

Morph: A (Shape) Polymorphic XML Query Language

Curtis Dyreson
Utah State University
Logan, Utah

Curtis.Dyreson@usu.edu

Sourav Bhowmick
Nanyang Technological University
Singapore

assourav@ntu.edu.sg

Aswani Rao Jannu
Utah State University
Logan, Utah

aswani.jannu@aggiemail.usu.edu

Kirankanth Mallampalli
Utah State University
Logan, Utah

kirankanth.mallampalli@aggiemail.usu.edu

Shuohao Zhang
Marvel
San Jose, California

shuohao@msn.com

ABSTRACT

By imposing a single hierarchy on data, XML makes queries *brittle* in the sense that a query might fail to produce the desired result if it is executed on the same data organized in a different hierarchy, or if the hierarchy evolves during the lifetime of an application. This paper presents a new XML query language, called Morph, which supports more flexible querying. Morph is a shape polymorphic query language, that is, a single query can extract relevant data from a variety of differently hierarchies. The Morph data model distills an XML data collection into a graph of *closest relationships*. This paper describes the syntax, semantics, and a prototype implementation of Morph.

Categories and Subject Descriptors

H.2.3 [Database Management]: Query languages – XML.

General Terms

Design, Languages.

Keywords

XML, query language, polymorphism, algebra.

1. INTRODUCTION

In 1970, E. F. Codd critiqued the hierarchical model because it needs *asymmetric* path expressions to locate data [1]. A path expression is a specification of a path (or a set of paths) in a hierarchy. Asymmetric path expressions depend on how the data in a hierarchy is structured, but the *same* data can be organized in *different* hierarchies. Codd presented five reasonable hierarchies for a simple part/supplier data collection and demonstrated that, in general, a path expression formulated with respect to one hierarchy would fail on some other hierarchy.

Many years later a hierarchical data model has resurfaced with the advent of XML. At a logical level, an XML data model is a tree-like, hierarchical model. As a consequence, asymmetric path expressions have reappeared in XML query languages. The core of most XML query languages is a path language to navigate to

various places in a hierarchy; for XQuery the path language is (a subset of) XPath.

There are five core areas where asymmetry in XML adversely impacts XML applications.

- 1) **Queries are less portable.** Asymmetric path expressions make XQuery queries *brittle* in the sense that a query might fail to produce the desired result if it is executed on the same data organized in a different hierarchy. Queries often have to be reformulated for each new data collection even when the data and schema “vocabulary” (i.e., the names of the elements and attributes) is the same.
- 2) **Potential failure to integrate heterogeneous hierarchies.** In data integration and change detection scenarios, finding the same data from among multiple data sources is critical. But when the sources have different hierarchies it becomes much harder to identify which data is the same, potentially leading to a failure to integrate data.
- 3) **Steep learning curve of a hierarchy.** Detailed knowledge of the data’s hierarchy or schema is often needed in order to correctly formulate an asymmetric path expression. Many data collections lack a schema, and even when a schema is present, it may be complex and difficult to decipher for some users.
- 4) **Queries fail to adapt to ad-hoc hierarchies.** The decentralized nature of the web has facilitated a growth in the generation and exchange of data authored by casual users. More often than not, data provided by these users does not conform to a strict schema; rather the data in a single collection is *irregularly structured*. Depending on asymmetric path expressions alone may miss relevant data because the same collection has varying hierarchies.
- 5) **Queries have a short shelf-life.** Hierarchies evolve. Shifts in business strategy and corporate environments often engender evolution in how data is organized. *Legacy* path expressions that depend on a particular hierarchy may no longer work when a schema evolves.

The common theme underlying these various scenarios is that there are drawbacks to *tightly coupling* queries to hierarchies.

In this paper we propose a new query language, called Morph, which *de-couples* XML queries from XML hierarchies. In principle, de-coupling will make it easier for a user to query and transform XML data. Morph was designed to have the following features.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLAN-X Workshop '08, January 24, 2009, Savannah, GA, USA.
Copyright © 2009 Curtis Dyreson et al..

- 1) **Easy to transform the data's shape.** The primary component of Morph is a *morph* in which the user declares the desired shape or hierarchy of the result. Morph reorganizes the source data to match that shape. [4] and [8] propose declarative languages for specifying transformations of XML data. These languages hide from users much of the specification details necessary in a language such as XQuery or XSLT. However, these special-purpose techniques are limited because they are tree structure-dependent. A transformation query might have to be rewritten for a different tree.
- 2) **Shape Polymorphic.** Shape polymorphism was first described by Jay and Crockett [3] (and extended to structural polymorphism by Reuhr [9]). In shape polymorphism a function, e.g., to print a value, adapts to the shape of the data, e.g., adapts to a tree or a list by visiting all the values in the data structure. We can apply this notion to database query languages as follows: a query language is *shape polymorphic* if any query evaluated on the *same* data in *different* structures yields (approximately) the *same* result.¹ Shape polymorphism is sketched in Figure 1. A query evaluated by a query engine over several different structures yields the same result. XQuery, like all other languages that rely on asymmetric path expressions, is not shape polymorphic. XQuery extensions have been proposed that support shape polymorphism, namely Schema-free XQuery [6] and the closest axis [10], but Morph is not an XQuery extension, and its semantics and implementation differs from Schema-free XQuery.
- 3) **Tree-like Syntax.** Morph queries have a tree-like syntax so the tree transformations of Morph can be applied to Morph queries. For instance, Morph query optimization rules are expressible in Morph.
- 4) **Ability to treat attributes as indistinct from subelements.** Data modelers often arbitrarily choose to use attributes rather than subelements, or vice-versa. The goal of Morph is to decouple queries from hierarchies so it is important to avoid *forcing* users to differentiate (syntactically) between attributes and subelements.
- 5) **Easy creation of groups.** While XQuery has only ad-hoc support for groups using a distinct-values function. Morph supports both persistent and dynamic group creation.
- 6) **Vocabulary translation.** To use Morph, all a user has to know is the “vocabulary” (e.g., the names of the elements) in an XML data collection. But Morph also supports vocabulary translation, so that users can change queries and data to their terminology.
- 7) **Data streaming.** Morph's is designed to generate output as a data stream and to read input from a data stream so that data can be transformed in a single pass as envisioned by the STX (Streaming Transformations for XML) working group².
- 8) **Shape reflection.** XQuery queries are unable to discover the shape of data, but Morph supports a limited kind of shape reflection.

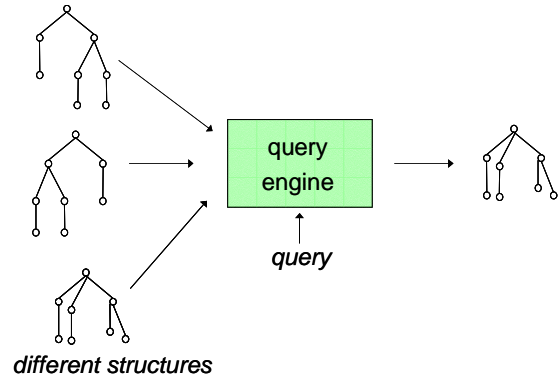


Figure 1 Shape polymorphism, where the same query extracts the same data from different hierarchies

The goal of making it easy to query a data collection is shared by XML search engines [2]. Search engines have a simple, easy-to-use interface. Like Morph, they de-couple queries from specific hierarchies. But unlike Morph, XML search engine queries typically only involve values, and not the schema “vocabulary.” Morph straddles the middle ground between XML search engines and path expression-dependent XML query languages by borrowing useful techniques from each end of the spectrum.

2. MORPH OVERVIEW

This section gives a short tutorial on Morph through a series of examples of increasing complexity. Figure 2 gives the ANTLR syntax for Morph (leaving out the token definitions and where condition). The examples will transform the data about books written by E. F. Codd shown in Figure 3.

```

program :
    function ( '|' function )? EOF ;

function:
    ('morph' | 'mutate') pattern
    | 'data' ( STRING | '{' function '}' )
    | 'translate' dictionary ;

pattern :
    ID ( '.' ID )* ( ',' modifier)?
    ( '[' pattern+ ']' )? ;

modifier:
    'hide' | 'clone' | 'optional'
    | 'where' condition
    | 'group' ( '(' pattern+ ')' )?
    ( ',' modifier)? ;

dictionary:
    pattern '->' pattern ( dictionary )? ;

```

Figure 2 (Part of) ANTLR Syntax for Morph

¹ The same result *modulo* duplicates, ordering, and attribute, subelement swaps. For more details see Section 3.

² <http://stx.sourceforge.net/documents>

Though database query languages tend to be *declarative* rather than *functional*, Morph is a functional query language (i.e., similar to a query algebra rather than a query calculus).

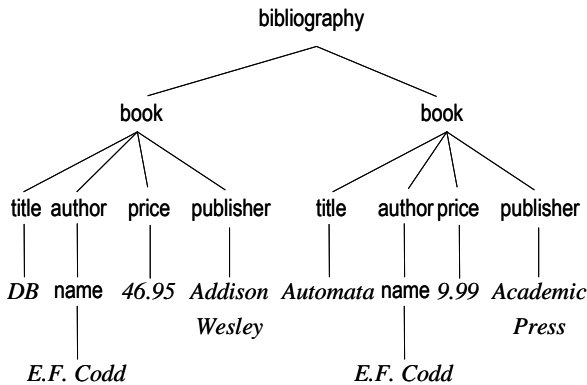


Figure 3 Authors listed by book

The primary function in Morph is a *morph*, which places children below a parent in the result. The parameter of the morph function is a *pattern*, which specifies the structure of the result. Figure 4 gives a simple example. The query is intended to list the titles written by each author extracted from a collection of book data. The pattern specifies that <title> and <name> elements are placed as children of <author> elements.³ Only <title> and <name> elements that are *closest* to an <author> element are placed within that <author>. The notion of *closeness*, which forms the core semantics for Morph, is explained in detail in Section 3, but intuitively the idea is that authors are closely related to the titles of their own books and articles (and their own names), but not close to titles written by others (or the names of others). Figure 4 also shows the result of the query when evaluated on the data in Figure 3.

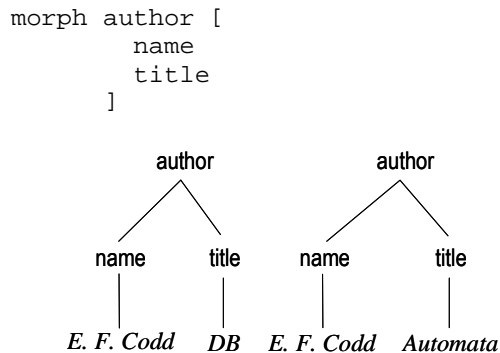


Figure 4 List titles by author, and result

A morph can be restricted to select individual authors. Suppose we want only the titles by the author E. F. Codd. Then we can use the query given in Figure 5 which selects <author> elements where the value of the element is 'E. F. Codd'. The result is the

same as that in Figure 4 since only E. F. Codd has authored books in the source data.

```
morph author [
  name, where value = 'E.F. Codd'
  title
]
```

Figure 5 List titles by author E. F. Codd

There may be duplicate authors in the data, but authors can be *grouped* to eliminate the duplicates. Figure 6 shows an example that consolidates titles under a single E. F. Codd author using a 'group' modifier for both the author and his titles. Modifiers are listed after a label, separated by commas. The group modifier uses the default, persistent grouping for author (e.g., author is grouped by its 'key' as specified by the data's schema, or by the distinct-values function for a schema-less data collection). Dynamic grouping, that is, grouping during query evaluation according to a given pattern, is also supported.

```
morph author, group [
  name, where value = 'E.F. Codd'
  title
]
```

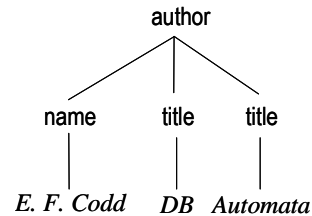


Figure 6 List titles grouped by author E. F. Codd

E. F. Codd wrote both books and articles, and we may want to select only book titles. In the query given in Figure 7, <title> elements closest to a <book> element are selected but books are *hidden* in the output; a <book> is only used to find a closely-related <title>. The result is the same as that in Figure 4 since the data has only <book>s.

```
morph author, group [
  name, where value = 'E.F. Codd'
  title [ book, hide ]
]
```

Figure 7 List book titles grouped by E. F. Codd

Though Morph assumes that a user is familiar with the vocabulary of the data, it also has a *translate* function that translates the query or the result into the terms desired by the user. The translation can be specified before a morph, with the output of the translation being piped into the morph (as shown in Figure 8) or after a

³ In the explanation of this example, we've assumed elements rather than attributes, but, in general, "author", "name", or "title" could be either an attribute or element.

morph, in which case the output of the morph is piped into a *translate* function.

```
translate
  author -> writer
| morph
  writer [
    name, where value = 'E.F. Codd'
    title [ book, hide ]
  ]
```

Figure 8 List book titles by the writer Codd

The *pipe* operator, '|', is used to connect the output of one Morph function into the input of another function. Initially, the input is assumed to come from a *default* data collection, but the data collection could be explicitly named using a *data* specification as illustrated in Figure 9.

```
data 'dblp.xml'
| morph author [ name [ title ] ]
```

Figure 9 List titles by author from dblp.xml

To this point, the descriptions of the queries have avoided describing the shape of the input, that is, the same query could be applied to data in a variety of hierarchies. Moreover each query, when applied to the same data in different hierarchies will produce the same output. The only hierarchy that the user needs to specify is that of the output. To illustrate this, consider the query shown in Figure 10. The query applies a *morph* to the result of a *morph*. The first *morph* produces a hierarchy which lists the titles published in each year and within each title the authors for that title. The second morph is the transformation of Figure 4.

```
data {
  morph year, group [
    title [ author [ name ] ]
  ]
} | morph author [ name title ]
```

Figure 10 Morphing a morph

Morph also supports *mutation* of the data's hierarchy. A mutation is like a morph in that it changes the shape of the hierarchy, but unlike a morph, the entire hierarchy is implicitly involved. Figure 11 sketches a mutation which explicitly lists only three types of elements. The mutate outputs the entire hierarchy, with two mutations. First it moves the <publisher> elements to within the closest <author> elements, and second, it clones <title> elements to also place them under the closest <author> (the uncloned <title> elements will remain in place). The rest of the hierarchy is not changed.

These examples show a few of the uses of Morph and illustrate its most important design feature: shape polymorphism. In a shape polymorphic query language users specify only what they want as output. A query adapts to the shape of the input data to produce

the desired output. In Morph, this adaptation is based on the notion of closeness, which is described next.

```
mutate author [
  publisher
  title, clone
]
```

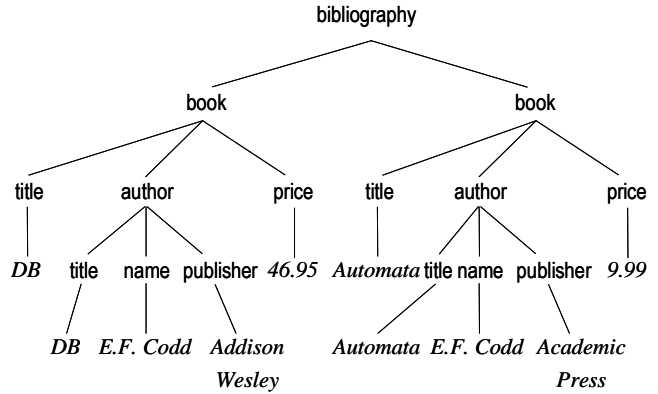


Figure 11 Mutating the data

3. CLOSENESS

While XQuery path expressions are wedded to a particular hierarchy, the key to developing a technique that works for any hierarchy is to identify what is *invariant* across the “same” data organized in different hierarchies. Observe the two hierarchies for the same book data in Figure 3 and Figure 12. In each of the hierarchies, the book titles by an author are *closest* to that author. Here “closeness” is roughly defined as the distance on the path between nodes in the hierarchical model of an XML document (details and a formal definition of closeness are given elsewhere [10][11]). This is not something specific to authors and titles only. In fact, whenever two nodes are closest in Figure 3 so are their counterparts in Figure 12.

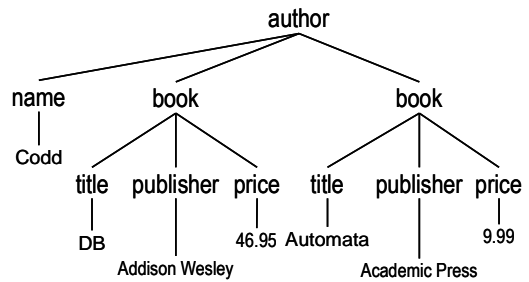


Figure 12 Books listed by author

To better understand the use of closeness, let's consider the example shown in Figure 13. Assume that the *type* of each node is its label. Let the context node be the leftmost title node. The nodes closest to the title node are pointed to by dashed arrows. (In this example there is only one node of each other label that is closest, but in general there could be several nodes with the same label that are closest.) Note that none of the nodes in the other book subtree is closest to this title.

It is not quite that simple because many hierarchies are irregular in the sense that some closest nodes are “missing.” Consider the example given in Figure 14. Most books have a price, but the price information is missing from the leftmost book in the figure. So the closest price might be in the rightmost book, but this connection is restricted by using the type information; in effect the closest price to the title node is a missing price. Closeness is a refinement of the related notion of the *meaningful least common ancestor* which provides the basis of Schema-free XQuery [6].

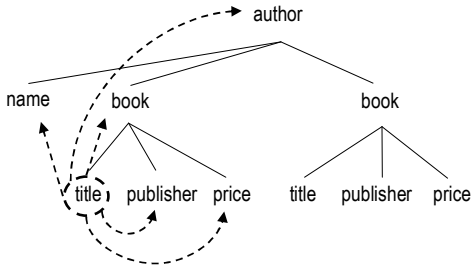


Figure 13. Nodes closest to the first title

For closeness to be of practical value for database applications, it needs to be computed efficiently in persistent implementations, as described in the next section.

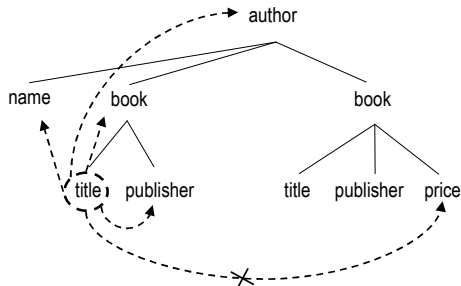


Figure 14. Closeness is restricted by the type

4. MORPH DATA MODEL

The Morph data model is an undirected graph. Each node in the graph corresponds to an element or attribute, and has six pieces of information.

- 1) **Document.** Each node belongs to some document or data source. We assume each document has a unique name. At present, Morph assumes a flat space for document names, though we plan to add support for collections.
- 2) **Type.** By default, the type is the concatenation of labels on the path from the root to the node, though other notions of type could be utilized, e.g., the type could be specified in a schema or involve descendent nodes; we merely assume that each node has a well-defined type in the data collection. The type does not include the name of the document to which the node belongs.
- 3) **Label.** The label is the tag name of an element or name of an attribute.
- 4) **Kind.** The kind is either attribute or element.

- 5) **Value.** A string value associated with the node. The value of an element is the concatenation of its immediate text content. For an attribute it is the value of the attribute.

The graph has two kinds of edges: *closest* and *group*.

- 1) **Closest.** A closest edge represents a closest relationship between a pair of nodes, e.g., there is a closest edge from node X to node Y if $closest(X,Y)$. Closest edges are intra-document edges.
- 2) **Group.** A group edge is a (persistent) group relationship, e.g., there is a group edge from node X to node Y if $type(X) = type(Y)$, $value(X) = value(Y)$, and $value(X)$ is not empty. Group edges can span across document boundaries.

Figure 15 depicts the data model for the data of Figure 3. In the figure, a solid line represents a closest edge, while a dashed line indicates a group edge.

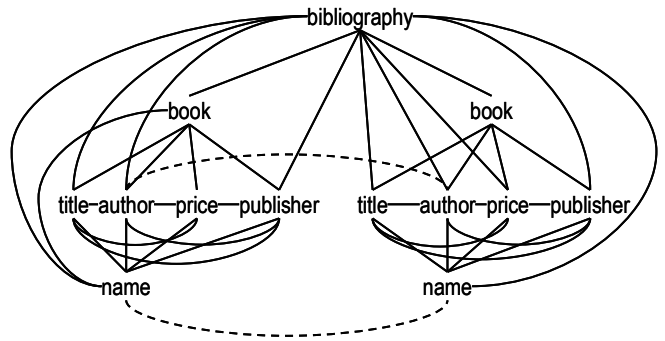


Figure 15 Data model instance for the data of Figure 3

The data model is easy to construct in a single pass using a SAX parser, though the parser has to compute and maintain groups while shredding.

5. MORPH ALGEBRA

Morph queries are translated to an algebra consisting of the following operations. Let L be a label corresponding to the set of types $T \subseteq \{ t \mid t = L_1.L_2. \dots .L \}$ and let S_T be a sequence of nodes (output from some other operation) of types T .

- **type(L) = S_T** – Generates a sequence of nodes, S_T , of types T . The label, L , may be ambiguous, e.g., there may be several types for a label such as ‘author’ in the data collection. Morph disambiguates the type by choosing it from among all of the closest parent/child distances in a Morph pattern (the body of a *morph* or *mutate*) as described in Section 5.1. Alternatively, the user can disambiguate the type by giving a more precise label, e.g., book.author vs. journal.author.
- **closest(S_p, S_c) = S_p** – Generates a sequence of parent nodes, S_p , with closest child nodes chosen from the sequence S_c . This operation will be efficiently implemented as an LCA-join, described below.
- **grouped(S_p) = S_G** – Generates a sequence of unique P nodes, where P is grouped by its default, persistent grouping.

- **group**(S_P) = S_G – Generates a sequence of unique P nodes, where P is dynamically grouped by a pattern.
- **clone**(S_T) = S_C – Generates a cloned sequence, S_C , from S_T .
- **hide**(S_T) = S_T' – Generates a sequence of nodes, where each node is marked as hidden.
- **where**(S_T, c) = S_T' – Generates a sequence of nodes, where a node is in the sequence only if condition c evaluates to true for the node's value.
- **translate**(S_T, c) = S_T' – Translate each node label in S_T to m .

Note that all of these operations are *generators*, operating on node sequences.

Let's consider two example translations to see how Morph queries are compiled to the algebra. Consider the simple query given in Figure 4. It translates to the algebra shown in Figure 16.

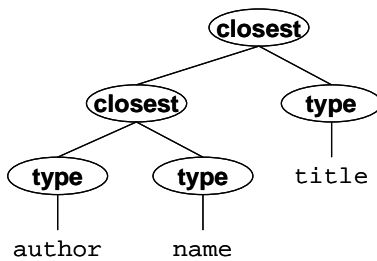


Figure 16 Morph algebra for the query in Figure 4

As a second example, Figure 17 shows the algebra for the query in Figure 7.

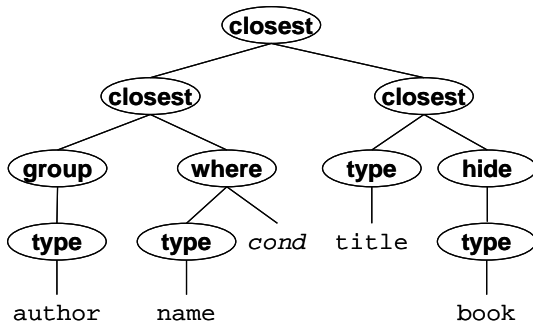


Figure 17 Morph algebra for the query in Figure 7

5.1 Type Analysis and Query Evaluation

A query is evaluated in two stages. In the first stage, a *type analysis* infers the type of every operation. The type analysis first flows up the tree and then back to the leaves. Initially, each leaf that is a **type** operation node 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 node, three 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 4, 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 pairing of 'author' and 'name' types is farther (in distance) than some other pairing, then it is not used.

Second the *least common ancestor type* of each pair is determined. Third, the type of the **closest** operation node, 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 nodes for types which are unused higher in the tree).

As an example, consider Figure 18. In the figure the algebra tree is adorned with the types generated by each node in the first phase of type analysis. (For this example, we assume many types exist, not just those present in the running examples.) The topmost **closest** node generates only the type book.author (with children of type book.author.name and book.title). The second phase of type analysis pushes the inferred types down the tree, allowing the leaves to discard unused types as shown in Figure 19.

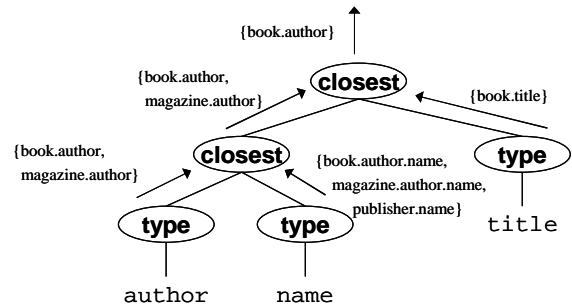


Figure 18 Types are inferred up the algebra tree

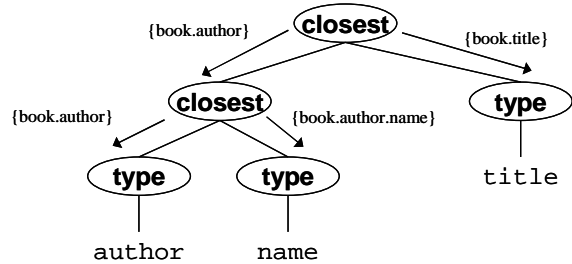


Figure 19 Needed types are pushed down, allowing unused types to be pruned at the leaves

In the second stage of query evaluation, the result is generated. Each node behaves as a generator and generates the next node in its output sequence in response to a request. So in Figure 16, the first result is computed by requesting the top **closest** node to compute a result. It in turn asks each of its children to produce a result, and they in turn ask their children, etc.

5.2 Limited Reflection

The algebra for a *mutate* function is more complicated because the shape of the input must be duplicated, which basically means that many **closest** operations are implicitly specified. Figure 20 shows the algebra for the query given in Figure 11 assuming that the data's shape is as shown in Figure 12. Note that the query mutates the structure to move publishers under author and clones title to also be a child of author (the original titles remain within book).

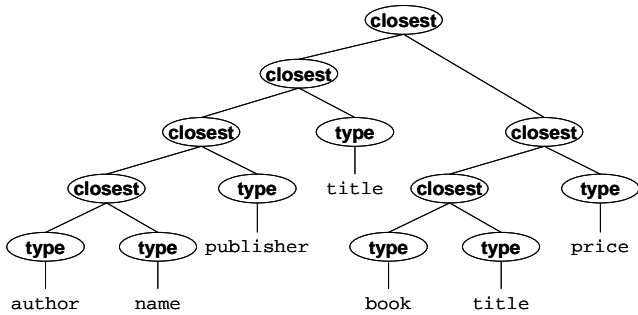


Figure 20 Morph algebra for the query in Figure 11

5.3 LCA-Join

Examining the algebra for each Morph query shows that the **closest** operation frequently appears. Elsewhere, we introduced an *LCA-join* operation that efficiently finds the closest nodes [10], and implemented the closest axis in a tree-unaware XML DBMS [5]. Below we describe how the LCA-join is used to implement the **closest** operation.

We can use a simple node numbering scheme and indexes to quickly evaluate queries that utilize LCA-joins on sequences, especially sequences generated by **closest** operations. We use the following scheme. Given a tree, we assign each node a number according to its ordinal in document order. The numbers range from 1 to n , the total number of the nodes. This can be achieved by a preorder traversal of the tree. Each node is also assigned the number of its maximum descendent, so that it can quickly be determined which nodes are descendants of a given node. Figure 21 shows the numbering for the data tree of Figure 12. The first book node has the number 3 and its maximum descendent is 6. So its descendants are all the nodes numbered between 3 and 6.

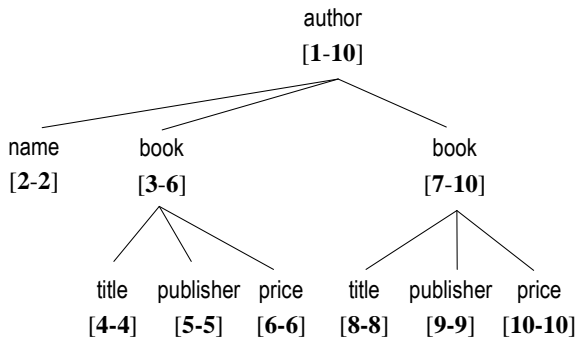


Figure 21. Numbering the tree of Figure 12

Next, an index of *types* is created.⁴ The index maps each type to an ordered sequence of node numbers for nodes of that type. The closest nodes can be computed by simply merging three sequences as depicted in Figure 22. The sequence merging is an LCA-join.

⁴ The type could be specified in a schema, assumed to be the concatenation of labels on the path from the root to a node, or involve descendent nodes; we merely assume that each node has a well-defined type in the data collection.

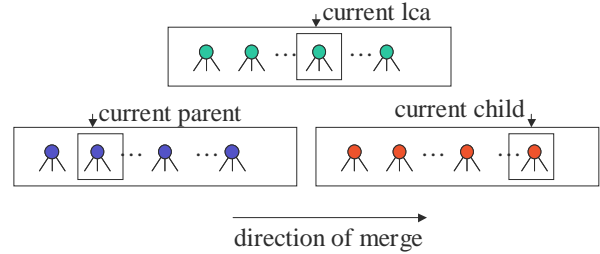


Figure 22. An LCA-join

In the figure, there are three sequences of nodes: *parents*, *children*, and *least common ancestors (lca)* in short). The parents sequence is the sequence of nodes generated by the left child in a **closest** operation. The children sequence is the nodes generated by the right child. The lca sequence corresponds to the type for the lca. For instance, for *title* children and *publisher* parents in Figure 12, the lca is *book*. The sequences 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 descendants 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. Often only two sequences are merged instead of three since the parent or child is often also the lca.

6. IMPLEMENTATION

We are in the process of implementing a prototype of Morph. The architecture is sketched in Figure 23. In the left of the figure, the Morph data shredder takes XML documents, shreds them to several database collections. The *Shape Metadata* collection stores information about the types, labels, and shape of each stored XML 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 only in the evaluation of **where** conditions, and to construct XML for output. The two other tables, *TypeToSequence* and *GroupedSequence*, are similar. Each maps a type to a sequence of nodes. They are used to evaluate **type** and (persistent) **group** operations, respectively. The Morph Interpreter takes a Morph query, parses it, translates the parse tree to an algebra tree, which is subsequently evaluated. Results are formatted as XML.

Currently we've implemented the parser (using ANTLR), the algebra code generation, and sequenced evaluation. We've implemented exclusively in Java, but only have an in-memory prototype. We are in the process of converting the in-memory system to use BerkeleyDB as the back-end data store, and separately exploring how to translate the tree algebra to an efficient RDBMS query evaluation plan. We plan to report on and demo a complete, persistent implementation at the workshop.

7. SUMMARY

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 effectively query it. And if that shape changes, or if the shape is other than what the programmer expects, then the query may fail. An XML search engine is easy but imprecise. Not much is required of search

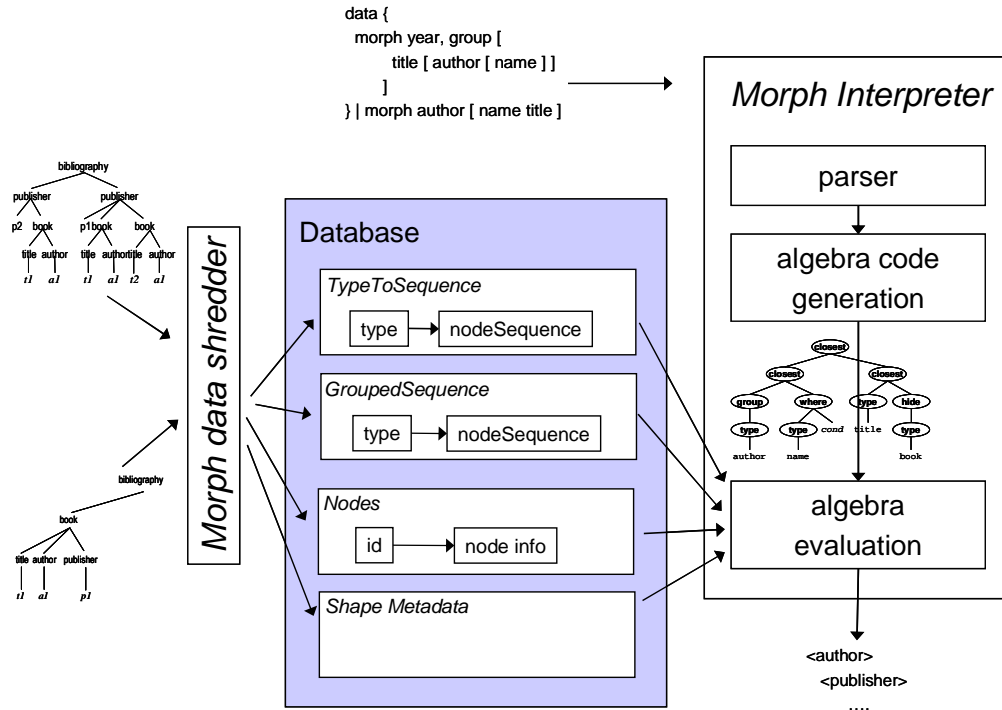


Figure 23 Morph implementation architecture

engine users, but XML search engine queries dispense with the shape of data entirely. Between these two extremes are shape polymorphic query languages. Queries in such languages avoid both the brittleness of XQuery and the loss of shape in XML search engine queries. In this paper we present Morph, a shape polymorphic query language for XML. We presented a syntax for Morph, sketched a translation to an algebra, and outlined an prototype implementation.

Much remains to be done. Our immediate goal is to complete the implementation and test the prototype against restructuring queries in a native XML DBMS (e.g., MonetDB). After that we plan to apply Morph to integrate data for a virtual herbarium (the problem that motivated our interest in data transformation languages). Shape polymorphism yields a “translative” semantics for determining the “sameness” of data, whereby differently shaped data can be translated to a common representation for comparison and integration. Ultimately we plan to investigate the expressiveness of Morph. Morph is certainly less expressive than XQuery; our goal is to make data transformations easier. However, there may be simple, efficient extensions of Morph that can increase its power.

8. REFERENCES

- [1] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. CACM 13(6): 377-387 (1970).
- [2] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv, “XSearch: A Semantic Search Engine for XML,” in Proceedings of VLDB, Berlin, Germany, 2003, pp. 45-56.
- [3] C. B. Jay and J. R. B. Crockett, “Shapely types and shape polymorphism,” in Programming Languages and Systems - ESOP '94: 5th European Symposium on Programming, Springer-Verlag, 1994, pp. 302-316.
- [4] S. Krishnamurthi, K. Gray, and P. Graunke, “Transformation-by-example for XML,” in Proceedings of the 2nd International Workshop of Practical Aspects of Declarative Languages, LNCS 1753, 2000, pp. 249-262.
- [5] E. Leonardi, S. Bhowmick, Z. Ng, and C. Dyreson, Towards Evaluation of a Symmetric XPath Axis in a Tree-Unaware RDBMS. Technical Report, Nanyang Technological University, Singapore, 2008. [http://www.cais-ntu.edu.sg/~assourav/TechReports/Closest-TR.pdf](http://www.cais.ntu.edu.sg/~assourav/TechReports/Closest-TR.pdf). Submitted to WWW '09.
- [6] Y. Li, C. Yu, and H. V. Jagadish. “Schema-Free XQuery,” Proceedings of VLDB Conference, Sep. 2004, Toronto, Canada, pp. 72-83.
- [7] I. Manolescu, D. Florescu, and D. Kossmann, “Answering XML Queries over Heterogeneous Data Sources,” Proceedings of VLDB Conference, 2001, pp. 241-250.
- [8] T. Pankowski. “A High-Level Language for Specifying XML Data Transformations,” in Proceedings of ADBIS, Lecture Notes in Computer Science 3255, 2004.
- [9] F. Ruehr. “Structural Polymorphism,” Informal Proceedings, Workshop on Generic Programming, WGP'98, 1998, Marstrand, Germany.
- [10] S. Zhang and C. Dyreson, “Symmetrically Exploiting XML,” in Proceedings of the World Wide Web Conference, 2006 Edinburgh, Scotland, May 2006, pp. 103-111.
- [11] S. Zhang and C. Dyreson, “Polymorphic XML Restructuring,” in Proceedings of IIWeb, a WWW Workshop, May 2006. iiweb2006/cs.uiuc.edu/6.pdf