Node type is important in determining which nodes are in the closest axis. The use of types takes into account the fact that different kinds of real-world entities may be represented by nodes of identical labels in a tree. So a proper type function $T$ should for example distinguish a book price from a car price.

The type function can sometimes be easily inferred when the data is accompanied by schema information in the form of DTD or XML Schema. But a common situation is that we do not know the schema. We now introduce a technique to compute the types in the absence of a schema. The node types produced by these are potentially helpful in refining the possible results of a closest axis.
**Definition 25** [root-to-node path type] A tree \((V, E, \Sigma, L, C, T)\) uses *root-to-node path type*, or *path type* in short, if for any \(v \in V\), \(T(v)\) is a list of the labels of the nodes on the inclusive path from the root \(r\) to \(v\). □

The path type is essentially a concatenation of the labels of the nodes from the root to the node. With path type, nodes of the same type always have the same label, but not vice versa. The rationale behind path type is the following claim:

*If two nodes are of the same type, then their respective child nodes with the same label should be of the same type too.*

It is rather common for two nodes in a tree to have the same label but represent different kinds of entities. However, it is rarely the case that such two nodes’ respective parents have the same type. In our previous example of book price and car price, suppose their respective parents are book and car. The paths from the root node to a book price node and a car price node are different; therefore book price and car price have different path types.

Path type can be efficiently computed when the tree is parsed. The type of a node is obtained by appending its own label to the end of the path type of its parent. The complexity of the computation is \(O(n)\) where \(n\) is the number of nodes in the tree.

### 4.4.4 Signature Type

The signature type is also useful in specifying queries with the closest axis. Furthermore, it is crucial to the efficient implementation of the closest axis.
4.4.4.1 Signature

We have defined the concept of signature in Definition 4 on page 13. A signature is a succinct description of the structure of a data forest. A signature forest never contains two sibling subtrees that are isomorphic.

For example, the tree in Figure 28 (the tree data model of cs_authors.xml in Figure 25) has the signature shown in Figure 30. This signature is smaller in size than the data. The data tree contains two book subtrees of identical structure – each book node has title, publisher and price children, all of which are leaf nodes. Keeping one copy of this book structure in the signature tree is sufficient to capture the structure of the data tree.

![Figure 30 Signature of cs_authors.xml](image)

We now extend the concept of signature from trees to nodes.

**Definition 26** [node signature] The signature of a node \( v \) in a data tree is the signature of the tree rooted at \( v \).  □
Node signature is compatible with tree signature. The signature of a tree is just the node signature of the root node.

4.4.4.2 Signature Type

Observe that there is a correspondence between nodes in a data tree and nodes in the signature tree. We can define the type of a node in a data tree as its corresponding node in the signature.

Definition 27 [signature type] Suppose \( \text{sig}(F) = S \). The signature type of a node in \( F \) is its corresponding node in \( S \). □

Because a tree signature is recursively computed in a bottom-up fashion, node signature is simultaneously determined while computing the signature of the whole tree. As the pre-condition for the node classification algorithm, we assume that node signature of each node is already available, represented by a the function \( \text{sig}() \). The algorithm is listed in Figure 31.

The algorithm computes signature type for all nodes by invoking the recursive function \( \text{visit()} \) on the root of the data tree. It starts with the obvious base condition: the root of a tree is mapped to the root of its signature tree. It then recursively computes the type of each node in a top-down fashion. At a node \( n \) in the data tree, the function returns if \( n \) has no child; otherwise, it decides the type of each of its child by comparing the signature of this child to the each of the child subtree of \( T(n) \), the node in the signature tree that \( n \) corresponds to. (Note that in the algorithm, the tree rooted at \( s \) is equivalent to \( \text{sig}(s) \),
because a signature tree does not contain isomorphic sibling trees.) Once the type of a child \( c \) is determined, \( \text{visit}(n) \) calls \( \text{visit}(c) \) recursively. Here the most costly part is to determine isomorphism between trees. As mentioned before, comparing unordered trees can be efficiently computed with the help of a sort.

**pre-condition:**
- data tree \((V, E, \Sigma, L, C, T)\) and its signature \( S \),
- \( T \) is only defined on the root node \( r \),
- \( T(r) = r', r' \) is root of \( S \),
- \( \text{sig}: V \rightarrow \text{set of trees rooted at } v, v \in V' \).

**computing the types:**

\( \text{visit}(r) \)

**post-condition:**

\( T: V \rightarrow V' \) is defined on all nodes in \( V \)

**function definition** \( \text{visit}(n) \):

if \( n \) has no child

return

else

for each child \( c \) of \( n \)

for each child \( s \) of \( T(n) \) in \( S \)

if \( \text{sig}(c) \) is isomorphic to the tree rooted at \( s \)

then \( T(c) = s \)

\( \text{visit}(c) \)

**Figure 31 An algorithm for computing node type**

### 4.5 Implementation

This section investigates how the closest axis can be efficiently computed. At first sight, it seems quite probable that the evaluation of the closest axis would be completely different from that of a usual XPath axes. Axes like descendent are directional, while the
closest axis is non-directional; its semantics just describes the property of the closest nodes without giving a specific direction to where it is located.

XPath/XQuery implementations can be broadly classified as either *in-memory* or *persistent*. We present both in-memory and persistent implementations in this section. An in-memory implementation loads the entire data tree into memory and evaluates the axis directly on the tree. However, some data collections are too large for memory. In a persistent implementation, the data resides predominately on disk. Indexes are commonly used in persistent implementations to optimize performance by reducing the number of blocks read from disk during query evaluation.

### 4.5.1 In-memory Evaluation

The closest axis can be naively implemented by exploring from the context node in all directions until each kind of label is reached that is within the minimal distance between the type of the label and the type of the context node. Such an evaluation simply has to enumerate all the possible paths starting from the context node to look for the closest node(s). The algorithm that computes all the closest nodes to a context node, \( v \), is shown in Figure 32.

Though the naïve algorithm computes the closest axis it has high cost. The algorithm explores \( \text{maxDistance} \) edges from the context node, potentially covering the entire tree.
pre-condition:
- data tree \( (V, E, \Sigma, L, C, T) \)
- \( typeDistance(\Sigma) \) is a hash table that maps each label to a distance, initially the distance for every label is the distance between the context node type and the closest type for this label
- \( maxDistance \) is the maximal type distance over all the labels
- \( closest(\Sigma) \) is a hash table that maps each label to a list of closest nodes, initially each list is empty
- \( v \) is the context node

computing the types:
\( closest(v, 0, maxDistance) \)

post-condition:
\( closest(\Sigma) \) is a hash table that maps each label to a list of closest nodes

function definition closest(c, d, maxDistance):
// Return if distance exceeds maximum possible
return if \( d > maxDistance \)
// Try each edge from c
for \((c,x) \in E\)
  // Check if this is the right distance
  if \( d = typeDistance(label(x)) \)
    // x is at the right distance
    insert(x, closest(label(x)))
  // Continue exploring from this edge
  closest(x, d+1)

Figure 32 A naïve, in-memory algorithm for evaluating the closest axis

Further, the naïve algorithm assumes the existence of a typeDistance hash table that has already computed the minimal distances between pairs of types. This table can be constructed when a DOM is built or just prior to evaluating the axis using the signature described in Section 4.4.4.1. Essentially, the strategy is to evaluate the closest axis in the
signature forest to find all of the types closest to the type of the context node. For example, in the evaluation of the closest axis from the first title node in Figure 28, the signature shown in Figure 30 could be explored to determine the distance from the title type to the closest types corresponding to each label.

4.5.2 Node Test Optimization

We anticipate that the closest axis will almost always be used with a node test for a specific label, e.g., “closest::price.” (See Section 4.6 for closest axis use cases). The evaluation cost can be significantly reduced in such cases. One way to reduce the cost in the naïve algorithm is to set the maxDistance to the distance of the type corresponding to the label in the node test, e.g., price in the example given above. On average this will cut the cost in half. However, a significantly better strategy is possible.

The better strategy is to convert the non-directional search to a directional search. Observe that a signature provides both a distance and a path to the desired type. To continue with the example, assuming that the context node is a title type, a single path connects the price type to the title type in the signature. The path climbs to the book parent and then drops to the price child. In general, the path between the any two types traverses through the least common ancestor (LCA) of the two types in the signature. So the optimization is to replace the closest axis with a different expression that follows the path to the nodes specified by the node test. In the example, the non-directional expression “closest::price” would be replaced with the following directional path expression:
The conversion can be performed by a pre-processor prior to evaluating an expression, or the directional path expression can be substituted during evaluation of a closest axis. Note that in general, there might be several closest types, so a union path expression that follows all of the paths might be needed.

4.5.3 Persistent Implementation

For the closest axis to be of practical value for database applications, it needs to be computed efficiently in persistent implementations. Since the axis will almost always be used in combination with a non-wildcard node test, we focus on the implementation of the node test optimization. We employ the LCA-join operation presented in Section 3.4.2 to efficiently evaluate the closest axis. We do not reiterate the details already covered in Section 3.4.2, but highlight the key ideas of the LCA-join operation with this chapter’s example data.

Consider the following tree node numbering scheme. Given a tree, assign each node a number according to its ordinal in document order. The numbers range from 1 to \( n \), the total number of the nodes. Each node is also assigned the number of its maximum descendent. All nodes with a number larger than the number of a node \( v \) and no larger than its maximum descendent’s are descendents of \( v \). Figure 33 shows the numbering for the data tree of \texttt{cs_authors.xml} in Figure 25. The first book node has the number 3 and its maximum descendent is 6. So its descendents are all the nodes numbered between 3 and 6.
Next, an index of types is created. The index maps each type in the signature to an ordered list of node numbers for nodes of that type. Then the closest axis can be computed by simply merging three lists. This process has been illustrated by Figure 19 on page 65.

For instance, for title children and publisher parents in Figure 30, the lca is book. The lists are merged in the direction of a lexical ordering of the data (from left to right in the figure). A parent is closest to a child if both are descendent of the same lca. If a parent is not a descendent of the current lca, then either the current lca is before the current parent (child), in which case the current lca pointer is advanced, or the current parent (child) is before the current lca, in which case the current parent (child) pointer is advanced. Typically only two lists are merged instead of three since the parent or child is the lca.

Figure 34 illustrates the task of finding the closest <title> elements to the <publisher> elements for the data tree of cs_authors.xml in Figure 25. The lca is book. The
merging process starts with the pointers at the start of each list. The first publisher and title are both descendents (within the range) of the first book, so this publisher is closest to the title. The next publisher however is not within the range of the current lca hence it is not closest to the first title. The LCA-join continues by advancing the lca and child pointers to find the next closest pair.

The LCA-join is of special importance in database management systems. If an XML DBMS can iterate through elements of a particular type, then the closest axis with a non-wildcard node test can be efficiently computed with an LCA-join. The time complexity of an LCA-join is $O(n)$, where $n$ is the number of nodes in a type list. Indexes to map an element type to a list of nodes for that type are commonly available in native XML DBMSs (e.g., Xindice, eXist, and BerkeleyDB-XML provide element type indexes.)
4.6 Using the Closest Axis in Practice

In this section we turn our attention to the use of the closest axis in practice. To show its wide applicability, we demonstrate how the closest axis can replace directional axes in queries from the first use case in the W3C XML Query Use Cases [59].

We first take a look at two queries from the first use case—“Experiences and Exemplars”. For each query, the problem and the solution using the closest axis are shown.

Q1. List books published by Addison-Wesley after 1991, including their year and title.

Solution using the closest axis:
<bib>
{  
  for $b in doc("http://bstore1.example.com/bib.xml")->bib->book  
  where $b->publisher = "Addison-Wesley" and $b->@year > 1991  
  return  
  <book year="{$b->@year}">  
    {$b->title}  
  </book>  
}
</bib>

Each of the five closest axes in the above query replaces a child axis in the original query. As we can see, there is no directional axis in the modified query.

Q9. In the document cs_books.xml, find all section or chapter titles that contain the word "XML", regardless of the level of nesting.
Solution using the closest axis:
<results>
{
   for $t$ in doc("cs_books.xml")//(chapter | section)->title
   where contains($t->text(), "XML")
   return $t
}
</results>

The closest axes in the above query replace child axes in the original query as well. However, we choose not to replace the descendant-or-self axis with the closest axis. Although it is usually the case that all chapter nodes are of the same distance to the root of a document, it may not always be true. When some chapter nodes are farther away from the root, the closest axis will miss these nodes even though it is the intention that they be selected. In this particular case, doc("cs_books.xml")//chapter is still the best way to properly locate chapter nodes. This example shows that when the descendant-or-self axis is invoked from the document root it functions as a symmetric locator. The expression doc("anyBibDoc.xml")//book is not structure-dependent, and hence is symmetric in effect.

In addition to the queries shown above, we have also inspected other queries in this use case. It turns out that every directional axis in the various use cases can be replaced by the closest axis, with the exception of the descendant axis. This can be explained by the fact that these XPath expressions are all used to located related nodes, and the related nodes are always closest to the context nodes. In this use case, there is no instance of the use of the parent axis; but the closest axis should be effective in replacing it as well should it be used.
Although not present in the entire use cases document, recursive hierarchies can also be problematic. Consider a recursive schema in which all part elements are nested to represent subparts. Located at the leaves in the hierarchy are those atomic parts that have no subparts. In such a hierarchy the expression part->part may or may not give the expected result. This expression returns the immediate subparts while some users may believe all (recursive) subparts should be returned. Again, the descendant-or-self axis should be used to locate all subparts relative to a part.

Some XPath expressions fundamentally depend on a direction and cannot be augmented with a closest axis. An example would be that a query that to find the names of all the elements that enclose a book element, a parent axis would be necessary to locate the enclosing element.

4.7 Related Work

By facilitating the symmetric exploitation of hierarchical data, this chapter contributes to the following areas. First, it is a means for the integration of XML data, because heterogeneous data can potentially be queried with the same query. Second, it offers a novel (not only syntactically but also semantically) construct to XML query language. A user can effectively query XML data without knowing its specific structure. We review related work in these two areas respectively.

Many research projects have focused on the problem of data integration [13] [24] [26] [42]. The goal of data integration is to combine data from different sources into a single
source. From a user’s point of view, there is only one source and she can query the data using the schema of that single source. For example, YAT [13] and SilkRoute [24] translate relational data into XML. Query over the underlying relational data is expressed through an XML interface. The heterogeneity considered in these data integration systems largely lies in the form of the data, e.g., relational and XML data. This chapter considers XML data only, but with heterogeneous structures.

Data integration systems are usually classified as global-as-view (GAV) and local-as-view (LAV). GAV means that there is a global view, defined as a view over local schemas. In contrast, a LAV approach defines local views in terms of the global schema. Most data integration solutions are GAV, with just a few (for example [42]) being LAV. Since our approach is structure-independent, there is no schema or view visible to a user at all. However, the philosophy of symmetric exploitation can in some sense be regarded as a GAV data integration approach. The notion of a global schema is manifested by the non-directional nature of the closest axis. This schema is essentially a virtual graph. In this graph, each directed edge represents the fact that the destination node is in the closest axis of the source node. None of the (undirected) tree edges needs to be represented in this graph. Such a graph is a special kind of global schema in that the user is not even expected to be aware of it.

Querying hierarchical data is often a non-trivial task. There has been some work on the convenient formulation of queries over XML data. For example, [35] and [47] propose descriptive languages for specifying transformations of XML data. Similar to the closest
axis, these languages hide from users much of the procedural specification necessary in a language such as XQuery or XSLT. However, these special-purpose techniques are limited because they still suffer from being structure-dependent. A query might have to be rewritten when the data changes. Our objective to flexibly issue queries independent of the structure is shared by [16] and [38]. [16] presents a semantic search engine for XML. The search relies on an interconnection relationship to decide whether nodes are “semantically related.” Two nodes are interconnected if and only if the path between them contains no other node that has the same label as the two nodes. [38] proposes a schema-free XQuery, facilitated by a Meaningful Lowest Common Ancestor Structure (MLCAS) operation. Both the interconnection in [16] and the MLCAS in [38] are similar to the “closest” relationship between nodes in this chapter. However, the closest axis is more flexible due to the use of node type. Reasoning solely on node label can lead to more counterintuitive results. With types, a query can be much richer in semantics and can thus produce desirable results more easily.

4.8 Conclusion

XPath suffers from a lack of symmetric exploitation in path expressions. Path expressions in XPath are asymmetric because they are enmeshed in the structure of a hierarchy to navigate to desired data. Asymmetric path expressions are brittle and have a tendency to break when the hierarchy evolves or when the expression is applied to a new hierarchy with a different structure. This chapter proposes a new axis, which we call the closest axis, that can be used to exploit data symmetrically. The closest axis contains nodes that
are closest to the context node, where closeness is measured as the distance from the context node in any direction. Unlike other axes, the closest axis is non-directional. So though the structure of the data may vary, the nodes in the closest axis for a given context node remain closest. We described the syntax and semantics of the closest axis. We also showed how the closest axis can be efficiently implemented for both in-memory and persistent XPath/XQuery evaluation engines. The key to efficient implementation is to use type information to quickly find a path that leads to a closest node. We introduced an LCA-join operation to compute such paths in a persistent implementation. Finally, we showed how some XQuery Use Cases could be rewritten using the closest axis. Though the closest axis does not make the queries significantly shorter, the same queries can be evaluated over heterogeneously structured hierarchies.

Much still remains to be done. Though we have specified the semantics of the closest axis and sketched efficient algorithms to evaluate it, we have yet to implement the axis in a product. We plan to test two implementation strategies using eXist, an open source native XML DBMS. One strategy will utilize a pre-processor to convert non-directional queries into directional queries. This will demonstrate that the closest axis can be implemented as a layer, at low cost. The second strategy will modify the internals of eXist to implement the LCA-join. A benchmark comparison of the two techniques will help to determine the effectiveness of the LCA-join. The LCA-join also has applications in restructuring data, where data is transformed from one hierarchy to another. Another avenue of future work is to define a complete set of non-directional axes. We speculate that there exist other non-directional axes that involve conditions expressed on labels and types. We are also
working on a functional query language called *PathFree* that will entirely eliminate path expressions since the closest axis can be used to locate data and a related technique can restructure data.
5. SCHEMA-LESS, SEMANTICS-BASED CHANGE DETECTION

Schema-less change detection is the processes of comparing successive versions of an XML document or data collection to determine which portions are the same and which have changed, without the knowledge of a schema. Change detection can be used to reduce space in a historical data collection and to support temporal queries. Most previous research has focused on detecting structural changes between document versions. But techniques that depend on structure break down when the structural change is significant. This chapter develops an algorithm for detecting change based on the semantics, rather than on the structure, of a document. The algorithm is based on the observation that information that identifies an element is often conserved across changes to a document. The algorithm first isolates identities for elements. It then uses these identities to associate elements in successive versions.

5.1 Introduction

The Extensible Markup Language (XML) is a popular format for data exchange and document publishing over the Internet. It is rapidly becoming the lingua franca of the Web [18]. As documents and data on the Web are modified over time, it is important to identify in different versions of an XML document what changes and what remains the same.

Change detection is a process that identifies changes between successive versions of an XML document or data collection. At the logical level, change detection aids in
understanding the temporal behavior of data. Putting data into the context of its evolution usually entails more meaningful information. This is particularly true of data on the web, which often has a high rate of change [7] [11]. At the physical level, change detection helps an archival system reduce space by storing only the changes to a new version because the size of the change is generally a small fraction of the entire version. In systems where storage is not a concern but data is shipped across a network, change detection can reduce network traffic, since often only the changes, and not the entire document, can be transferred.

Change detection is also significant in temporal query support and incremental query evaluation. In contrast to queries that are issued against a single version of the data, temporal queries can involve both current and past versions [28] [25]. Temporal queries that access more than one version are important in mining historical facts and predicting future trends. However, the semantics of many temporal queries depends on identifying which data has changed and which has continued to reside unchanged in a database. Incremental query evaluation, when applicable, can significantly reduce the cost of query evaluation. Some continuous (and non-continuous) queries can be incrementally evaluated, just by using the change. A continuous query is re-evaluated when new data arrives, for instance, as the query reads from a data stream [39] [40]. For certain types of queries it is sufficient to combine the result of the query on the changed data with the previous result, rather than re-evaluate the query on the entire database, e.g., as in semi-naïve query evaluation in datalog.
This chapter is organized as follows. Section 5.2 presents an example that motivates this research. We show that when the structure of a document changes while the data remains the same, a new method is needed to associate nodes in successive versions of a document. Related work in structural change detection is presented in Section 5.3. Section 5.4 formalizes the notion of semantic identities for XML and outlines an algorithm for computing identities. Next, the chapter shows how to use the semantic identities to detect changes in XML by matching elements based on their identities. A match means that an element has retained its semantic identity across successive versions. The algorithm has been implemented in Java, and we report on experiments in Section 5.6. The chapter then concludes with a discussion of future work.

5.2 Motivation

Figure 35 shows a bibliography XML document. Figure 36 shows the next version of the document. The change from the old version to the new version can be effected in two steps: (1) an update operation — a modification of the book price from the list price in the old version to the sale price in the new version (note that the element enclosing the price is also renamed), and (2) an insertion operation – the addition of an ISBN element and text content. These changes are reflected in Figure 35 where the element that is replaced is underlined and in Figure 36 where the insertion is highlighted in a bold font. The rest of the new version is an exact copy of the old version.
Measuring how much of a document has changed can be useful. One metric of change is the *minimum edit distance* between two versions [10]. A sequence of operations that minimally transforms one document version to the next version is called a *minimum cost edit script*. The script is a sequence of *insertion*, *deletion*, and *update* operations. It is common to consider the cost of each operation to be the same, e.g., each operation costs one unit. In the example, the minimum cost edit script consists of two updates and two insertions (an element and its content are considered as separate insertions or deletions).

But consider a different, more substantial change. Figure 37 shows an alternative version of the document fragment in Figure 35. It is an alternative version in the sense that it has basically the same information, just arranged in a different schema. The version in Figure 35 has books grouped by author. The version in Figure 37 is books grouped by publisher. What has changed from Figure 35 to Figure 37? The two fragments are far...
from identical. In fact, the cost to construct an exact copy of this new version from the old one is quite high. Intuitively, it requires a significant number of element insertions and deletions. The high cost of the edit script can be somewhat alleviated by the introduction of a new kind of atomic operation, subtree move [11] [10]. If an element (e.g., a subtree rooted at the element node in the data model created when a document is parsed) in the old version has changed its position but not its content, then the many individual deletions and insertions can be counted as just one subtree move. In our example, the minimum cost edit script can be described by two such moves: (1) move the <author> node along with its <name> subtree to appropriate locations under both <book> nodes, and (2) move each <publisher> node together with its text child above the appropriate <book> node.

```
<publisher>Doubleday
  <book>
    <title>The Da Vinci Code</title>
    <author>
      <name>Dan Brown</name>
    </author>
    <listprice>$24.95</listprice>
  </book>
</publisher>
<publisher>Pocket Star
  <book>
    <title>Angels & Demons</title>
    <author>
      <name>Dan Brown</name>
    </author>
    <listprice>$7.99</listprice>
  </book>
</publisher>
```

Figure 37 An alternative new version of the bibliography document
Note that only one piece of author name information (<name> and its content) exists in the old version, while the new version has one <author> in each <book>. On the other hand, there is a <publisher> element for each publisher in the new version while in the old version there may be multiple <publisher>s with the same text content. Such one-to-many, many-to-one, and potentially, many-to-many, subtree moves are not usually supported.

The high cost of change in this particular situation, however, is not our concern. Regardless of how large the change is, the cost is always an order of the size of the document. The problem lies in the fact that significant structural change makes it difficult to associate elements in different versions. An association is a correspondence between elements in successive versions. The association establishes that an element in one version has a successor in the next version of the document. The association is possibly a new version of the element. For instance, the Pocket Star <publisher> element in Figure 37 should be associated with the Pocket Star <publisher> element in Figure 35; ostensibly, it is a new version of that element. But each <publisher> is structurally very different and therefore cannot be associated by change detection processes based on recognizing structural copies. This creates an obstacle for temporal query or incremental query evaluation. If we are unable to appropriately associate elements in successive versions, these applications cannot be implemented correctly. The example sketched above suggests that if the structural change is significant, even though semantically the same data is present, it becomes difficult to associate elements.
At issue is how to define “change.” Previous research considered two documents, or portions thereof, the same if and only if they are identical (see Section 5.3 for a review of previous research). This requires not only textual equality but structural as well. Some researchers have considered unordered trees as the basis of comparing versions, which unlike the XML data model, ignores order among siblings.

This chapter proposes a semantics-based change detection framework. Nodes that are semantically equivalent are considered unchanged and will be associated, regardless of their structural contexts. Finding all such associations between two versions has two important benefits: (1) it allows elements to exist across successive versions of a document (thus providing good support for temporal queries, for example), and (2) it detects semantic changes (which also includes “easier” changes, i.e., those that do not involve significant amounts of structural change).

5.3 Related Work

Early work in change detection mainly deals with string matching [36] [43] [53]. The subject is thus “flat” (plain-text) documents without hierarchy structures like those in SGML or XML. GNU’s `diff` [27] is probably the most widely-known tool for finding differences between flat documents. It is based on a technique called the D-Band algorithm [46]. Another change detection application from GNU is CVS [17]. It uses `diff` to detect changes in two versions of a program. Plain-text change detection does not work well for XML because it is insensitive to hierarchy structure. It is also poorly-suited to finding semantic change, which is the goal of this chapter.
There has been some research in change detection for HTML prior to the emergence of XML [19] [20]. HTML is syntactically similar to XML (e.g., a document is a properly-nested collection of elements). However, techniques for HTML are not entirely suitable for XML since elements in HTML are usually for presentation while elements in XML serve as metadata. In structural XML change detection, data enclosed in different elements cannot be matched. Change detection for general hierarchy structured documents has been researched [9] [10]. To achieve efficiency, some simplifying assumptions are made. For example, [10] assumes that any leaf in one version has at most one leaf in another version “close” enough to be its match. This may not be true for such XML documents, as illustrated in Figure 37.

In almost all research on XML, the data model is considered or presumed to be a tree. It is thus natural to make use of the results from tree matching (sometimes also called tree correction) [32] [41] [49] [51] in XML change detection. It is important to point out that most research adopts an ordered, labeled tree data model [11]. In an ordered tree, order among siblings is important. In contrast, only parent-child relationships are important in an unordered labeled tree. The best known algorithm for general ordered labeled tree matching is by Zhang and Shasha [64]. To the best of our knowledge, only a few papers have considered the scenario where XML is modeled as an unordered labeled tree [54] [66], in part because the tree matching problem for unordered labeled trees is NP-complete [65]. Due to space limitations, we omit presenting the complexities of the algorithms mentioned above since they all consider structural equality for a match while we focus on semantic equivalence. A match defined by our approach might not be an
exact copy, and even exact copies may not match (for instance two different <author>s that happen to have the same <name> should not necessarily have their names match).

Our semantic change detection technique is based on finding a (semantic) identity for each node in an XML data model. An identity is like a key for XML, but does not play a role in validation. XML Schema permits keys to be defined for elements [60]. A document will fail validation if the key does not uniquely identify each element (of a particular element type). A more primitive solution is offered in the XML recommendation. As part of a DTD an element can be designated as having an ID attribute [58]. The value of the ID attribute must be unique for every element in a document. When a validating parser parses the document, the document will be deemed invalid if the uniqueness constraint is violated. Buneman et al. define a key as the pairing of a target path with a set of key path expressions [8]. Both the target path and key path expressions are regular expression queries (similar to XPath queries) on the data model for the document. The target path specifies a set of nodes corresponding to elements in the document, called the target set. Typically, the target set is a collection of nodes for elements that share the same name (e.g., the set of <book> elements). The key path expressions form the key for nodes in the target set. The key path expressions must evaluate to a different result for each node in the target set. Our work on identities differs from Buneman et al.’s research in four ways. First, we use a path list rather than a path set since the order of the expressions in an identity is important. Second, it is possible for the identities of two nodes to evaluate to the same value (when they have the same semantics), whereas a “key” always has a unique value. Third, we provide a method to
automatically compute identities. Fourth, we focus on using identities to associate nodes across multiple versions of a document’s data model.

5.4 Semantic Identities

This section develops an algorithm for computing semantic identities. Basically, an identity is a query expression that can be used for element identity, that is, to distinguish one element from another. We show how semantic identity is related to identity based on structure alone. Some elements can have exactly the same structure, yet be semantically different.

We adopt an ordered, labeled tree data model for XML, as defined in Definition 23 on page 94. As mentioned before, this data model only considers element and text nodes in the DOM XML model. Document, comment and processing instruction nodes can be regarded as special kinds of element nodes which cannot have any children. An attribute node can be treated as a subelement (a child element node), with the name of the attribute serving as the name of the child element and the value of the attribute serving as the text content of the subelement. We assume that the data model is created when an XML document is parsed, or is provided by an XML data management system for query purposes.

In a given data model, each element and text node is of a specific type, as defined in Definition 11 on page 53. In this chapter, we will denote this type as $T$. The type is the concatenation of labels (element names) on the path from the root to a node, separated by a ‘/’. A text node and its parent (an element node) will have the same type, but they are
distinguished by the fact that they are different kinds of nodes. For example, in the data model of the document shown in Figure 35, the text node “Doubleday” has text type: author/book/publisher, while its parent has element type: author/book/publisher. We will refer to the final label in the type as the abbreviated type. The abbreviated type in the example given above is publisher. It suffices to use the abbreviated type except when two types happen to have the same abbreviation.

Finally, we assume that the immediate text content of an element is a single text node. Multiple text children are merged to a single text node whose value is the concatenation of the text content of each child, without affecting the semantics for our purposes.

We do not assume that a schema, such as a DTD or XML Schema document, is available; only the data model exists. This does not mean that a schema is useless. Additional schema information is certainly helpful in understanding semantics and detecting changes. However, our goal is to develop a general strategy for change detection that can handle any well-formed XML document.

5.4.1 Structural Identity

The structure of an element node is its type and the set of all its descendants (assuming lexical order is not important, a list of descendants otherwise). We will consider a pair of elements to be structurally different if the elements are of different types or if there is some descendant of one element that is structurally different from every descendant of
the other element. Otherwise, the elements are considered to be *structurally identical*.

Consider the two nodes, $x_1$ and $x_2$, in Figure 38. Assume that the two nodes are identical copies, that is, the nodes are of the same type, and the subtrees below the nodes (represented by the triangles) are the same (modulo reordering of the children). Then $x_1$ and $x_2$ are structurally identical.

![Figure 38 Structurally identical nodes](image)

5.4.2 *Relating Semantic to Structural Identity*

This section develops a notion of semantic identity as different from, but related to, structural identity. We give two axioms that capture the intuition with which we reason about semantic identity. Thus these axioms serve as a bridge that connects intuitive understanding to the rigor necessary for computer processing.

**Axiom I**: Nodes that are structurally different are semantically different. □

Let’s consider text nodes first, and then element nodes. The structure of a text node is its type and value. Axiom I states that two text nodes are considered to be different, semantically, when their structures are different. As an example, consider the document in Figure 37. The text nodes corresponding to “Angels & Demons” and “The Da Vinci
“The Da Vinci Code” are different in semantics because they are textually different. But the text nodes corresponding to “Dan Brown” are structurally identical, and therefore could be semantically identical. Each <book> element has <title>, <author> and <listprice> subelements. Since the two text children of title nodes are “The Da Vinci Code” and “Angels & Demons,” the two <book> nodes are semantically different regardless of the <author> or <listprice> nodes. (In Figure 37 their prices are different. But even if they were the same, the two <book> nodes would still be semantically different.)

If two element nodes are semantically different, it is not necessarily the case that they are structurally different. In fact, it is possible for two element nodes to be structurally identical but semantically different. This is stated in the following axiom.

**Axiom II:** Nodes that are structurally identical are semantically identical if and only if their respective parents are semantically identical, or if they are both root nodes. □

Axiom II states that nodes that have the same structure also have the same semantics if and only if their parents are semantically the same. Axiom I distinguishes nodes that have different content; but when nodes have exactly the same content, Axiom II offers an alternative to distinguish them by their context. The context is the semantic identity of the parent node. Consider the structurally equivalent nodes, \( x_1 \) and \( x_2 \), in Figure 38. The two nodes can only be identified by their parents, \( p_1 \) and \( p_2 \), respectively. \( x_1 \) and \( x_2 \) are semantically identical if and only if \( p_1 \) and \( p_2 \) are semantically identical.
For example, the two `<name>` nodes in the data model for the XML document in Figure 37 both have a text child “Dan Brown” and are thus structurally equivalent. Are they semantically equivalent? It depends on their context. If we inspect their parents’ semantics, we find that the two `<author>` nodes are structurally different (in the `<book>` subelement, or similarly, in `<listprice>`), and so by Axiom I are semantically different. Therefore the two `<name>` nodes are structurally identical but semantically different since each is in the context of a different book.

If two structurally equivalent nodes have semantically identical parents, then they are regarded as identical. This is reasonable because we cannot semantically distinguish two exact copies when they are enclosed in the same context. Lemma I below pictures the special case where such nodes share a common parent.

**Lemma I:** Structurally identical siblings are semantically identical. □

If siblings are structurally identical, then semantically they are redundant copies of the same information. We can keep just one copy, removing the duplicates, without semantic loss, although we might want to keep a count of how many redundant copies there are.

The semantic identity relation induced by these two axioms is reflexive, commutative and transitive. Moreover, the axiom system that consists of these two axioms and appropriate inference rules is sound and complete. The proof is omitted due to space limitations.
5.4.3 Identities

This section defines several terms that are important to the algorithm for semantics-based change detection presented in the next section.

An identity is based on the evaluation of XPath expressions so we first define what it means to evaluate an XPath expression.

**Definition 28** [XPath evaluation] Let $\text{Eval}(n, E)$ denote the result of evaluating an XPath expression $E$ from a context node $n$. Given a list of XPath expressions, $L = (E_1, \ldots, E_k)$, then $\text{Eval}(n, L) = (\text{Eval}(n, E_1), \ldots, \text{Eval}(n, E_k))$. □

Since an XPath expression evaluates to a list of nodes, $\text{Eval}(n, L)$ evaluates to a list of lists.

**Definition 29** [identity] An identity for a type, $T$, is a list of XPath expressions, $L$, such that for any pair of type $T$ nodes, $x$ and $y$, $x$ and $y$ are semantically different if and only if $\text{Eval}(x, L) \neq \text{Eval}(y, L)$. □

An identity serves to distinguish nodes of the same type. Two nodes are considered semantically the same if and only if their identities evaluate to the same result. Two lists are considered equivalent if they have the same cardinality and are equal at each position.
The choice of XPath expressions as the basis for specifying an identity is a means rather than an end. It could be any mechanism that is able to properly locate nodes in a data model. We use XPath since it is widely adopted and supported.

**Definition 30** [identity map] An identity map (denoted $M$) is a relation that maps each type to its corresponding identity (an XPath expression list), i.e.,

$$M = \{(T, L) \mid L \text{ is an identity for type } T\}.$$

Identities are constructed from XPath expressions that locate values (text nodes) in the subtree rooted at a node.

**Definition 31** [type-to-leaf path list] The type-to-leaf path list for a type, $T$, denoted $\text{typeL}(T)$, is a list of XPath expressions such that $\text{typeL}(T)$ is a sorted list of XPath expressions, $\text{sort}(S)$, where

- $E = \{e \mid e \text{ is of type } T\}$ is a set of all of the elements of type $T$ in a document,
- $D = \{d \mid d \text{ is a text descendant of some } e \in E\}$ is a set of all of the text descendants of type $T$ elements (if $T$ is a text type then $\text{self()}$ is the only descendant), and
- $S = \{s/\text{text()} \mid s = \text{suffix}(T, t) \text{ where } t \text{ is the type of some } d \in D\}$ is a set of all of the relative XPath expressions that locate a text descendant from a context node of a type $T$ element. (Note that if $T$ and $t$ are the same then $S$ includes $\text{text()}$.)
A type-to-leaf path list is a specific list of XPath expressions that locate text values that are descendants of nodes of a particular element type. In a given XML document, each type $T$ has exactly one $typeL(T)$. In the document shown in Figure 35, for example,

$$typeL(author/book/title) = (\text{text}())$$

and

$$typeL(author/book) = (title/text(), publisher/text(), listprice/text()).$$

Note that for the exact document shown in Figure 35, $typeL(book)$ should contain one more XPath expression — $\text{text}()$. We believe that trivial text nodes, i.e., text nodes whose contents are all white-space characters, are insignificant in semantics. Thus the expression $\text{text}()$ appears in a type-to-leaf path list $typeL(T)$ only if there exists at least one type $T$ node with non-trivial text children.

**Definition 32** [unique with respect to $typeL(T)$] Suppose that node $n$ is of type $T$. Then $n$ is unique with respect to $typeL(T)$, if and only if $typeL(T)$ is an identity for type $T$. That is, if and only if for any $n'$ of type $T$,

$$\text{Eval}(n, typeL(T)) \neq \text{Eval}(n', typeL(T)). \quad \square$$

In the above example, since $\text{text}()$ evaluates to different results from the two title nodes, they are unique with respect to $typeL(author/book/title)$. 


5.4.4 Computing Identities

The algorithm to compute identities will operate in a bottom-up fashion, working from the leaves towards the root of the tree. The following definitions describe positions in the bottom-up traversal. We use the term “floor” to evoke the idea that the algorithm ascends the tree like a person might climb floors in a building.

Definition 33 [floor-0 node] All text nodes and only text nodes are floor-0 nodes.

Definition 34 [floor-k node] A node is a floor-k node if and only if the maximal floor among its children is k-1.

All text nodes are floor-0 nodes, and vice versa. Each element node is at least floor-1, and its exact floor number depends on its children’s floors. Here is an intuitive idea of what “floor” is. Text nodes are childless. They are at the “bottom” in the data model tree, at floor-0. Every text node has a parent element node. If this node does not have element children, then it is the lowest element, and hence at floor-1. In general, an element node’s floor is determined by adding one to the maximum of its children’s floors.

Figure 39 shows the data model for the document in Figure 35. The circles represent element nodes and the rectangles represent text nodes. The floor of each node is shown as the label inside the circle or rectangle.
If all of the elements of type $T$ are floor-1 nodes, then $typeL(T) = (text())$; for example, \(typeL(\text{author/book/title}) = (text())\).

Above floor one, the XPath expressions in a type-to-leaf path list become more complex; for example, \(typeL(\text{author/book}) = (\text{listprice/text()}, \text{publisher/text()}, \text{title/text()})\).

The three XPath expressions in $typeL(\text{author/book})$ are sorted alphabetically. But as long as all type $T$ nodes have the same $typeL(T)$, any order will do.

Note that not all nodes of a certain type are of the same floor, and vice versa. Types are computed top-down while floors bottom-up in the data model tree. Both concepts are important in computing identities.
**Definition 35** [local identity] An identity is a *local identity* if the XPath expressions evaluate to descendants of the context node. Otherwise the identity is said to be non-local.

An identity contains XPath expressions that evaluate to some leaf nodes in the document tree. It is either a local identity or a non-local identity. A non-local identity locates at least one leaf node that is not a descendant of the context node. For example, the <name> node in Figure 1 is identified by its text content, Dan Brown; so `text()` is a local identity for <name>. On the other hand, the two <name> nodes in Figure 3 have identical contents. It is impossible for the identity of this type to contain only descendants of the <name> nodes. Thus <name>’s identity must be non-local.

The algorithm for computing identities is shown in Figure 40. The algorithm consists of two phases. Phase 1 finds all local identities, working bottom-up from the floor-0 nodes. This phase corresponds to Axiom I. When Phase 1 terminates, all semantically distinct nodes that Axiom I can determine are found. Phase 2 recursively computes the identities for the remaining types. This corresponds to Axiom II. When Phase 2 terminates, all semantically distinct nodes are found. Any remaining node is a redundant copy of another node in the document.
Pre-condition:
\[ M = \{(T_k, \{\}) \mid 1 \leq k \leq \text{number of different types}\} \]

Post-condition:
\[ M = \{(T_k, L_k) \mid L_k \text{ is a identity for type } T_k\} \]

Computing identities:
Phase 1: find local identities
1) \( i = 0; \)
2) For each floor-\( i \) node \( n \) of type \( T \) such that \( M(T) = \{\} \), if every type \( T \) node is unique with respect to \( typeL(T) \), add \((T, typeL(T))\) to \( M \) and for each type \( T' \) that is a prefix of \( T \), add \((T', \text{suffix}(T', T)/typeL(T))\) to \( M \);
3) \( i = i + 1; \) terminate Phase 1 if the next floor is the root, or go to 2).

Phase 2: expand with non-local identities
Starting from the root and working down the tree, for each node \( n \) of type \( T \) such that \( n \) is not unique with respect to \( typeL(T) \), add \((T, Id)\) to \( M \) where \( Id \) is a list obtained by appending to \( typeL(T) \) the identity of \( n \)'s parent.

Figure 40 Algorithm for computing identities

5.4.5 Computing Identities for a Sample Document

We will now illustrate how the algorithm works through an example. Figure 41 shows a sample document bib1.xml. The data model for this document is depicted in the lower left part of Figure 43. The intuitive semantics for this document is that author n2 writes a book t2 published by p2, and also collaborates with author n1 on a book t1 published by p1.
Phase 1 starts with floor-0 nodes; there are eight floor-0 (text) nodes, two of type name, three of type title and three of type publisher. The typeL(\mathcal{T}) for each of the three text types is (self()). Neither title nor publisher is unique on the respective typeL(\mathcal{T}). Each name however is unique for the typeL(name) = (self()). This step then examines the name (element node) parent of each name text node. A name’s identity becomes (text()). The algorithm then proceeds to the parent of each name element node, which happens to be an author. The identity for the type author becomes (name/text()).

Eventually, Phase 1 terminates without finding any more identities. Phase 2 finds identities for title, publisher and book. Since we have already found the identity for author, we only need to look for their identities within two separate trees rooted at the
two author nodes. Within each subtree, the local identity for title (and publisher) is (text()). The local identity for book depends on whether title or publisher is the first to update book. Suppose title does, then book’s local identity is (title/text()). We now have the local identities in these two subtrees. The identity is then constructed by appending author’s identity to the local identity. The final result is shown in Table 3. (Only element identities are shown due to space limitation.) The evaluation of an identity for a type will produce a unique list of values for each node of that type. For example, the identity for the three publisher nodes evaluate to, in document order: ((n1), (p1)), ((n2), (p2)) and ((n2), (p1)). What this means is that a publisher in bib1.xml can be uniquely identified by its local identity combined with the identity of its parent.

<table>
<thead>
<tr>
<th>Type</th>
<th>Identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>(text())</td>
</tr>
<tr>
<td>author</td>
<td>(name/text())</td>
</tr>
<tr>
<td>title</td>
<td>(.../author/name/text(), text())</td>
</tr>
<tr>
<td>publisher</td>
<td>(.../author/name/text(), text())</td>
</tr>
<tr>
<td>book</td>
<td>(.../author/name/text(), title/text())</td>
</tr>
</tbody>
</table>

### 5.4.6 Complexity Analysis

The cost for Phase 1 depends on the value of \( i \) when the algorithm terminates. Suppose that there are \( n \) leaves and \( f \) floors. Then there are \( n \) floor-0 nodes. In the first iteration, suppose there are \( k \) different types of floor-0 nodes, and \( v_1, \ldots, v_k \) nodes of each type, so
\[ n = v_1 + \ldots + v_k. \] We need to determine if a node of the \( i^{\text{th}} \) type is unique in the \( v_i \) nodes of the same type. Since uniqueness could be determined by sorting, the cost of this iteration is bounded by \( v_1 \log_2(v_1) + \ldots + v_k \log_2(v_k) \), and hence by \( n \log_2(n) \) (assuming an in-memory sort; an external sort would be more expensive).

If the first phase does not terminate after the first iteration, then the complexity of the second iteration is bounded by \( (n/2) \log_2(n/2) \) because there are at most \( n/2 \) floor-1 nodes. In general, the worst-case complexity is bounded by

\[ n \log_2(n) + \ldots + (n/2^{f-1}) \log_2(n/2^{f-1}), \]

which is in turn bounded by \( 2n \log_2(n) \).

Phase 2 determines non-local identities by appending to local identities. No comparison of values is involved. The complexity of Phase 2 is thus bounded by the number of types (which in the worst case is the number of nodes).

The total cost of the algorithm is therefore bounded by \( m + 2n \log_2(n) \), where \( m \) is the number of internal (element) nodes and \( n \) is the number of leaves. Since \( m < 2n \), the overall complexity is \( O(n \log(n)) \).
5.4.7 Minimal Identities

Notice that the algorithm generates an identity but does not guarantee a minimal identity. For example, even though \((a/text(), c/text())\) would be enough to identify \(<x>\) elements in the following XML fragment

\[
\begin{align*}
<x><a>1</a><b>1</b><c>2</c></x> \\
<x><a>2</a><b>1</b><c>1</c></x> \\
<x><a>2</a><b>1</b><c>2</c></x>
\end{align*}
\]

the algorithm would produce the identity

\[(a/text(), b/text(), c/text()).\]

To achieve minimal identities, we need to potentially inspect \(2^k - k - 1\) different possibilities where \(k\) is the size of the corresponding type-to-leaf path list. In this example, four possible combinations need to be checked. Our algorithm does not attempt to find minimal identities as the worst-case complexity is exponential.

5.5 Semantic Change Detection

We are now able to uniquely identify a node in an XML document. In this section, we discuss how nodes in different versions can be matched based on their identities. Once all semantically identical nodes are matched, we regard the unmatched elements as change participants.

5.5.1 Semantic Node Matching

The following definitions assume that
element nodes \( p \) and \( q \) are both of type \( T \) and reside in different versions \( V_p \) and \( V_q \) of an XML document,

\( \cdot \) \( \text{ID}(p) \) is \( p \)'s identity,

\( \cdot \) and \( \text{ID}(q) \) is \( q \)'s identity.

**Definition 36** [type territory] The territory of a type \( \mathcal{T} \), denoted \( T_{\mathcal{T}} \), is the set of all text nodes that are descendants of the least common ancestor, denoted \( lca(\mathcal{T}) \), of all of the type \( \mathcal{T} \) nodes. □

Within the type territory is the territory controlled by individual nodes of that type.

**Definition 37** [node territory] The territory of a type \( \mathcal{T} \) node \( p \), denoted \( N_p \), is \( T_{\mathcal{T}} \) excluding all text nodes that are descendants of other type \( \mathcal{T} \) nodes. □

The type territory of \( \mathcal{T} \) contains all of the information that is possibly useful in identifying any type \( \mathcal{T} \) node. The territory of a specific node of that type is contained in the type territory, but does not contain any node that is a descendant of another type \( \mathcal{T} \) node.

Figure 42 visualizes the idea of type territory and node territory. Suppose there are three type \( \mathcal{T} \) nodes, \( p, p' \) and \( p'' \). The type territory of \( \mathcal{T}, T_{\mathcal{T}} \), is the subtree rooted at the node \( lca(\mathcal{T}) \). \( N_p \), that is, the node territory of \( p \) is the area shaded dark in the figure. \( N_p \) is \( T_{\mathcal{T}} \) excluding the two subtrees rooted at \( p' \) and \( p'' \) (represented by the striped areas).

The type territory of *book* in Figure 5, for example, is (*Dan Brown, The Da Vinci Code, Doubleday, $24.95, Angels & Demons, Pocket Star, $7.99*). The
node territory of the leftmost book node is (Dan Brown, The Da Vinci Code, Doubleday, $24.95).

Now we are ready to match nodes in successive versions.

**Definition 38** [admits] $q$ admits $p$ if $\text{Eval}(q, \text{ID}(q)) \subseteq N_p$. □

In general, $\text{Eval}(p, \text{ID}(p))$ is a list of lists because each XPath expression in the identity evaluates to a list of values. Here in semantic matching we implicitly convert (flatten) it to a list of distinct values. This is because only the values are important in our semantic matching.

**Definition 39** [node match] Nodes $p$ and $q$ are matched if and only if $p$ and $q$ admit each other. □
Intuitively, admission and match can be described as follows. $q$ is identified by a list of text values $q_1, \ldots, q_n$ in $V_q$. If a node $p$ in $V_p$ has at least as much information as $q$ does, then $p$ should have a group of text values $q_1, \ldots, q_n$ in its own territory $N_p$. A match implies semantic equality between two nodes, thus it requires admissions in both directions.

5.5.2 Semantic Matching for Sample Documents

We now show how nodes are matched based on the criteria described above. To save space, we will display XML documents in tree diagrams rather than their textual form. This makes possible side-by-side document comparison and the representation of node match as well.

Figure 43 shows bib1.xml (from Figure 25) and the next version, bib2.xml, in which there has been significant structural change. The change is similar to that presented in Figure 35 and Figure 37. Intuitively, we can tell that the semantics of the two versions are the same insofar as they contain the same data, but arranged to different schemas. We will now illustrate how to match nodes to associate elements in different versions.

First we need to compute the identities for each node in both versions. Identities for bib1.xml are already available in Table 3. Identities for bib2.xml can also be computed by the algorithm in Figure 40 and are shown in Table 4.
We then need to evaluate the value of each node’s identity. For example, the values of 
<book>’s identities in both versions are shown in Table 5. The territory of the leftmost 
<book> in bib2.xml is (p1, t1, n1, n2, p2); the identity of the leftmost <book> in 
bib1.xml evaluates to (n1, t1). Hence the leftmost <book> in bib1.xml admits the 
leftmost <book> in bib2.xml. Similarly, the first <book> in bib2.xml admits the 
first <book> in bib1.xml. Therefore, the first <book> in bib1.xml and the first 
<book> in bib2.xml match. This is represented in Figure 43 by a dashed line 
connecting the two.
All matches for book and author are shown in Figure 43. (Two matched author nodes are connected by a dotted line.) To preserve the clarity of the figure, we do not show matches for all nodes. (We will discuss only book and author nodes in the rest of this chapter because they are representative enough.) In fact, each node is matched to one or more nodes in the other version. For a change (insertion, deletion or update) to occur, there has to be at least one unmatched node in one version. We can thus conclude that there is no semantic change from bib1.xml to bib2.xml.

### Table 4 Identities for bib2.xml

<table>
<thead>
<tr>
<th>Node Type</th>
<th>Identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>(../../title/text().text())</td>
</tr>
<tr>
<td>author</td>
<td>(../title/text().name/text())</td>
</tr>
<tr>
<td>title</td>
<td>(text())</td>
</tr>
<tr>
<td>publisher</td>
<td>(text())</td>
</tr>
<tr>
<td>book</td>
<td>(title/text())</td>
</tr>
</tbody>
</table>

### Table 5 Values of identities for book

<table>
<thead>
<tr>
<th>bib1.xml</th>
<th>Value of Identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>leftmost book</td>
<td>((n1),(t1))</td>
</tr>
<tr>
<td>middle book</td>
<td>((n2),(t2))</td>
</tr>
<tr>
<td>rightmost book</td>
<td>((n2),(t1))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bib2.xml</th>
<th>Value of Identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>leftmost book</td>
<td>((t1))</td>
</tr>
<tr>
<td>rightmost book</td>
<td>((t2))</td>
</tr>
</tbody>
</table>
5.5.3 **Complexity Analysis**

Using the notations in Section 5.5.1, we do the following to decide whether \( q \) admits \( p \).

We first determine the territory of \( p \). To do this, for each element node \( p' \) of type \( T \) other than \( p \), we find the least common ancestor of \( p \) and \( p' \). Each \( lca \) could be found in constant time with a \( O(N) \) time preprocessing, where \( N \) is the size of the tree (number of nodes) [31] [50]. This step takes \( O(N+T)=O(N) \) time (\( T \) is the number of type \( T \) nodes).

Since the preprocessing could be shared by all nodes, the constant factor in \( O(N) \) would remain the same. We now have \( T-1 \) nodes, all of which are \( p' \)’s ancestor. Among them, find the node \( x \) that is nearest to \( p \). This takes \( O(T) \) time using the same technique as above. The territory of \( p \) is the subtree rooted at a node which is (1) a child of \( x \) and (2) an ancestor of \( p \) or \( p \) itself. In total, finding the territory of \( p \) takes \( O(N) \) time.

Suppose \( S \) is the set of leaves in \( T_p \). Our next task is to decide if \( Q=\{q_1, \ldots, q_n\} \) is a subset of \( S \). We first sort each of them in time \( O(n\log_2(n)) + O(|S|\log_2(|S|)) \), and then find out whether \( Q \) is a subset of the \( S \) in time \( |S| \). This is in total bounded by \( L\log_2(L) \), where \( L \) is the number of leaves.

In summary, deciding whether \( q \) admits \( p \) takes a combination of \( O(N) \) and \( O(L\log(L)) \), which reduces to \( O(N\log(N)) \). Thus deciding whether \( q \) and \( p \) match takes \( O(N\log(N) + P\log(P)) \), where \( N \) and \( P \) are numbers of nodes in the two trees.
5.5.4 Whole Document Matching

The above analysis is for a single pair of nodes. In general, we need to match two entire versions. The complexity for matching entire documents is \(O(N^2 \times (N \times \log_2(N) + N' \times \log_2(N')))\) if we try to directly match all possible pairings of nodes in two documents. Though the cost is quite high, there are two strategies to reduce that cost. The first strategy is to only attempt to match a pair of nodes that have the same type, significantly reducing the number of attempted matches. However, our experiments show that the speed of such “direct” matching is still quite slow.

To further reduce the time cost of matching, a second strategy is based on the following heuristic. If the identity for a node type remains the same, then we can base our matching solely on the values of the evaluated identities, effectively skipping the expensive test to search for a match in the node’s territory. The notion of node territory is, in a sense, created to deal with our inability to match nodes when their identities are different; two nodes match if each node’s territory includes the other’s evaluated identity. However, if the identity for a node type remains unchanged, we can in fact base our matching solely on the evaluated identities. In this case, two nodes match if and only if their evaluated identities are the same. There is no need to compute node territories. In most real-world applications, the extent of change over versions is usually small. Hence, it is reasonable to expect that the identities of most of the types remain the same over versions. Based on this assumption, the number of direct matching attempts is often relatively small. For large documents, computing node territories takes a major portion of the processing time.
5.6 Implementation and Experiments

We implemented the semantics-based change detection algorithm in Java. Although a Java implementation will not be as fast as a C implementation, we chose it because it has libraries for XML parsing and DOM building that enable rapid, robust prototyping of applications that process XML. The code is available from our website.\(^8\)

This section describes two experiments that we performed with the prototype. The experiments give insights into the relative cost of the various steps in the algorithm. Experiment one evaluates the cost of matching with structural reordering. In this experiment one document is generated that is a list of \(<\text{book}\>\) elements. Each \(<\text{book}\>\) contains three subelements: a unique \(<\text{title}\>\), an \(<\text{author}\>\) (an author may have written up to seven books), and a \(<\text{publisher}\>\) (there are twenty, different publishers). A second document that contains exactly the same data, but has a different ordering of the \(<\text{book}\>\)s is then generated. Because of the permuted ordering, a structural change-detection algorithm based on ordered tree-matching would be unable to effectively match the pair of documents.

Experiment two evaluates the cost of matching with significant structural change. In this experiment we reused the first document from experiment one. But the second document has the same data with a very different structure. Instead of being a list of \(<\text{book}\>\)s, the second document is a list of \(<\text{publisher}\>\)s. Within each \(<\text{publisher}\>\) is a list of \(<\text{author}\>\)s that wrote a book published by that publisher. Within each author is list of the \(<\text{book}\>\)s

\(^8\)www.eecs.wsu.edu/~cdyreson/pub/ObservantSystems/ChangeDetection
that were written by that author. Structural change-detection algorithms would have difficulty matching these two documents with the same data, but very different schemas.

For each experiment, we generated documents of varying sizes, from 10,000 to 100,000 nodes in order to evaluate how the algorithm performs as the size of the document increases. We also tested the algorithm on how much of the document matches. We performed experiments that match none of the document (0% match) to all of the nodes in the document (100% match). Both experiments were performed on a typical PC with the following configuration: hyperthreaded 2.8GHZ CPU, 2GB SDRAM, Windows XP, and Java (jdk 1.4.2).

Figure 44 shows the result of experiment one. When matching the two documents the semantic identities are the same, so matching uses the heuristic described in Section 5.5.4. The graph shows that the cost of matching is linear in the size of the document.

Figure 45 shows the result of experiment two. In this experiment the identities are different for each document so the node territory has to be searched, resulting in the most expensive kind of match. The running time further increases as a greater percentage of the document matches.
Figure 44 Experiment one matches differently ordered lists of books

Figure 45 Experiment two matches documents with different structures
To further understand the experiments, we calculated the cost of each step in the algorithm. Figure 46 shows the percentage of the total cost for each step in our matching algorithm for the first experiment. The pie chart plots the percentage of the total time for each step, averaged over all of the runs in experiment one. The chart shows that over half
the time is spent in computing the identities (Phase 1 and Phase 2). The actual matching
takes only 2%. The “Other” part of the pie chart is the cost of initializing data structures
used by the algorithm. Since the identities are the same for both documents, a possible
optimization is to skip computing the node territory (which is not used). Another
optimization that could reduce the time spent in Phase 1 and Phase 2 is to reuse the
semantic identity from the first document, possibly storing the computed identities for
future use. Figure 47 shows the analysis for experiment two. The matching is more
expensive, but the cost of Phase 1 and Phase2 could be reduced by storing and reusing
identities, rather than computing them.

5.7 Conclusions and Future Work

This chapter proposes a new approach to detect changes in successive versions of XML
documents. The approach is novel because it utilizes semantic change; previous work
focused on structural change. We first define the notion of a semantic identity, which is
an expression that serves to distinguish elements of a particular type. This chapter then
sketches an algorithm to efficiently compute these identities. Changes in successive
versions are obtained by matching nodes based on their semantic identities. Our
approach is to observe that the information that identifies an element is conserved across
changes to a document. We sketch an algorithm that matches a pair of nodes by looking
for the identifying information of one node in the territory of the second node. The
advantage of our approach is that we can match nodes even when there has been
significant structural change to a document. Compared to conventional structural change
detection, our semantics-based technique is able to detect semantic change in various conditions, without prior knowledge of schema.

The next stage of this research is to refine and extend the implementation by re-implementing in C and integrating the code into the Apache web server to support server-side versioning of XML documents [21]. Another task is to devise a metric to evaluate how good a semantic matching algorithm is. When an algorithm generates associations between nodes in two versions, we should be able to judge the quality of such a result mechanically. With a good framework of semantics-based node association in place, we then plan to build a transaction-time XML repository and implement a system such as TTXPath [22] to support transaction-time XPath queries.
BIBLIOGRAPHY


Proceedings of the 10th International World Wide Web Conference, pp. 201-210, 
Hong Kong, China, May 2001.

Data”. Proceedings of the 1997 ACM SIGMOD International Conference on 

Hierarchically Structured Information”. In Proceedings of the 1996 ACM SIGMOD 
International Conference on Management of Data, pp. 493-504, Montreal, Canada, 
June 1996.

Incremental Crawler”. Proceedings of 26th International Conference on Very Large 
Data Bases, pp. 200-209, Cairo, Egypt, September 2000.

[12] V. Christophides, S. Cluet, and J. Simèon. “On Wrapping Query Languages and 
Efficient XML”. Proceedings of the 2000 ACM SIGMOD International Conference 
on Management of Data, pp. 141-152, Dallas, Texas, USA, May 2000.

Conversion”. Proceedings of the 1998 ACM SIGMOD International Conference on 


[22] C. Dyreson. “Observing Transaction-time Semantics with TTXPath”. In
Proceedings of Web Information Systems Engineering, pp. 193-202, Kyoto, Japan,

Proceedings of the 1st International Workshop on XML Schema and Data
Management, Springer-Verlag, Lecture Notes in Computer Science 2814, pp. 342-


Queries”. Proceedings of the 29th International Conference on Very Large Data

Ullman, and J. Widom. “The TSIMMIS Project: Integration of Heterogeneous


160


APPENDIX
A. Restructuring Algorithm

This restructuring algorithm builds the output forest \( F \) in a top-down fashion, guided by the target signature. First of all, each node in \( F \) is an exact copy of a certain node in the original graph \( G \). Function \texttt{clone()} makes a copy of a node, and function \texttt{addNode()} adds the clone to \( F \). \( F \) is grown by a recursive function \texttt{addChildren()}, the building block of the restructuring algorithm. For each node in \( F \), \texttt{addChildren(parent)} decides which nodes in \( G \) need to be cloned and then adds the new nodes as children of \texttt{parent}. Essentially, the function ensures that 1) every node in \( G \) has been cloned at least once, and 2) two nodes in \( F \) are related only when their counterparts in \( G \) are connected. Thus, \( F \) is guaranteed to have a canonical graph isomorphic to the input graph \( G \).

The body of \texttt{addChildren()} contains three blocks separated by empty lines. These are where the appropriate children of \texttt{parent} are chosen, and we briefly explain what each block does. The first block adds all identifying nodes of \texttt{parent} as its children. This is necessary to ensure that \texttt{parent} and \texttt{parent'} have the same identifier. If the remaining nodes (nodes in the set \texttt{childrenToAdd} after the first block is done) connected to \texttt{parent} are all of the same type, then the second block clones them and adds the clones as children of \texttt{parent'}. Otherwise, the remaining nodes connected to \texttt{parent} are not all of the same type, and the third block deals with this situation. It only clones one node in \( G \) for each remaining type. Why? For example, in \texttt{author.xml} in Figure 1, author n2 is related to two titles and two publishers. Suppose the target signature contains the fragment \texttt{author#(name,title,publisher)}, where title and publisher are non-identifying types. In
such a case, it is incorrect to add the two titles and two publishers as the children of a
single author n2 in the output. Instead, we make two copies of author n2 in the output
forest. Grouping is discussed in more detailed in the next subsection.

• **Input and output specification**

**Input:**
1. a graph $G (V, E, \Sigma, L, C)$
2. a target signature $\text{sig}$

**Transformation:**
- add a node $\text{handle}$ to $G$ so that it is connected to every node in $G$ ($\text{handle}$ has no label or value)
  
  
  
  
  $\text{handle} := \text{clone} (\text{handle})$   \quad // $\text{handle}$ serves as a handle to the generated forest.

- initialize $F (V^{'}, E^{'}, \Sigma^{'}, L^{'}, C^{'})$ such that $E^{'}, \Sigma^{'}, L^{'}, C^{'}$ are all empty sets and $V^{'}) = \{ \text{handle} \}$

- addChildren($\text{handle}$)   \quad // Node $\text{handle}$ may be cloned in $F$ more than once.

**Output:**
- The modified forest $F (V^{'}, E^{'}, \Sigma^{'}, L^{'}, C^{'})$

• **Function specification**

$\text{addNode} (\text{child}, \text{parent})$:

$V^{' := } V^{'} \cup \{ \text{child} \}$

$E^{' := } E^{'} \cup \{ (\text{child}, \text{parent}) \}$

addChildren($\text{child}$)

$\text{clone} (v)$:

create a node $\text{new}$ such that

$L(\text{new}) = L(v)$ and $C(\text{new}) = C(v)$

return $\text{new}$
addChildren(parent):

parent := clone(parent)
childLabels := { label | label node is child of L(parent) node in sig }
childrenToAdd := { v | L(v)∈ childLabels and (parent,v)∈ E }

if ( childLabels = Ø ) // leaf node
    return
else // internal node
    idChildLabels := { label | label node is part of the identifier of L(parent) }
    idChildren := { v | L(v)∈ idChildLabels and (parent,v)∈ E }

while (childrenToAdd≠Ø)
    parent’ := clone(parent)
    V’ := V ∪ { parent’ }

for each idChild∈ idChildren // adding identifying children
    idChild’ := clone(idChild)
    addNode(idChild’, parent’)
    childrenToAdd := childrenToAdd – { idChild }
    addChildren(idChild’)

if nodes in childrenToAdd are of the same type // adding non-identifying children, if of the same type
    for each child∈ childrenToAdd
        child’ := clone(child)
        addNode(child’, parent’)
        addChildren(child’)
else // adding non-identifying children, if of different types
    choose a node new∈ childrenToAdd
    new’ := clone(new)
    childrenToAdd := childrenToAdd – { new }
    addNode(new’, parent’)
    for each label such that label∈ childLabels and label≠L(new)
        pick a node child∈ V such that
            L(child) = label and child, parent)∈ E and
            (child,u)∈ E for every child u (other than new) of parent that has been cloned
        if child has not been cloned before
            childrenToAdd := childrenToAdd – { child }
            child’ := clone(child)
            addNode(child’, parent’)
            addChildren(child’)

168