Querying XML Data: As You Shape It

Curtis E. Dyreson [#1], Sourav S. Bhowmick [*2]

[#]*Department of Computer Science, Utah State University, Utah, USA*
[1]Curtis.Dyreson@usu.edu

[*]*School of Computer Engineering, Nanyang Technological University, Singapore*
[2]assourav@ntu.edu.sg

Abstract— **A limitation of XQuery is that a programmer has to be familiar with the shape of the data to query it effectively. And if that shape changes, or if the shape is other than what the programmer expects, the query may fail. One way to avoid this limitation is to transform the data into a desired shape. A data transformation is a rearrangement of data into a new shape. In this paper, we present the semantics and implementation of XMORPH 2.0, a *shape-polymorphic data transformation* language for XML. An XMORPH program can act as a *query guard*. The *guard* both transforms data to the shape needed by the query and determines whether and how the transformation potentially loses information; a transformation that loses information may lead to a query yielding an inaccurate result. This paper describes how to use XMORPH as a query guard, gives a formal semantics for shape-to-shape transformations, documents how XMORPH determines how a transformation potentially loses information, and describes the XMORPH implementation.**

I. INTRODUCTION

In recent times, the database community has made considerable progress in devising efficient strategies to query large XML databases. XML data has a tree-like data model. Data in the tree is arranged in a particular shape as described by, for instance, a *data guide* [16]. About 40 years ago, E. F. Codd observed that this kind of data model has a problem. Queries in tree-like data models utilize path expressions that are *necessarily* tightly coupled to the shape of the data [7]. For example, assume that we want to extract the book titles written by an author using the following XQuery query.

```
<data> {
   for $a in doc("x.xml")//author,
       $n in $a/name
   let $t := $a/book/title
   return <name>{$n/text()} {$t}</name>
} </data>
```

The query is to be independently applied to each of the three XML data instances depicted in Figure 1. Intuitively, each of the instances has the *same* data about books, authors, and publishers. But the shape of each instance is *different*. The query will only succeed for instance **(c)**, it will fail to produce any XML for instances **(a)** and **(b)** because the shape of the data is different from what the query needs.

Codd astutely observed that there are myriad natural shapes to any tree-like data collection and that tightly coupling path expressions to just one shape (or even a subset of the potential shapes) prevents queries from being ported to new collections that have similar data but differ in shape, and also increases the cognitive burden on query writers since they have to know the data's shape to write queries.

Inspired by Codd's observation that the same data may be shaped very differently, in this paper we propose a shape-polymorphic data transformation language called XMORPH 2.0 for XML that enables a user to query XML data as she likes it rather than how it is initially *shaped*. Specifically, we investigate the issue of whether data with the *wrong* shape can be transformed to a shape needed by a query. This extends how we currently query data by making it possible to query more data collections with the same query. We propose that each query have two components: 1) a *query guard*, which is a *lightweight, reusable specification* of the shape needed by the query, and 2) an XQuery query.

The query guard protects the query by testing whether the data can be transformed (without losing information) to the shape specified in the guard, and transforms the data as needed. The guard is not fixed to a single query, rather we assume that the same guard will be reused for many queries. For the example query, the needed shape is one in which `<book>` and `<name>` are children of an `<author>`, and `<title>` is a child of `<book>`. A guard is expressed in XMORPH as follows. (The syntax for XMORPH is reviewed in Section III.)

```
MORPH author [ name book [ title ] ]
```

The guard specifies that each `<author>` has `<name>` and `<book>` children, and that `<title>` is a child of `<book>`. The keyword `MORPH` indicates that the desired shape is only composed of the specified types.

In query evaluation, the guard is evaluated first, then the query. The evaluation of the guard has two purposes. First, it checks whether the data is in the needed shape. If so, query evaluation can proceed. If not, the second aspect of the guard comes to life. The guard transforms the data to the needed shape as described in more detail later.

Figure 2 shows the result of evaluating the example guard on each instance in Figure 1. Data instances **(a)** and **(b)** in Figure 1 are (logically) transformed to the same instance, while instance **(c)** differs, but only in the grouping of authors by name (the grouping is in the source data).

Observe that using a query guard reduces the cognitive burden of writing XQuery queries. Instead of learning the shape of the data and crafting a query for it, a query writer can simply declare a desired shape with a guard and write

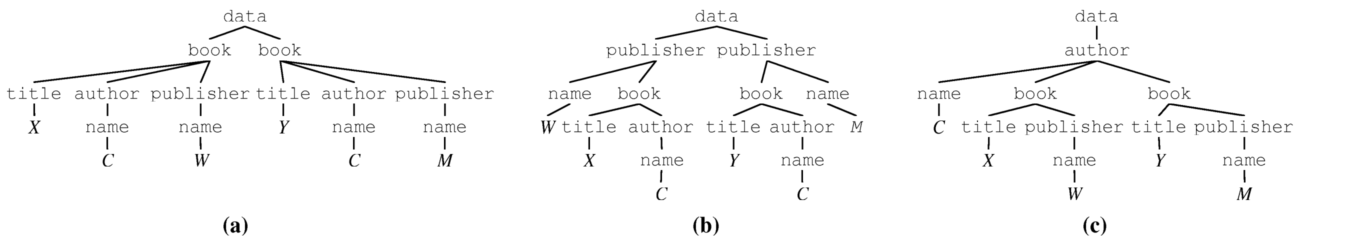


Fig. 1. The same data with three different shapes.

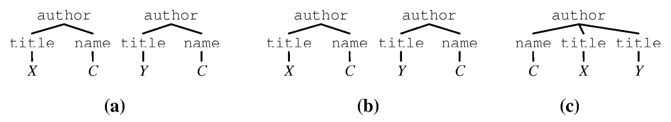


Fig. 2. Transformed instances.

a query expecting to use that shape. They can then take that query and guard and apply it to *any* XML data collection. The guard will determine if the query can be correctly evaluated.

It is worth mentioning that query guards have compelling practical significance as XML document structures evolve over time in the real world. Specifically, database administrators may revise the design over time to address issues such as redundancy, space overhead, performance, and usability [10], [26]. For instance, path `author/name` is repeated under every subtree of element `book` in the database fragment in Figure 1(a). The database administrator may normalize the schema of the document to remove redundancy (Figure 1(c) shows an instance of the normalized schema) [2].

A key challenge in designing a query guard is that some transformations *potentially lose information*. Consequently, it is important for the guard to identify and report *lossy* transformations. Notice that it is not readily apparent in the aforementioned example whether the guard is *good* in the sense that it protects the query by neither manufacturing nor discarding data. This issue is vital to a user. If the transformation specified by a guard is *lossy* then the subsequent query evaluation will be similarly lossy and inaccurate.

Let's introduce terminology to more precisely describe what we mean by a good guard. This terminology is adapted from the vocabulary of type systems in programming languages since a guard plays a role similar to a *data type* in a programming language, i.e., it defines how the data is structured or encoded. A guard is *narrowing* if it ensures that data is not created, *widening* if it ensures that no data is lost, *strongly-typed* if it both narrowing and widening, *weakly-typed* if it neither narrowing nor widening, or has a *type mismatch* if the guard mentions a type that is absent from the source.

The guard given above turns out to be strongly-typed (see Section V for more details), but consider the following, slightly different guard.

```
MORPH author [ title name publisher [name] ]
```

This guard transforms each instance to that shown in Figure 3. The transformation for instance **(c)** is *widening*. Observe that in the result both titles, X and Y, are closest to the first publisher, W, which adds data, i.e., closest relationships, not present in the source.

XMorph provides detailed feedback about potential information loss in a transformation. XMORPH identifies and reports precisely which part of a guard is lossy. An XQuery programmer can use this feedback to add syntax to a query guard to indicate that the loss is acceptable, e.g., most narrowing transformations will be fine, just as a C++ programmer might add a `cast()` to transform the result of an expression to a suitable type when permissible.

This paper builds on previous XMORPH research [11], [13]. In [11] we presented the advantages of a shape-polymorphic (shape adapting) data transformation language, gave the syntax of XMORPH 1.0 (which extend to XMORPH 2.0 in this paper), illustrated several program examples, and informally sketched the notion of *closeness*, which we developed elsewhere [12], [28]. In [13], we described the user interface for the query tool. In contrast, this paper makes the following new contributions.

- We develop the notion of a query guard. *To the best of our knowledge, this is the first work that introduces query guards for database query languages.*
- We develop a formal data model and give a formal semantics for XMORPH. *This semantics is the first such description for a shape-to-shape transformation language.*
- We show exactly how XMORPH determines potential information loss in a data transformation, which is a key part of how XMORPH works.
- We present an algorithm for rendering a transformed shape. Though the "read" cost of the transformation is linear in the size of output regardless of the size or complexity of a query guard; the "write" cost of the algorithm is quadratic since the transformation may duplicate snippets of source data.
- We describe the implementation of XMORPH and present the results of several experiments that empirically measure the cost of a transformation.

This paper is organized as follows. The next section reviews related work. Section IV develops a novel data model for query guards. Section V describes how a query guard transforms the data by preserving closeness relationships and protects a query by detecting whether the transformation is potentially lossy. A denotational semantics for query guard constructs is given in Section VI, followed by an algorithm to render a guard to XML. Section VIII presents the architecture for an XMORPH implementation. The cost of transforming data is quantified in Section IX, and the paper concludes in Section X.

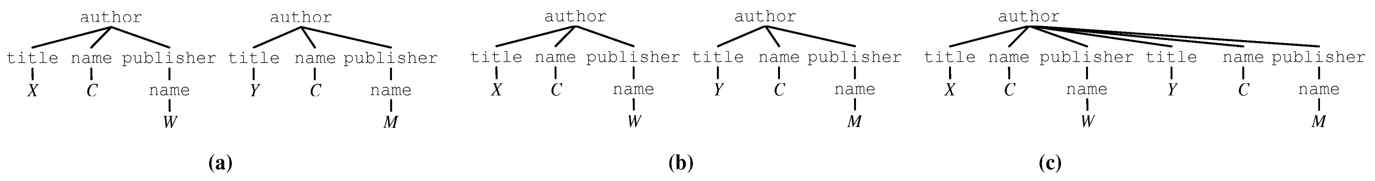


Fig. 3. A lossy transformation.

II. Related Work

One way to loosen the tight coupling of path expressions to the shape of data is to relax the path expressions or approximately match them to the data by exploring a space of shapes that are within a given edit distance [1], [3], [18]. Though such techniques work well for small variations in shape or values, there is a *very large* edit distance among every pair of instances in Figure 1, which we would like to consider as the same data. Relaxing a query to explore all shapes within a large edit distance is overly permissive, and includes many shapes which do not have the same data. Query correction [8] and refinement [4] approaches are also best at exploring only small changes to the shape. Another approach would be to use XML *search* to find the data, regardless of the shape [9], [21], [25]. Search engines are also overly permissive in finding data of all different shapes. Moreover, once found, search engines do not transform data to a shape needed by a query, or report on potential information loss in a transformation. The data could be transformed with a program in an XML transformation language [19], [22]. However, each transformation depends on the shape of the input and would have to be re-programmed for a different shape (i.e., two separate transformation programs would be needed for the example, one from instance **(a)** to **(c)** and one from **(b)** to **(c)**). It would be more desirable if a programmer could simply declare the desired shape in a single guard. Another alternative is to specify query constructs, usually involving the least common ancestor of pairs of nodes, to query data independent of its shape (c.f., [20], [27]). But none of these approaches identifies the potential information loss in explicitly transforming and mutating the shape of data. Moreover, query guards also need to transform XML values since the values play an important role in for instance the distinct-values function and the return clause of an XQuery query; it is the values in the target shape rather than the source shape on which the query should be evaluated.

Data integration is another area of related research [5]. Data is integrated from one or more source schemas to a target schema by specifying a mapping to carry out a specific, fixed transformation of the data. Once the data is in the target schema, there is still the problem of queries that need data in a shape other than the target schema. In some sense schema mediators integrate data to a fixed schema, which is the starting point for what query guards are designed for. The different problem leads to a difference in techniques used to map or transform the data. For instance tuple-generating dependencies (TGDs) are a popular technique [14], [17]. Part of a TGD is a specification of the source shape from which to extract the data. Specifying the source shape will not work for a query guard, a query guard must be agnostic about the source. A second concern for query guards is that the transformation must be fully automatic. A third difference is the need to determine potential information loss, which is an important part of a query guard, but absent from such mappings for data integration. For schema mediation if a programmer programs a data transformation that loses information, that information is gone and subsequent queries on the transformed data will never know about the information loss.

Recently, research has explored preserving information in data integration, namely by describing schema embeddings that ensure *invertibile* mappings that are *query preserving* [15]. Such embedding and mappings are too restrictive for query guards since any mapping that permutes ancestor/descendent relationships—which are useful in query guards—are potentially not invertible (since duplicates are potentially created for each of many descendents in the mapping).

III. XMorh 2.0 Syntax

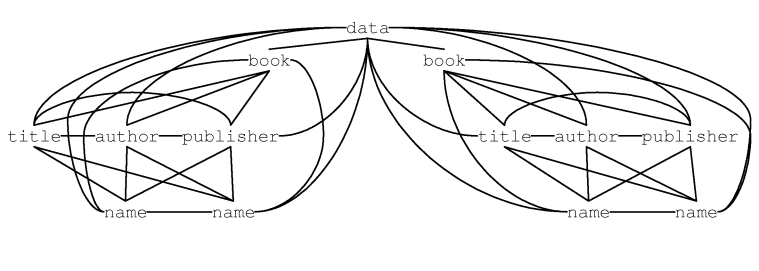
A query guard specifies the shape of the output. A programmer can use the following constructs to specify a guard (we capitalize the keywords for emphasis, guards are case- and whitespace-insensitive). While these constructs do not increase the theoretical power of query guards, since any shape can be specified by simply nesting labels (i.e., by giving a data guide), they help to make guards easier to program and more concise.

- `MORPH` *shape* - The *shape* is the desired shape, it uses only the types specified in the *shape*. The shape of a `MORPH` can contain the following.
  - `CHILDREN` *label* (alternatively *label* `[*]`) - Include the type matching the *label*, together with its children from the source shape.
  - `DESCENDANTS` *label* (alternatively *label* `[**]`) - Include the type matching the label and all descendants from the source shape.

  As an example consider the following query guard.

  ```
  MORPH data [author
               [* book [** publisher [*]]]]
  ```

  It specifies that the desired shape has `data` as the root. At the next level down is `author` (its children are included with `*`). Below `author` is `book` (and its descendants, included with `**`). Finally, below `book` is `publisher` (and its children). This guard specifies a range of shapes that includes the shape of data instance **(c)** in Figure 1.
- `MUTATE` *shape* - Modify the entire shape of the input, rearranging those parts specified in the *shape*. The shape of a `MUTATE` can contain the following.
  - `DROP` *shape* - Remove the types from in *shape*.



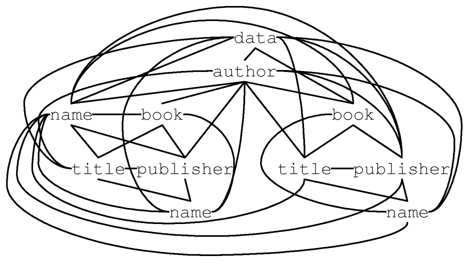


Fig. 4. Closest graphs.

– CLONE *shape* - Duplicate the types in *shape*.

As an example consider a guard to MUTATE Figure 1**(b)** to **(a)**.

```
MUTATE book [ publisher [ name ] ]
```

The mutation moves publisher below book leaving the rest of the shape unchanged.

- TRANSLATE *label* → *label* - Rename the types matching the *label* to the *label*.

Additional constructs can apply within any shape.

- RESTRICT *shape* - The type of the root is restricted by the shape, but only the root type in the shape appears in the output.
- NEW *label* - Introduce a new label into the shape.

Query guards can also be composed.

- COMPOSE *guard*$_1$ *guard*$_2$ (alternatively *guard*$_1$ | *guard*$_2$) - Pipe the output of the first guard into the second guard. Below is an example of composing a MORPH and a MUTATE using the abbreviation for composition, |.

```
MORPH author [name] | MUTATE (DROP name)
```

The final shape consists only of author (closest to a name).

Finally, the behavior of the type-checking can be controlled. By default only strongly-typed guards are allowed.

- CAST-NARROWING *guard* - Allow narrowing guards.
- CAST-WIDENING *guard* - Allow widening guards.
- CAST *guard* - Allow weakly-typed guards.
- TYPE-FILL *guard* - Generate new labels for missing types.

The following guard fills in missing types and checks to ensure that the guard is not narrowing.

```
CAST-WIDENING (TYPE-FILL
       MUTATE author [ title ])
```

As we noted above XMORPH can express any shape described by a data guide, but XMORPH cannot express an ordering among siblings as part of a transformation, i.e., the shape is unordered. Nor value-based transformations be expressed, for example to transform the <author> "Codd" differently than other <author>s. We plan to explore such transformations in future. It should also be noted than since XQuery is Turing-complete, clearly query guards do not increase the expressiveness of XQuery.

IV. DATA MODEL

The data model for a query guard has two components: a *closest graph* and a *shape*. The closest graph captures relationships among the XML data, while the shape describes the structure of the types in the data.

Definition 1: (**Closest Graph**) The closest graph, $G = (V, E_C)$, for an XML data collection, D, is a graph, where

- V is a set of vertices, one vertex for each element or attribute in D, and
- $E_C = \{(v, w) \mid v, w \in V \wedge (v, w) \in \textbf{closest}(D)\}$ is a set of (undirected) *closest edges* (the **closest** relation is defined below). ∎

Figure 4 depicts the closest graph for the data instances of Figure 1. In the figure, a solid line represents a closest edge. A closest edge represents a closest relationship (which is described in detail below) between a pair of vertices. Text content is not displayed.

We assume that the following auxiliary functions, which are used later in the paper, are available for vertices, $v \in V$.

- **value**(v)- The text content of an element or attribute v.
- **name**(v) - The name of the attribute or element.
- **typeOf**(v) - The *type* of an element or attribute. Each vertex has a well-defined type. We are agnostic about how a vertex is typed, e.g., the type could be derived from a schema. By default we assume that the type is specified as a concatenation of the names of the elements on the path from the data root to the vertex, v_k, e.g., **name**(v_0).**name**(v_1). ... .**name**(v_k) where v_i is the vertex at level i.

Closeness depends on two kinds of *distance* as defined below. The first kind of distance is the familiar graph distance: for all v, w in an XML data model instance, D, the *distance* from v to w, denoted **distance**(D, v, w), is the number of edges on the path from v to w (or in the case of an XML graph, the shortest path). The **closest** relation also needs the distance between types: The type distance, **typeDistance**, is the minimal distance between every pair of vertices with the given types.

$$\textbf{typeDistance}(D, t_1, t_2) = \textbf{min}(\{\textbf{distance}(D, v, w) \mid v, w \in D \ \wedge \ \textbf{typeOf}(v) = t_1 \ \wedge \ \textbf{typeOf}(w) = t_2\})$$

The closest relation relates vertices that have a distance equal to the type distance.

Definition 2: (**Closest Relation**) For an XML instance, D,

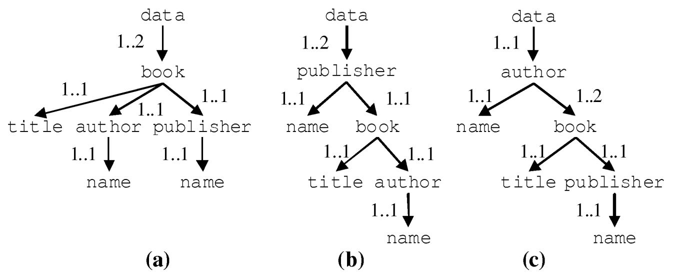


Fig. 5. Adorned shapes for the data of Figure 1.

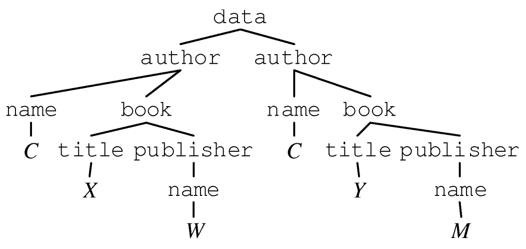


Fig. 6. Data of Figure 4(a) xformed using the shape of Figure 5(c).

$$\textbf{closest}(D) = \{(v, w) \mid v, w \in D \wedge \textbf{typeDistance}(\textbf{typeOf}(v), \textbf{typeOf}(w)) = \textbf{distance}(D, v, w)\}. \quad \blacksquare$$

The other part of the data model is a *shape*. The shape describes the parent/child relationships among the *types* in a data collection, i.e., it is a *DataGuide* [16]. We *adorn* the shape with a cardinality (the adornment is used in Section V-A).

Definition 3: (**Adorned Shape**) An adorned shape is a forest, S, where

- $S = \{(t, u, p) \mid t, u \in T \ \wedge \ p \in \{n..m \mid n, m \in \mathbb{N}\}\}$ is a set of labeled, directed *type edges*; an edge from t to u represents that a node of type t is a parent of a node of type u in the data, and p is an edge label indicating a cardinality range such that n is the minimum (m is the maximum) number of children of type u for any parent of type t, and
- T is a set of data types and $\circ$ (a symbol used to indicate the forest leaf boundary).

Every leaf, t, will be represented by a leaf edge $(t, \circ, 0..0))$ in S. S obeys the normal conditions of a forest, e.g., every type has at most one parent. $\blacksquare$

We assume that the following helper functions are defined for a shape, S.

- **types**$(S) = \{t \mid (t, \_) \in S\}$ is the set of types in S.
- **roots**$(S) = \{t \mid (t, \_) \in S \wedge \neg\exists v((t, v) \in S)\}$ is the set of types in S that have no incoming edges.
- **rootEdges**$(S) = \{(t, \_) \mid (t, \_) \in S \ \wedge \ t \in roots(S)\}$ is the set of edges emanating from the roots.

Figure 5 displays the adorned shapes for the data instances of Figure 1. As the data instances are small, the shapes are essentially the same size, but in general a shape for a data collection will be much smaller than the data. Almost every edge has a cardinality of 1..1, though some are 1..2 indicating that each parent node has at least one child and at most two children of the given type. To further exemplify the cardinality, assume that in data instance **(a)** of Figure 1, the leftmost `author` does not have a `name`. In that case the edge from `author` to `name` would be labeled 0..1 to indicate that some `authors` do not have a `name`.

V. Data Transformations

Data is transformed by a query guard to a desired shape. The transformation is performed by a closeness-preserving transform function, **xform**(G, R), which constructs an XML data instance in the shape of R with data chosen from G.

Definition 4: (**Closeness-preserving Transform**) The closeness-preserving transformation function,

$$\textbf{xform}(G, R) = (W, E),$$

takes as input a closest graph, $G = (V, E_C)$, with shape S, and transforms it to an instance, (W, E), with shape R, where

- $W \subseteq V$ is the subset of vertices in G,
- $E = \{(v, w) \mid v, w \in W \wedge (v, w) \in E_C \wedge (t, u) \in R \wedge$ **typeOf**$(v) = t \wedge$ **typeOf**$(w) = u\}$ is a set of parent/child edges. $\blacksquare$

As an example consider the rearrangement of the data of Figure 4(a) using the shape of Figure 5(c). The **xform** produces the XML data instance shown in Figure 6.

A. Relating Closest Graphs

Before discussing potential information loss, we give a definition that helps to relate two closest graphs.

Definition 5: (**Closest Graph Subset**)Closest graph $H = (W, E_H)$ is said to be a subset of closest graph $G = (V, E_G)$, which we will denote as $H \subseteq_C G$, if and only if $E_H \subseteq E_G \wedge W \subseteq V$. $\blacksquare$

Each transformation has some information loss property. Let $G = \textbf{closest}(D)$ and $H = \textbf{closest}(\textbf{xform}(G, R))$, then **xform**$(G, R)$ is said to be

- *non-additive* iff $H \subseteq_C G$, and *additive* otherwise,
- *inclusive* iff $G \subseteq_C H$, and *non-inclusive* otherwise, or
- *reversible* iff $H \subseteq_C G$ and $G \subseteq_C H$, otherwise *irreversible*.

Ideally, a transformation will be *reversible* and neither lose nor create a closest edge. At the least a good transformation should be *non-additive*, otherwise some edges may be introduced. As an example, consider the transformations of Figures 2 and 3. Do any of these transformations lose information, or are they all reversible? We answer this question below.

B. Potential Information Loss

Although we could transform the data and compare the resulting closest graphs to determine reversibility, it would be faster, especially for large data collections, to determine, prior to transforming data, whether a transformation is reversible by reasoning about the shape of the data and the shape of the result. Determining reversibility depends on the *cardinality* of the relationships between types in a shape.

TABLE I
CARDINALITY FOR PAIRS OF TYPES IN FIGURE 5(C)

	data	author	author name	book	title	pub	pub name
data	1..1	1..1	1..1	1..2	1..2	1..2	1..2
author	1..1	1..1	1..1	1..2	1..2	1..2	1..2
name	1..1	1..1	1..1	1..2	1..2	1..2	1..2
book	1..1	1..1	1..1	1..1	1..1	1..1	1..1
title	1..1	1..1	1..1	1..1	1..1	1..1	1..1
pub	1..1	1..1	1..1	1..1	1..1	1..1	1..1
name	1..1	1..1	1..1	1..1	1..1	1..1	1..1

An adorned shape has the bare minimum of cardinality information. Since any types can be related, we are interested in computing the cardinality along the path connecting two types.

Definition 6: (**Path cardinality**) Let S be an adorned shape. For any pair of types, $t, s \in S$, let the path from the least common ancestor, v, of t and s to s be

$$(v, x_1, n_1..m_1), \ldots, (s, x_k, n_k..m_k).$$

Then the path cardinality from t to s,

$$\textbf{pathCard}(S, t, s) = 1 * n_1 * \ldots * n_k \;..\; 1 * m_1 * \ldots * m_k.$$

The cardinality from t to v (up the shape) is always 1..1. ∎

Table I shows the path cardinality for every pair of types in adorned shape **(c)** of Figure 5. We will use the path cardinality to determine potential information loss by predicting the cardinality on each edge in the transformed shape.

Definition 7: (**Predicted adorned shape**) Let S be an adorned shape for $G = \textbf{closest}(D)$. Then the predicted adorned shape, R_p, for $\textbf{xform}(G, R)$ is

$$R_p = \{(t, s, n..m) \mid (t, s) \in R \wedge n..m = \textbf{pathCard}(S, t, s)\}.$$

∎

In reasoning about information loss we are only interested in *type-complete* transformations, that is, transformations that transform all of the types in the source, since it is trivial to choose any subset of a closest graph, G, as the source.

Definition 8: (**Type-complete xform**) Let XML data instance, D, have shape S and $\textbf{closest}(D) = G$). Then a transformation, $\textbf{xform}(G, R)$, is *type-complete* iff there is a 1-to-1 correspondence between $\textbf{types}(S)$ and $\textbf{types}(R)$. ∎

The definition captures the idea that the desired shape, R, is a rearrangement of *all* and *only* the types in S.

An **xform** is *inclusive* if it can be ensured that no closest relationships are lost. Since an **xform** is a closeness-preserving transformation, it only loses relationships if there are one or more vertices that are discarded in the transformation.

Theorem 1: Let S be the adorned shape of data instance D, $G = \mathbf{closest}(D)$, $\mathbf{xform}(G, R)$ is type-complete, and R_p be the predicted adorned shape for R. Then $\mathbf{xform}(G, R)$ is inclusive if for every pair of types $t, s \in S$ the minimum path cardinality does not increase from zero to non-zero in R_p.

Proof: We need to show that $G \subseteq_C \textbf{closest}(\textbf{xform}(G, R))$ under the given conditions. Let $\textbf{closest}(\textbf{xform}(G, R)) = (W, F_c)$ and $G = (V, E_c)$. Assume that $V \subseteq W$, i.e., assume that $\textbf{xform}(G, R)$ does not lose any vertices. Then it must be the case that $E_c \subseteq F_c$ since $\textbf{xform}(G, R)$ is closeness-preserving, i.e.,

$$\forall v, w \in W[(v, w) \in E_c \Rightarrow (v, w) \in F_c].$$

If not, the **xform** did not preserve a closest relationship between v and w. So to ensure inclusiveness, we must ensure that $V \subseteq W$. The only way for a vertex, w, to be discarded from the transformation is if w is not close to any vertex of type t in G, but R requires that every node of type $\textbf{typeOf}(w)$ be a descendant of some vertex of type t. But w cannot have such an ancestor with the given cardinality condition, hence the condition ensures that the transformation is inclusive. ∎

As an example, any type-complete transformation of data instance **(c)** of Figure 5 is inclusive because no path has a minimum path cardinality of zero. However, suppose that the `name` of an `author` is optional, i.e., the cardinality of that edge in Figure 5**(c)** is 0..1. Then the following query guard is (potentially) non-inclusive (a `MUTATE` transforms the entire shape as described in Section VI and the Appendix).

```
MUTATE name [ author ]
```

The query guard is non-inclusive because the minimum path cardinality from `author` to `name` increases from 0 to 1. Each `name` is required to have a closest `author` in the transformed data, but not in the source. Said differently, any `author` that does not originally have a `name` will be omitted from the result. The following transformation will however be inclusive.

```
MUTATE data [ name author ]
```

No path with a min. cardinality of zero becomes non-zero.

Some transformations can also be shown to be non-additive.

Theorem 2: Let S be the adorned shape of data instance D, $G = \mathbf{closest}(D)$, $\mathbf{xform}(G, R)$ be type-complete, and R_p be the predicted adorned shape for R. Then $\mathbf{xform}(G, R)$ is non-additive if for every pair of types $t, s \in S$ the maximum path cardinality does not increase in R_p.

Proof: We need to show that $\textbf{closest}(\textbf{xform}(G, R)) \subseteq_C G$ under the given conditions. Let $\textbf{closest}(\textbf{xform}(G, R)) = (W, F_c)$ and $G = (V, E_c)$. We know that $W \subseteq V$, so we only need to show that $F_c \subseteq E_c$, that is, that no new closest edges are created. Since $\textbf{xform}(G, R)$ is closeness-preserving, i.e.,

$$\forall v, w \in W[(v, w) \in E_c \Rightarrow (v, w) \in F_c]$$

all the closest edges for which there are vertices in W that are in E_c belong to F_c. Since the condition states that there can be no in the maximum cardinality, no more edges can be added to F_c and the transformation is ensured to be non-additive. ∎

As an example, the following type-complete transformation of data instance **(c)** of Figure 5 is non-additive.

```
MUTATE name [ author ]
```

Since `name` to `author` is 1..1, swapping their position does not change the predicted maximum path cardinality of any pair of types in the shape. However, reconsider the example from Section I:

```
MORPH author [ title name publisher [name] ]
```

applied to instance **(c)** in Figure 1. Observe the corresponding adorned shape for **(c)** in Figure 5. It is easy to see why the transformation is additive. The predicted maximum cardinality for `title` to `publisher` increases from 1 to 2, which implies that each `title` in the source data has a single closest `publisher` but in the result has (potentially) two closest `publishers`. The transformation might add closest relationships that are not present in the source.

Both inclusiveness and non-additiveness can be computed during translation of a query guard at little additional cost (assuming the size of the data to transform is much larger than the number of distinct types).

VI. XMORPH 2.0 TRANSFORMATION SEMANTICS

The single most important thing to understand about a query guard is that it specifies a shape. So each component of the guard is a function that maps a shape to a shape; the shape can subsequently be rendered as XML but a query guard is only a specification of a desired shape. In this section we present a denotational semantics for the evaluation of query guard, P, on a data instance, (G, S). Let

$$\Psi : \textit{Guard} \rightarrow ((\textit{Graph} \times \textit{Shape}) \rightarrow (\textit{Graph} \times \textit{Shape}))$$

be a semantic function that takes a guard and produces a function that maps one data instance to another, and let

$$\xi : \textit{Guard} \rightarrow (\textit{Shape} \rightarrow \textit{Shape})$$

be a semantic function that takes a guard and produces a function that maps a shape to a shape. Then the meaning of a guard is

$$\Psi[\![P]\!](G, S) = \textbf{render}(G, \xi[\![P]\!](S)).$$

The semantics states that a guard constructs a shape. The shape is subsequently used to render the data extracted from the closest graph (e.g., as XML).

A denotational semantics for the shape transformation, ξ, is given below.

- $\xi[\![\texttt{MORPH } P]\!](S) = \xi[\![P]\!](S)$. A `MORPH` constructs the shape corresponding to the pattern specified by its child, P.
- $\xi[\![p_0 \texttt{ [} p_1\, p_2 \, \ldots \, p_n \texttt{]}]\!](S) = \textbf{extend}(\xi[\![p_0]\!](S), R)$ where

  $$R = \xi[\![p_1]\!](S) \ \cup \ \xi[\![p_2]\!](S) \ \cup \ldots \cup \ \xi[\![p_n]\!](S)$$

  and

  $$\textbf{extend}(X, R) = X \ \cup \ R \ \cup \ T.$$

  T is the set of *closest type pairs*, which is computed as follows. Let

  $$A = \{(x, r) \mid x \in \textbf{roots}(X) \ \wedge \ r \in \textbf{roots}(R)\}$$

  and

  $$m = \textbf{min}(\textbf{typeDistance}(x, r) \mid (x, r) \in A),$$

  then

  $$T = \{(x, r) \mid (x, r) \in A \wedge \textbf{typeDistance}(x, r) = m\}.$$

  This construct builds a shape by connecting parent to child types in the shape, but only those parents and children that are closest.
- $\xi[\![\textit{label}]\!](S) = L \times \{\circ\}$. The set of types corresponding to a *label* is

  $$L = \{t \mid t \in \textbf{types}(S) \wedge \textit{label} \in \textbf{name}(t)\}.$$

  A new shape is created which is a set of leaves. There are three possible outcomes to using a label to select a set of types.
  1) The label *does not match* any type in S, in which case a semantic type error is generated.
  2) The label *matches a single* type, in which case it is said to be unambiguous.
  3) The label *matches more than one type*, e.g., there may be several types for a label such as `author`. When a label is ambiguous, a user can disambiguate it by more fully specifying the desired type, e.g., `book.author` vs. `journal.author`.
- $\xi[\![\texttt{CHILDREN } P]\!](S) = \xi[\![P]\!](S) \ \cup \ C$, where C is a set of edges to children,

  $$C = \{(t, s) \mid t \in \textbf{roots}(\xi[\![P]\!](S)) \wedge (t, s) \in S\}.$$

  This adds to the shape all of the children of the roots of P. Below is an example `MORPH` that includes all of the children of an `author` and adds the closest `title`(s) as children. We also give the example using the abbreviated syntax for children, $\star$.

  ```
  MORPH (CHILDREN author) [ title ]
  MORPH author [ * title ]
  ```
- $\xi[\![\texttt{DESCENDANTS } P]\!](S) = \xi[\![P]\!](S) \ \cup \ \{(v, w) \mid v, w \in T\}$, where T is a subtree of S rooted at $t \in \textbf{roots}(\xi[\![P]\!](S))$. This adds to the shape all of edges in the subtrees in S rooted at a root in the meaning of P. Below is an example `MORPH` that includes all of the descendants of a `book` and adds the closest `name`(s) as children. We also give the example using the abbreviated syntax for descendants, $\star\star$.

  ```
  MORPH (DESCENDANTS book) [ name ]
  MORPH book [ ** name ]
  ```
- $\xi[\![\texttt{MUTATE } P]\!](S) = \textbf{mutate}(\xi[\![P]\!](S)) = F$. The **mutate** operator merges the input shape with the shape corresponding to its child, P, constructing the new shape, $F = \xi[\![P]\!](S) \ \cup \ (S - T)$, where

  $$T = \{(t, u) \mid (t, u) \in S \wedge u \in \textbf{types}(\xi[\![P]\!](S))\}.$$

  The constructed shape consists of all the edges in the child's shape, plus all the edges in the original shape, except edges to types that appear in the child's shape. Below is an example `MUTATE` that moves `books` (and descendants) below `author` and its children.

  ```
  MUTATE author [ * book [ ** ] ]
  ```

If `author` is originally a descendant of `book` it is moved to being a parent along with its immediate children. The rest of the shape is unchanged.

- $\xi[\![$`DROP` $P]\!](S) = S - \{(t, \_) \mid t \in \xi[\![P]\!](S)\}$. This removes from the shape all of the edges in the meaning of P. The example below removes `titles` from `book`.
```
MUTATE (DROP title [ book ])
```
- $\xi[\![$`CLONE` $P]\!](S) = \{(\mathbf{clone}(t), \mathbf{clone}(s)) \mid t, u \in \xi[\![P]\!](S)\}$. The **clone**($t$) function creates a clone, i.e., a copy which is a distinct type, of type t. The example below copies `title`(s) and places each below an `author`.
```
MUTATE author [ (CLONE title) ]
```
- $\xi[\![$`NEW` *label*$]\!](S) = \mathbf{new}(\mathit{label} \times \{\mathit{circ}\}$. The **new** operator is used to construct a brand new type with the name *label*. The following example wraps each `author` in a `scribe` element.
```
MUTATE (NEW scribe) [ author ]
```
- $\xi[\![$`RESTRICT` $P]\!](S) = \mathbf{roots}(\xi[\![P]\!](S)) \times \{\circ\}$. This can be used to select specific types in a shape independent way. The following example chooses the `names` closest to an `author`.
```
MORPH (RESTRICT name [author]) [title]
```
- $\xi[\![$`COMPOSE` $P\ Q]\!](S) = \xi[\![Q]\!](\xi[\![P]\!](S))$. `COMPOSE` pipes the shape constructed by P into Q.
- $\xi[\![$`TRANSLATE` $D]\!](S) = Z$. `TRANSLATE` modifies the name of the type for each type in the dictionary, D, which is a total function mapping *Type* $\rightarrow$ *String*, creating the set $Z = \{(t_{new}, s_{new}) \mid (t, s) \in S\}$ where $t_{new} = \mathbf{setName}(t, D(\mathbf{baseType}(t)))$ and

$$s_{new} = \mathbf{setName}(s, D(\mathbf{baseType}(s))).$$

The translation changes the names of all of the cloned and restricted types that share the same base type. Below is an example of applying a translation to a `MORPH` to change `author` to `writer`.
```
MORPH author [name]
      | TRANSLATE author -> writer
```

VII. Rendering a Transformed Shape

If the type enforcement permits the guard to transform the data, the target shape is used to render the data. This section presents a rendering algorithm for transforming data stored as XML in the source shape to the target shape. The **Render** algorithm is shown in Figure 7. The input to **Render** is a target shape, S, an XML data model instance (a Forest), *In*, an edge in S, a set of nodes, N, and the output forest, *Out*. The algorithm is called for each root edge in S with the set of nodes, N, corresponding to the nodes of that type chosen from *In*. The algorithm recursively descends S from the starting edge and builds the output forest. For each edge in S the algorithm adds edges from parents already in the output to their closest children chosen from the source. The **sourceTypeOf**(w) function returns the type in the source for the corresponding type in the target w.

The key step in the algorithm is the "closest join," represented as $\bowtie_{CLOSE}$, which pairs up closest nodes. If the closest graph is stored the operation can simply select the desired edges from the graph. But since the closest graph has a size of $O(n^2)$ for a source of size n, it is not practical to store the graph. And in fact the closest relationships can be cheaply computed when needed in a join by reasoning about node numbers in a stored XML tree. Each node is given a prefix-based number (i.e., Dynamic or Dewey level number [6]). Since the type distance between two nodes that are closest is known *a priori*, the two nodes must have a least common ancestor at a known level in the tree, i.e., the join predicate is **distance**(n,LCA) + **distance**(u,LCA) = **typeDistance**(v,w), where $n \in N$, $u \in U$, and LCA is their least common ancestor. For example, suppose that for the data in Figure 1(a) we want to determine whether the first (leftmost) `<publisher>` (node number `1.1.3`) is closest to the first `<title>` (node number `1.1.1`), the second `<title>` (node number `1.2.1`), or both. The (minimal) type distance from `<publisher>` to `<title>` is two. So closest pairs must be at distance two. Comparing `1.1.3` to `1.1.1`, the shared prefix is `1.1`, hence the nodes are at distance two and therefore the first `<title>` is closest to the first `<publisher>`. The second `<title>` is not closest since the shared prefix of `1.1.3` and `1.2.1` is `1`, hence the nodes are at distance four which is more than the type distance required to be closest. So by reasoning about the pair of node numbers to join a standard database join can be performed to pair up closest nodes.

```
Render(Shape $S$, Forest $In$, Edge $(v, w)$, Nodes $N$, Forest $Out$)
  // Fetch nodes from the input forest for the child
  Let $U \leftarrow \{x \mid x \in In \wedge \textbf{typeOf}(x) = \textbf{sourceTypeOf}(w)\}$
  // Figure out the new, closest pairs
  Let $P \leftarrow N \bowtie_{\text{CLOSE}} U$
  // Add closest pairs to the output forest
  Let $Out \leftarrow Out \cup P$

  // Recursively, visit the children
  for each $(w, t) \in S$
    // Extend the output forest from closest nodes : $\pi_{\text{to}}(P)$
    Render$(S, In, (w, t), \pi_{\text{to}}(P), Out)$

end //Render
```
Fig. 7. Render algorithm

As an example consider the rendering of the graph in Figure 4(a) using the following guard.

```
MORPH author [ name book [ title ] ]
```

Initially, the algorithm is called with the root `<author>` nodes: {`1.1.2`, `1.2.2`}. Next the following joins occur.

1) Add `<name>` children to `<author>` nodes
   {`1.1.2`, `1.2.2`} $\bowtie_{\text{CLOSE}}$ {`1.1.2.1`, `1.2.2.1`} = {(`1.1.2`, `1.1.2.1`), (`1.2.2`, `1.2.2.1`)}
2) Add `<book>` children to `<author>` nodes
   {`1.1.2`, `1.2.2`} $\bowtie_{\text{CLOSE}}$ {`1.1`, `1.2`} = $\{(1.1.2, 1.1), (1.2.2, 1.2)\}$
3) Extend `<book>` with `<title>` children
   {`1.1`, `1.2`} $\bowtie_{\text{CLOSE}}$ {`1.1.1`, `1.2.1`} = {(`1.1`, `1.1.1`), (`1.2`, `1.2.1`)}

The **Render** algorithm can efficiently implemented by using sort-merge join and pipelining the joins. First, observe that closest joins happen only between nodes of a specific (source) type. If a sorted list of each type of node is available then closest pairs can be found by merging two sorted lists (e.g., merge the `<author>` list with the `<name>` list since the LCAs for closest pairs must also be in sorted order). Using sort-merge reduces the cost of a closest join to $O(n)$ and orders the output forest in the source's document order. Second, a closest join can start immediately upon receiving a node from a parent (it does not have to wait for entire list of nodes). So a transformation can immediately produce output, and stream the output node by node (in document order).

VIII. Architecture

An XMorph transformation has to transform the XML values within a data instance. There are (at least) three possible architectures for using XMorph to support query guards.

1) Physically transform the data. This is the architecture for the current XMorph implementation. It is the only architecture currently capable of coupling with XQuery evaluation engines and it is the most general insofar as a stand-alone transformation tool can be used in other applications. Unfortunately, this approach also has the highest cost since it involves a complete data transformation. The experiments described in Section IX compare the cost of transforming the data using the rendering of Section VII vs. a native XML DBMS. The high cost can be mitigated in several ways, e.g., by materializing the transformation and mapping XUpdate operations to updates of the transformation or by streaming the transformed data into a streaming XQuery evaluation engine.

2) Render the query guard as an XQuery view and use XQuery view rewriting to answer the query, c.f., [23]. Rendering to XQuery often creates a long, complex XQuery program since the *value* of a variable is (the text of) the entire subtree rooted at a node in the shape of the *source* rather than the desired *target*, so the source values must be teased apart and reconstructed to the target shape in the return clause piece-by-piece. Since the data must in any case be physically transformed, while there will be some speed-up over the previous approach for some queries, the worst-case cost is the same as the previous approach.

3) Logically transform the data. The first two approaches treat the XQuery evaluation engine as a black box. The alternative is to re-engineer an evaluation engine in a system to *logically* transform the data *in situ*. This alternative is the focus of our near-term development of XMorph but for the present remains future work.

The architecture for XMorph's stand-alone implementation is sketched in Figure 8. In the left of the figure, the XMorph data shredder takes XML documents and shreds them to several database tables. The *AdornedShapes* table stores information about the adorned shape of each stored document. The *Nodes* table maps a node identifier to all of the information about the node (node type, value, name, etc.). The table is used to

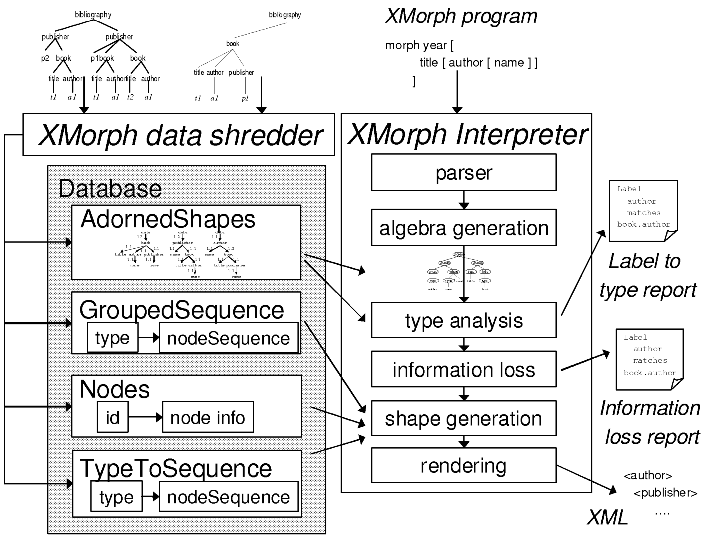


Fig. 8. XMorph implementation architecture.

construct XML for output. The *TypeToSequence* and *GroupedSequence* tables are similar. Each maps a type to a sequence of nodes. The XMorph Interpreter takes an XMorph program and evaluates it, producing XML output, and two information reports. The interpreter has several steps. First the program is parsed and a tree-like, algebraic representation of the program is generated (the algebra is described below). Next the algebra tree is analyzed to prune extraneous types and operations (also described below). Type analysis generates the *label to type* report which specifies how XMorph resolves ambiguous labels (e.g., if there are multiple `<name>` types in the data, the report describes to which type(s) the label `name` in a program maps). The type analyzed tree is then checked to determine potential information loss in a transformation as discussed in Section V-B, and the *information loss report* is produced. The algebra tree is then passed to shape generation which generates a target shape that, together with the stored data in the tables, is rendered by evaluating to XML. Prior to rendering, only the adorned shapes, which are typically tiny relative to the size of the data, are needed. *Hence rendering will be almost the entire cost of evaluating an* XMorph *program.*

XMorph programs are translated to an algebra consisting of the following operators, which are formally described in Section VI and informally described below.

- **compose**(Operator Q, Operator R) - Sequence the evaluation of Q then R, piping the output of Q into R.
- **morph**(Operator $Pattern$) - XMorph using the *Pattern*.
- **mutate**(Operator $Pattern$) - Mutate using the *Pattern*.
- **translate**(Dictionary D) - Translate using the dictionary.
- **type**(String *label*) - Select the type(s) named by *label*.
- **drop**(String *label*) - Remove the type(s) named by *label*.
- **closest**(Operator $Parent$, Operator $Child$) - Build edges between the *Parent* and *Child*.
- **clone**(Operator $Child$) - Clone the *Child*.
- **new**(String *label*, Operator $Child$) - Wrap a new label around the *Child*.

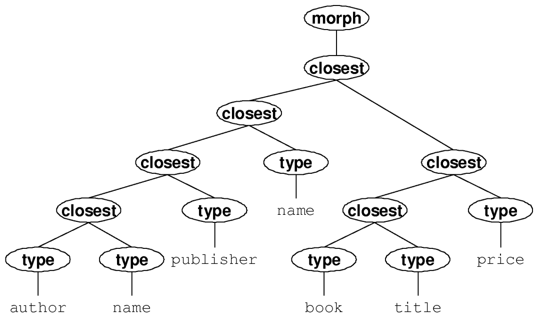


Fig. 9. XMORPH algebra for the example query.

- **restrict**(Operator *Child*) - Hide the *Child* in the result.

Translating an XMORPH query to the algebra is straightforward. An attribute grammar constructs an XMORPH algebra expression while parsing an XMORPH query. Each keyword maps to an algebraic operator. As an example, the following query is translated to the algebra shown in Figure 9.

```
MORPH author [name
   [publisher name book [title price]]]
```

After the algebra tree is constructed a *type analysis* is used to optimize it. The analysis infers the types used in every operation, potentially reducing the cost of query evaluation. The type analysis has two stages. In the first stage, type information flows up the algebra tree. In the second stage the type information is refined and pushed down to the leaves. Initially, each leaf that is a **type** operator reports that its type is the set of all possible types for a given label. These sets are then passed up the tree. When the sets reach a closest operation, two things happen. First, a **closest** operation chooses only pairs of types that are closest from among all of the possible pairs of parent and child types. For instance, in the query of Figure 9, if `author` and `name` are ambiguous, then the `author` type that is closest to some `name` type is selected. If more than one type is closest both types are used. But if some paring of `author` and `name` types is farther (in distance) than some other pairing, then it is not used. Second the type of the **closest** operation, which is the set of types of chosen parents, is passed up the tree, and type analysis continues. Once all types have been inferred, the type sets are pushed down the tree to the leaves (to avoid generating data for types which are unused higher in the tree).

IX. EXPERIMENTS

We implemented XMORPH 2.0 in Java[1]. The implementation uses ANTLR for the parser, a Xerces SAX parser for data shredding, and BerkeleyDB Java Edition for the data store. To quantify the cost of a data transformation we performed several experiments. The experiments were run on a dual processor Linux machine, with two Intel 686 2.66GHZ chips, 3.5GB of RAM, and mirrored 500 GB disks (RAID level 1). In each experiment we isolated the testing machine. We ran each experiment five times and chose the median cost. The cache was cleared for each run so the timings are "cold cache" numbers in every experiment.

Cost of transformation vs data size. The first experiment measures the impact of increasing data size on a data transformation. The experiment uses the XMark benchmark. We generated five documents, benchmark factors .1 through .5, representing document size of 11MB to 55MB. Internally, each document has 471 distinct types. We evaluated the cost mutating the entire document using the transformation `MUTATE site` (the mutated shape has all 471 types). For comparison we also evaluated the cost of performing a *lessor* task in eXist, version 1.4.0. eXist is a native XML DBMS implemented in Java. We tested using eXist's local `xmldb` API (i.e., we did not use the client-server setup in eXist which is slower). For eXist rather than using an XQuery query equivalent to the XMORPH mutation (which would have one variable for every type or 471 variable bindings in for/let clauses!) we used the following query, which simply dumps the entire document.

```
for $b in doc("xmark.xml")/site
return <data>{$b}</data>
```

The results are plotted in Figure 10. The 'XMORPH render' plotline represents the cost of rendering the data as XML. The 'XMORPH compile' plotline is the cost of the XMORPH interpreter prior to rendering, which *includes the cost of checking for information loss*. Finally, the 'eXist plotline' represents the cost of evaluating the above XQuery program in eXist. The 'eXist plotline' is the "best case" scenario for eXist. eXist internally stores an XML document in document order on disk pages, so the timing is essentially that of reading the document from disk to a String object. XMORPH on the other hand, constructs the result during evaluation. *Observe that the cost of the* XMORPH *render increases linearly with the document size, and that the cost of compiling an* XMORPH *transformation (checking for potential information loss) is a tiny fraction of the transformation's overall cost.* At a benchmark factor of 0.1, the compilation cost is 0.002% of the overall cost, and the percentage decreases as the data size increases. The raw cost of the compilation is 20 milliseconds for each factor. The cost of shredding the data to XMORPH's data store is not included in the figure. The shredding is done once and then multiple transformations can be applied to the stored data. The documents took 20, 43, 67, 85, and 115 seconds to shred as the benchmark factor increased.

To measure the impact of XMORPH on system resources we collected statistics using the Linux tool `vmstat` on disk use (block I/O), CPU utilization, and memory use as the experiment ran. Figure 11 plots the cumulative block I/O for each document factor in the experiment. The cumulative I/O is all blocks sent to and received from both disks (the disks are mirrored). The block I/O is steady throughout the experiment, with no sudden increases. *This shows that* XMORPH *is gradually processing the disk tables and generating output as the experiment runs.* Figure 12 plots the *wait percentage*. The

[1]XMORPH is open source and is available at `http://www.cs.usu.edu/~cdyreson/pub/XMorph`.

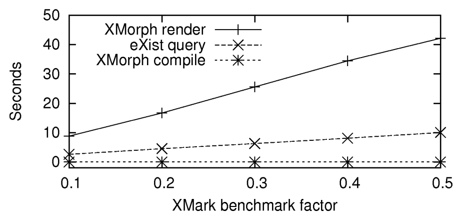


Fig. 10. Transforming XMark documents.

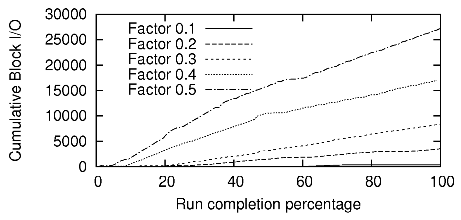


Fig. 11. Block I/O during experiment one.

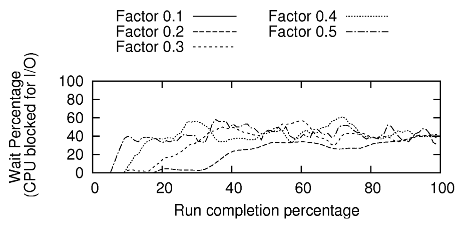


Fig. 12. Percent CPU blocked, waiting for block I/O.

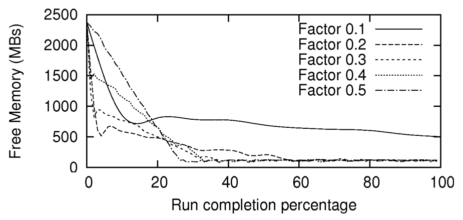


Fig. 13. Available memory.

wait percentage is the percentage of the time that the CPU was idle because it was blocked waiting on I/O. The graphs show that roughly 40% of the CPU time is spent waiting, i.e., the block I/O drives the cost of a transformation. Lastly, Figure 13 plots available memory. As XMORPH executes on the JVM, Java grabs all available memory within the first 30% of an experiment. Overall, the experiment exercised the system sufficiently to obtain "good" timings that are not dependent on caching effects (for XMark factors above 0.1 since the cumulative disk I/O and caching graphs show caching effects for the 0.1 dataset, observe that memory is available, cumulative block I/O is tiny, and wait percentage is close to zero, which is difficult to discern in the graph) and scale to larger datasets.

We decided to get a more accurate comparison and exercise XMORPH on larger datasets in a second experiment. We used slices of `DBLP.xml`, which roughly has the shape shown in Figure 1. The tested document sizes were 134MB, 268MB, 402MB, and 518MB. We tested three transformation sizes for each document size: small (`MORPH author`), medium (`MORPH author [title [year]]`), and large (`MORPH dblp [author [title [year [pages] url]]]`). The results are plotted in Figure 14. Though as the transformations become larger XMORPH outperforms eXist, both systems play different, complementary roles.

Effect of target shape. In this experiment we measure whether the kind of target shape matters in a transformation. Since XMORPH constructs the target in a single pass from lists of elements created when the data was parsed, the shape of the input and output should be irrelevant to the speed a transformation, rather only the size of the output should matter. In this experiment we used three datasets: 1) 23MB of astronomy data from NASA, 2) 112MB of conference papers from DBLP, and 3) 55MB of XMark data (benchmark factor of 0.5). We wrote XMORPH transformations to transform the input to different shapes ranging from a deep (skinny) tree to a bushy tree. We tested each kind of shape with two shape sizes: the *small* shape has four to six labels, while the *large* has ten to twelve labels. The results are plotted in Figure 15. Since the output size in each transformation is different, the y-axis in the graph plots the *throughput* (number of elements processed per second). *The measurement shows that the throughput remains steady across each dataset.* Variations between the datasets are due to differences in the size of element content (larger text content leads to slower times).

Cost of XMORPH operations. We now study the cost of each kind of operation in XMORPH. A transformation could have a number of different operations, such as `MUTATE`, `TRANSLATE`, etc. As discussed in Section VI these operations are compiled into a new shape, which is subsequently used to render the data. So their impact, in general, on transformation evaluation should be small. Figure 16 plots the result of evaluating different transformations `COMPOSE`d with a single `MORPH` on the XMark dataset (the same `MORPH` was used in each test to ensure that the size of the output is the same). The figure shows that *the cost of each operation is effectively the same*, and that operations like translating a label or adding a new label add little to the run-time cost.

X. CONCLUSIONS AND FUTURE WORK

XQuery is precise but brittle. An XQuery programmer can use path expressions that precisely locate data. But a programmer has to be familiar with the shape of the data to query it effectively. And if that shape changes, or if the shape is other than what the programmer expects, then the query may fail. We propose using a *query guard* to both transform the data to the shape expected by the query and to protect the query by determining whether and how the transformation potentially loses information; a transformation that loses information may lead to a query yielding an inaccurate result. We make three primary contributions. First we formally describe the semantics of query guards for XML as a way to both protect a query and automatically transform data to the shape needed by a query. The key contribution of the semantics is that it separates data from its shape. A query guard uses, mutates,

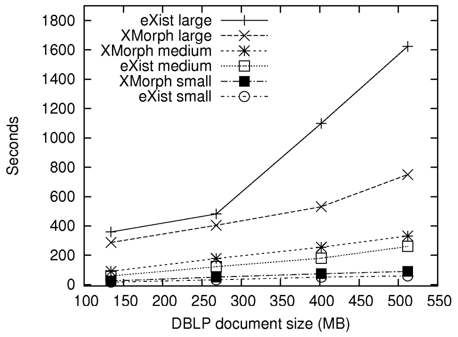


Fig. 14. Transforming DBLP documents.

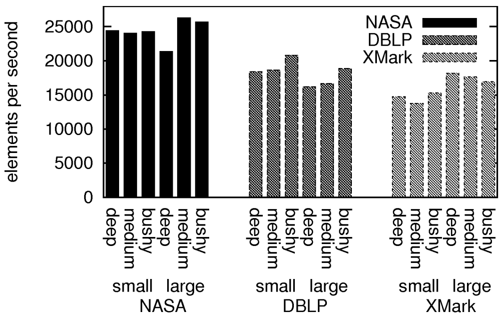


Fig. 15. Different target shapes.

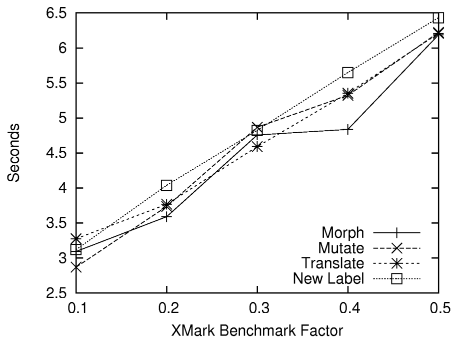


Fig. 16. Different operations.

and extends the shape into a shape desired by the user. The constructed shape is subsequently used to render the data. Second we develop a framework in which a guard can be used as a type specification, i.e., we can formally determine whether and how a transformation will potentially lose data. Third we show how to transform the data in a single pass over the source data.

There are several related problems that we plan to explore in future. The first problem is that a query guard is simple to specify, but the simplicity masks a semantically challenging problem: how to best match the *labels* in the guard to the *element types* in the data. The labels could be *ambiguous*, e.g., does "`name`," in `MORPH author [ name ]` refer to an author's name or a publisher's name? We resolve the ambiguity by matching each `author` type to the closest `name` type, and the evaluation of a query guard generates a detailed report of how labels in the guard are matched to types. So the query guards presented in this paper employ a *syntactic match*, addressing *semantic mismatch* is orthogonal to the research presented here. The second orthogonal problem is *guard inference*, that is, whether a guard can be automatically generated from a query [24]. The third orthogonal problem is how to quantify the amount of potential information loss. We articulated four "coarse" kinds of information loss, but these could be refined, e.g., the transformation manufactures 30% new information.

References

[1] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree Pattern Relaxation. In *EDBT*: 496–513, 2002.

[2] M. Arenas and L. Libkin. A Normal Form for XML Documents. *ACM Trans. Database Syst.*, 29(1), 2004.

[3] N. Augsten, M. H. Böhlen, and J. Gamper. The q-gram distance between ordered labeled trees. *ACM Trans. Database Syst.*, 35(1), 2010.

[4] A. Balmin, L. S. Colby, E. Curtmola, Q. Li, and F. Ozcan. Search Driven Analysis of Heterogenous XML Data. In *CIDR*, 2009.

[5] M. Bhide, M. Agarwal, A. Bar-Or, S. Padmanabhan, S. Mittapalli, and G. Venkatachaliah A. Balmin, L. S. Colby, E. Curtmola, Q. Li, and F. Ozcan. XPEDIA: XML ProcEssing for Data Integration. *PVLDB* 2(2):1330–1341, 2009.

[6] T. Böhme and E. Rahm. Supporting Efficient Streaming and Insertion of XML Data in RDBMS. In *DIWeb*: 70–81, 2004.

[7] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *CACM*, 13(6): 377–387, 1970.

[8] S. Cohen and T. Brodianskiy. Correcting Queries for XML. *Inf. Syst.*, 34(8): 690–710, 2009.

[9] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. In *VLDB*: 45–56, 2003.

[10] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful Database Schema Evolution: The Prism Workbench. In *VLDB*, 2008.

[11] C. Dyreson, S. Bhowmick, A. Jannu, K. Mallampalli, and S. Zhang. XMorph: A Shape-polymorphic, Domain-specific XML Data Transformation Language. In *ICDE*: 844–847, 2010.

[12] C. Dyreson and S. Zhang. The Benefits of Utilizing Closeness in XML. In *DEXA Work.*: 269–273, 2008.

[13] C. E. Dyreson, S. S. Bhowmick, and K. Mallampalli. Using XMorph to Transform XML Data. *PVLDB*, 3(2): 1541–1544, 2010.

[14] R. Fagin, L. Haas, M. Hernandez, R. Miller, L. Popa, and Y. Velegrakis. Clio: Schema Mapping Creation and Data Exchange *LNCS* 5600: 198–236, 2009.

[15] W. Fan and P. Bohannon. Information Preserving XML Schema Embedding *TODS* 33(1): 4, 2008.

[16] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*: 436–445, 1997.

[17] H. Jiang, H. Ho, L. Popa, and W.-S. Han Mapping-driven XML Transformation. In *WWW*: 1063-1072, 2007.

[18] Y. Kanza and Y. Sagiv. Flexible Queries over Semistructured Data. In *PODS*, 2001.

[19] S. Krishnamurthi, K. E. Gray, and P. T. Graunke. Transformation-by-Example for XML. In *PADL*: 249–262, 2000.

[20] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*: 72–83, 2004.

[21] Z. Liu, J. Walker, and Y. Chen. XSeek: A Semantic XML Search Engine Using Keywords. In *VLDB*: 1330–1333, 2007.

[22] T. Pankowski. A High-Level Language for Specifying XML Data Transformations. In *ADBIS*: 159–172, 2004.

[23] Y. Papakonstantinou and V. Vassalos. Query Rewriting for Semistructured Data. In *SIGMOD Conference*: 455–466, 1999.

[24] Y. Papakonstantinou and V. Vianu. DTD Inference for Views of XML Data. In *PODS*: 35–46, 2000.

[25] A. Termehchy, M. Winslett, and Y. Chodpathumwan. How Schema Independent Are Schema Free Query Interfaces? In *ICDE*, 2011

[26] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping Adaptation under Evolving Schemas. In *VLDB*, 2003.

[27] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *SIGMOD*: 537–538, 2005.

[28] S. Zhang and C. E. Dyreson. Symmetrically Exploiting XML. In *WWW*: 103–111, 2006.