Polymorphic XML Restructuring

Shuohao Zhang and Curtis Dyreson
Washington State University
PO Box 642752
Pullman, WA 99164, U.S.A.
{szhang2, cdyreson}@eecs.wsu.edu

ABSTRACT

Restructuring is a process that transforms data to a different structure. One can restructure XML data using a query language such as XQuery or XSLT, but formulating the query is sometimes cumbersome and the query may break when the source structure changes. This paper introduces *poly-transform*, a *polymorphic* restructuring algorithm for XML. A poly-transform takes a declarative specification of the target and can be applied to a variety of differently structured sources. We describe the polymorphic restructuring technique and report on experiments that suggest that the technique can be efficiently implemented.

Categories and Subject Descriptors
H.2.3 [Database Languages] Subjects: Query Languages.

General Terms
Algorithms, Languages.

Keywords
XML, Restructuring, Polymorphism.

1. INTRODUCTION
A common XML processing task is restructuring, which transforms data to a different structure. One can restructure XML data using a query language such as XQuery or XSLT. However, there are several complications in restructuring with these languages.

The first issue is *simplicity*. Even for an experienced XQuery programmer, writing a restructuring program is often a complicated task that demands much care and effort. A second issue is *reusability*. A restructuring program in XQuery has limited reusability since it usually needs to be rewritten for a different source structure. A third issue is *efficiency*. The lack of a dedicated restructuring mechanism in XQuery makes it difficult to write efficient restructuring queries. Since many XQuery queries need to restructure data, efficiency is an important concern.

This paper proposes a simple, reusable, efficient restructuring transformation called the *poly-transform*. The poly-transform is *polymorphic* because it can be applied to any source XML data (that shares the same element types), regardless of how that data is structured. It only needs a declarative specification of the target structure. The poly-transform can be efficiently implemented in a single pass over the data.

WWW 2006, May 23–26, 2006, Edinburgh, Scotland.
ACM 1-59593-323-9/06/0005.

Let's consider an example of restructuring. The XML data in Figure 1 lists books authored by E. F. Codd. Two <book>s and a <name> are enclosed within an <author> element. Each <book> has a <title>, <publisher>, and <price>. The structure of the data is ideal for readers to classify their book collections by author. Booksellers, on the other hand, prefer a different structure for the same data. Booksellers work directly with publishers, so they need to classify the books by publisher. Thus, <book>s need to belong to their respective <publisher>s, with the <title>, <author>, and <price> information listed within each <book>. Figure 2 shows the restructured data.

The transformation from Figure 1 to Figure 2 is an example of restructuring. Restructuring changes the structure of the data, but keeps the values and their relationships intact. Typically, restructuring is done with an XSLT template or XQuery program. The XQuery program in Figure 3 will produce the data shown in Figure 2 from the source in Figure 1.

There are complications, however, with programming a restructuring transformation in XQuery. The first is that the program is not reusable in the sense that it will only work for source data that has a particular structure. If the input to the XQuery program in Figure 3 is different in structure than the document in Figure 1 (for instance, the input is Figure 2), then the query will not produce Figure 2. Ideally, restructuring will be polymorphic, that is, it will morph to accommodate the structure of any source that has the same element types. (For simplicity, we use an element's tag as its type; the proposed restructuring technique can be easily modified to accommodate a richer typing system.) A second complication is that programming the transformation is tedious and error-prone. The XQuery program in Figure 3 has a subtle flaw. Suppose that Addison Wesley publishes both books. Then in the target there should be a single Addison Wesley <publisher>. But the XQuery program in Figure 3 will produce two <publisher>s. XQuery's distinct-values function can be used to restructure the data by first grouping publishers in the source document as shown in Figure 4. But even this program is flawed. If there is more than one author for a book, then all of the authors should be listed below the title of the book in Figure 2, but the program in Figure 4 will produce separate <book>s for each <author> (i.e., it does not group authors). Getting restructuring transformations correct can be challenging for complicated structures. Finally, note that the program in Figure 4 is potentially inefficient as it has seven path expressions, a nested FLWR expression, and a distinct-values function.

This paper takes a different approach to restructuring. The data is restructured by the poly-transform using a *declarative* specification of the structure of the target. (In contrast, transforming using XQuery or XSLT is rather *procedural*.) The specification is called a *structural signature* (see Section 2.1).

```
<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>$9.99</price>
  </book>
</author>
```

Figure 1 Original XML data

```
<publisher>Addison Wesley
  <book>
    <title>The Relational Model for Database Management</title>
    <author><name>E. F. Codd</name></author>
    <price>$46.95</price>
  </book>
</publisher>
<publisher>Academic Press
  <book>
    <title>Cellular Automata</title>
    <author><name>E. F. Codd</name></author>
    <price>$9.99</price>
  </book>
</publisher>
```

Figure 2 Restructured XML data

```
for $b in //book
return
  <publisher> {$b/publisher/text()}
      <book>
        {$b/title}
        <author> {$b/../name} </author>
        {$b/price}
      </book>
  </publisher>
```

Figure 3 Restructuring using XQuery

```
let $x := //book
for $p in distinct-values($x/publisher)
return
 <publisher> {$p}
 { for $b in //book
   where $b/publisher = $p
   return <book>
            {$b/title}
            <author> {$b/../name} </author>
            {$b/price}
          </book>}
 </publisher>
```

Figure 4 Revised XQuery restructuring

The structural signature used to obtain the data in Figure 2 is

publisher#book#(title,author#name,price).

Given the above signature and the data in Figure 1, the poly-transform outputs the data in Figure 2.

It should be emphasized 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). The novelty of this technique lies in its polymorphism – users only need to know what the desired target structure is, but not necessarily the structure of the source or the specific procedures that concretely accomplish the transformation. While a user can employ a polymorphic query construct for simplicity and reusability, she still has the 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 paper is organized as follows. Section 2 defines preliminary concepts. Section 3 presents the poly-transform in detail. We have implemented the poly-transform in Java, and Section 4 describes the result of experiments. Section 5 presents related work and Section 6 concludes the paper with a summary and a discussion on future work.

2. PRELIMINARIES

This section defines *structural signature* and the *Closest relation*, both important concepts to the poly-transform.

2.1 Structural Signature

A structural signature will be used to specify the structure of a forest.

Definition [structural signature] A *structural signature* (*signature* in short), denoted signature, is recursively defined as:

signature := label | label#signature | (signature,...,signature)

where the symbol # is neither part of any label nor a label itself. The following function *sig*() maps a forest to its signature.

- $sig(T)$ = label, where T is a single-node tree and label is the label of the only node;
- $sig(T)$ = label#signature, where T is a tree, the root node is labeled label, and the signature of the forest below the root is signature;
- $sig(F) = (sig(T^1),\ldots, sig(T^k))$, where F is a forest $\{T_1,\ldots, T_n\}$ that consists of n trees and $sig(T^1),\ldots, sig(T^k)$ are the k distinct signatures of the n trees. (That is, $\{T^1,\ldots,T^k\} \subseteq \{T_1,\ldots,T_n\}$ and $\{sig(T^1),\ldots,sig(T^k)\} = \{sig(T_1),\ldots, sig(T_n)\}$.) □

A signature describes the structure of a forest in terms of the hierarchical relationships among its labeled nodes. For example, the signature for the data in Figure 1 is

author#(name,book#(title,publisher,price)).

A signature is composed of *terms*. A label is called a *simple term*, while a list of signatures is called a *complex term*. As an example the signature for the document tree in Figure 1 has two terms. The first is a simple term: author, and the second is a complex term that contains two signatures: name and book#(title,publisher,price). Finally, the first term in a signature is known as a *prefix* while the rest is called the *suffix*.

A signature is a compact representation of the structure of a data instance, similar to a Data Guide. Usually there are only a few different "kinds" of trees in the data forest so a signature is usually much smaller in size than the data instance (as has been noted for Data Guides). A signature is *unlike* a schema specification such as a DTD because a forest has exactly one signature, but could conform to many schemas. Further, a schema can be recursive, but a signature cannot since it describes a data instance.

2.2 Closest Relation

Consider the motivating example in Section 1. We observe that whenever two nodes are *closest* in Figure 1, so are their counterparts in Figure 2. For example, the Cellular Automata <title> is closest to the Academic Press <publisher> in Figure 1; and so are their counterparts in Figure 2. The following definition captures the notion of "closest-ness".

Definition [Closest relation] Suppose $F=(V, E, L)$ is a forest, where V is the node set, E is the edge set, and L is the label function that maps a node to its label. Denote $min(a,b)$ to be the minimum distance between two nodes with labels a and b:

$$min(a,b) = min\{distance(u,v) \mid L(u)=a \wedge L(v)=b,\ u,v \in V\}.$$

Then

$$Closest=\{(u,v) \mid u,v \in V \wedge distance(u,v)=min(L(u),L(v))\}. \quad \square$$

(u,v) belongs to *Closest* if no two other nodes with the same labels are closer than u and v are. Figure 5 diagrams the nodes closest to the first title node in the data model instance for the data in Figure 1. The nodes closest to the title are connected with dashed arrows. In both Figure 1 and Figure 2, for example, the minimum label distance of title and publisher is two. Thus those title and publisher nodes with a distance of two are closest. Note that nodes connected by an edge are always closest. The binary relation *Closest* is reflexive and symmetric.

Intuitively, restructuring should *preserve closeness*. Consider the <name> of a specific <author>. If during a transformation that <name> is arbitrarily switched to a different <author>, then this particular <author> to <name> correspondence is lost and possibly a non-existent correspondence is manufactured. Closeness preservation ensures that when data is restructured elements that are close in the source will remain close in the target. Due to space limitation, we omit the detail on closeness preservation and refer interested readers to our website [10].

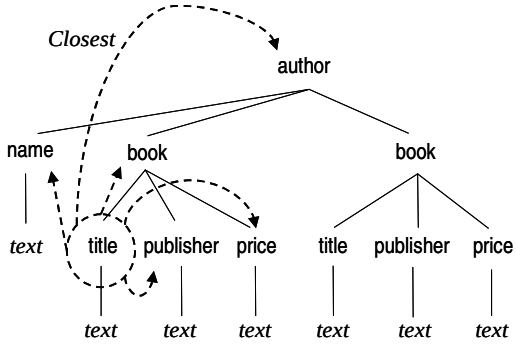

Figure 5 Nodes closest to the first title

3. THE POLY-TRANSFORM

This section discusses the poly-transform in detail. Section 3.1 presents the poly-transform algorithm and Section 3.2 evaluates the implementation complexity. Section 3.3 shows the poly-transform can be easily integrated into XQuery.

3.1 The Poly-transform Algorithm

Figure 6 depicts an algorithm that implements the poly-transform. Given a forest F, it produces a forest F' that conforms to the given signature *sig*. The forest F' is built in a top-down fashion. As we mentioned in the previous section, the poly-transform is devised as a closeness preserving transformation. The algorithm also outputs an *association relation A* that records the correspondence of nodes in the source and target.

```
poly-transform:
Input:  (1) a forest F=(V, E, L)
        (2) a signature sig
        (3) roots = [ r | r∈ V and L(r) = rootLabel ]
Transformation: polyTRANSFORM(roots,suffix(sig))
Output: a forest F'=(V', E', L') and a set of associations A

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

polyTRANSFORM(parents, sig):
  return if sig is empty
  term := prefix(sig)
  if term is a just a label
     for each node p in parents
        p' := clone(p), add (p, p') to A, add p' to V'
        // Find closest nodes of label term
        children := [clone(c) | c∈ V, L(c) = term and Closest(c, p)]
        for each node c' in children
            add (p', c') to E', add (c, c') to A, add c' to V', update L'
     // Do the next label
     polyTRANSFORM(children,suffix(sig))
  else
     // term is a complex term
     for each s in term
        polyTRANSFORM(parents, s)
```

Figure 6. The poly-transform algorithm

As an example, suppose we transform the XML data in Figure 1 using target signature publisher#book#(title,author#name,price). The **polyTRANSFORM** is called with the first label in the signature, publisher, as the root label. A list of two <publisher>s is created. Then the book label is extracted from the signature. For each <publisher> in Figure 1 we find the closest <book>(s). It turns out that each <publisher> will have one <book> child. The third term (title,author#name,price) implies that each <book> needs to have three kinds of children. The transformation is recursively invoked for each kind of child with book as the parent label. Each recursive invocation fleshes out a <book> element with additional children. For example, for the <book> under the Addison Wesley <publisher>, the closest <title> is The Relational Model for Database Management: Version 2 and the closest <price> is $46.95. A recursive call of **polyTRANSFORM**(book, author#name) further establishes that <author><name>E. F. Codd</name></author> will be appended to the aforementioned <book>. The other <book> is similarly transformed. The transformation ends when the last term is processed, yielding the result shown in Figure 2. The association relation *A* is updated each time a new node is created.

Duplicate elimination is an additional step in the algorithm in the creation of the *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.

3.2 Implementation Complexity

The most important and potentially costly step in the **polyTRANSFORM** 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. Each node also keeps the number of its maximum descendent. 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*. 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 7. 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.

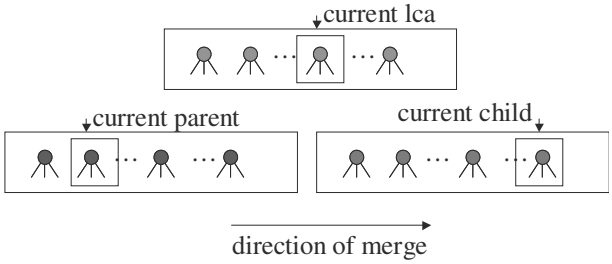

Figure 7. Merging lists of nodes to find closest nodes

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 distance (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.

3.3 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 the data in Figure 2 from the data in Figure 1. (The data in Figure 1 and Figure 2 is each a fragment of an XML document; for the source data to be well-formed XML, we now assume all book nodes are enclosed by a bibliography node).

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

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 sig, the poly-transform enabled XQuery is

```
for $root in //prefix
return polyTRANSFORM($root, sig)
```

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

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

Since transforming the source is potentially expensive, a lightweight alternative is to add a new axis to XPath: the closest axis [9]. 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 books.xml or publishers.xml.

```
for $a in //title[#author="Codd"]
return $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.

4. EXPERIMENTS

This section describes 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 [10].

We first evaluate the basic cost of the poly-transform. We test 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

1. a *deep* signature: last#publisher#title#first#author,
2. a *flat* signature: publisher#(last,title,first,author), and
3. 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.

In Figure 8, 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 9 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.

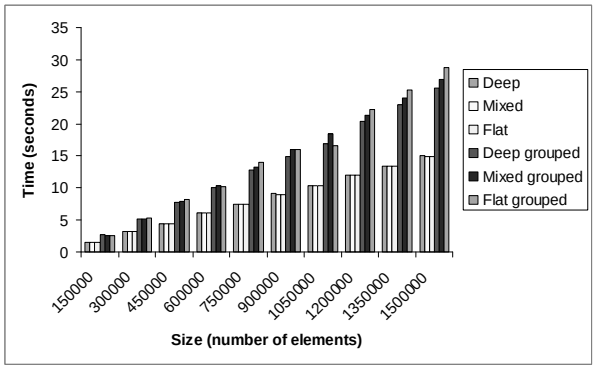

Figure 8. Experiment one measures three signatures

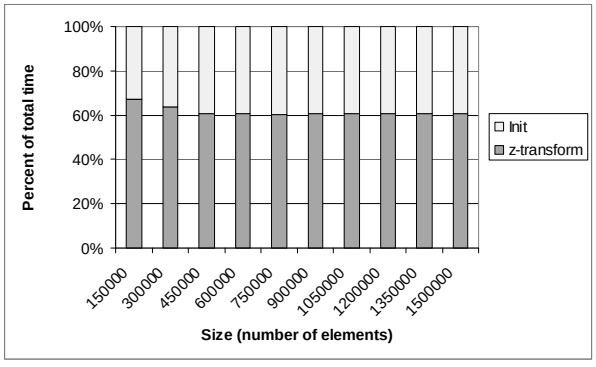

Figure 9. Relative cost of initialization vs. poly-transform

5. RELATED WORK

Correctly programming a restructuring transformation has previously been recognized as a non-trivial task. Pankowski [7] and Krishanmurthi et al. [5] 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 paper. Unlike the poly-transform these languages are not polymorphic, 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. Erwig discusses several basic transformation operations such as renaming, deletion and grouping [3]. It introduces the notion of *information content* and *information preservation*. We formalize the notion of information preservation by defining reversible transformations.

Restructuring is not limited to XML or hierarchical data. In a broader sense, restructuring can be viewed as a problem related to schema matching and data integration. YAT [2] and SilkRoute [4] translate relational data into XML. Abiteboul et al. [1] and BcBrien and Poulovassilis [6] 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 [6]. 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.

An important component of the polymorphic restructuring is the *Closest* relation. This is an application of the *closest axis* [9], which locates *closest* nodes in an instance of an XML data model. This axis is polymorphic in nature as well, because its valid use is not tied to certain specific hierarchical structure.

6. CONCLUSION

This paper 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, it employs the *Closest* relation and the notion of closeness preservation as semantic constraint in developing restructurings. Third, the paper presents the poly-transform algorithm. 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. Fourth, we described a new kind of join that can efficiently implement the poly-transform. Finally, the paper describes 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).

7. REFERENCES

[1] S. Abiteboul, S. Cluet, and T. Milo. Correspondence and Translation for Heterogeneous Data. In *Proc. ICDT*, 1997, pp. 351-363.

[2] V. Christophides, S. Cluet, and J. Simèon. On wrapping query languages and efficient XML. *SIGMOD Record*, Volume 29, Issue 2, June 2000, pp. 141 - 152.

[3] M. Erwig. Toward the Automatic Derivation of XML Transformations. In *Proceedings of the 1st Int. Workshop on XML Schema and Data Management* (*XSDM'03*), Springer-Verlag, Lecture Notes in Computer Science 2814, 2003, pp. 342 - 354.

[4] M. Fernandez, W. Tan, and D. Suciu. Silkroute: Trading Between Relations and XML. In *Proceedings of the Ninth International World Wide Web Conference*, 2000.

[5] S. Krishanmurthi, K. Gray and P. Graunke. Transformation-by-example for XML. *The 2nd International Workshop of Practical Aspects of Declarative Languages*, Springer-Verlag, Lecture Notes in Computer Science 1753, 2000.

[6] P. McBrien and A. Poulovassilis. A Semantic Approach to Integrating XML and Structured Data Sources. *Conference on Advanced Information Systems Engineering*, 2000.

[7] T. Pankowski. A High-Level Language for Specifying XML Data Transformations. In *ADBIS 2004*, Springer-Verlag, Lecture Notes in Computer Science 3255, 2004.

[8] World Wide Web Consortium. XML Query Use Cases. http://www.w3.org/TR/xquery-use-cases/.

[9] Shuohao Zhang and Curtis Dyreson. Symmetrically Exploiting XML Data. *The 15th International World Wide Web Conference*, Edinburgh, Scotland, May 2006. (to appear)

[10] http://www.eecs.wsu.edu/~cdyreson/pub/polytransform