

Reference Number: 609

Title: Merge as a Lattice-Join of XML Documents

Authors: Kristin Tufte, David Maier

Contact Author:

Kristin Tufte

OGI School of Science and Engineering at OHSU

20000 NW Walker Road

Beaverton, OR 97006

tufte@cse.ogi.edu

(503) 675-6436

Topic Area: Core Database Technology

Category: Research

Relevant Topics:

Semi-structured Data, XML

Databases and database services in new context - Internet and the WWW

# Merge as a Lattice-Join of XML Documents

Kristin Tufte

OGI School of Science and Engineering  
at OHSU  
20000 NW Walker Road  
Beaverton, OR 97006  
USA  
tufte@cse.ogi.edu

David Maier

OGI School of Science and Engineering  
at OHSU  
20000 NW Walker Road  
Beaverton, OR 97006  
USA  
maier@cse.ogi.edu

## Abstract

We explore theoretical foundations of the Merge operation. Merge functions as a kind of “recursive union” over similarly structured XML documents to produce a new XML document, and can be used for creating aggregates over streams of XML fragments. We describe the Merge operation and show that Merge is in fact the join operation of an upper semi-lattice of conformant documents. Our work relies on a representation of unordered XML documents as sets of attributed paths. We have found this representation useful for comparing the information content of documents, and determining when a given document satisfies certain key-like constraints.

## 1. Introduction

Data in XML format is permeating all corners of the computing milieu: e-commerce, data streaming, web content, messaging, data interchange, database management. Many of these areas require breaking up XML into fragments and combining pieces of XML into larger documents, for purposes such as querying, stream monitoring, incremental updates and producing “recombinant documents”. In the relational model, the flat structure makes such processes straightforward. They can generally be reduced to operations on individual rows: adding, deleting, splitting and combining. The hierarchical structure of XML, on the other hand, makes

matters more complicated. The points of manipulation may be buried deep within a document, and document elements share surrounding context with other elements, rather than being independent rows. Various path languages and syntaxes can be used to decompose XML into constituent parts or extract fragments. The work reported here focuses on the composition side. We define a binary operation Merge over XML elements that can be used as the basic building block for various processes involving combining XML fragments and documents. Merge can be used as the basis of a join-like operation over sets of documents, to append new information to existing documents, to accumulate a stream of fragments into a single document and to combine partial results for incremental query evaluation.

As one possible use for Merge, consider an on-line auction site that maintains information about items for auction and which receives a steady stream of bids on those items from its customers. Merge can be used to maintain an Auction Status Document containing descriptions of the items for auction and a list of bids for each item. Maintaining such a document is done by creating an initial document that contains auction item descriptions and one by one merging bids into this document. In doing so, we create an Auction Status Document that is continually updated and that can be queried by users at any time. We call such a document an “Accumulator.” Figure 1 shows a graphical representation of the initial Auction Status Document, Figure 2 shows a new bid, and Figure 3 shows the updated Auction Status Document. An important feature is that we can use Merge and the Accumulator together to maintain aggregates over the stream of bids. For example, we could keep track of the number of bids on an item, or the highest and lowest bids, further, we could even track the highest bid for each bidder. Finally, the same instance of Merge could be used to add new auction items into the Auction Status Accumulator.

Shanmugasundaram, et al. [15] proposed an architecture for processing data streams over nested

---

*Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment*

**Proceedings of the 28<sup>th</sup> VLDB Conference,  
Hong Kong, China, 2002**

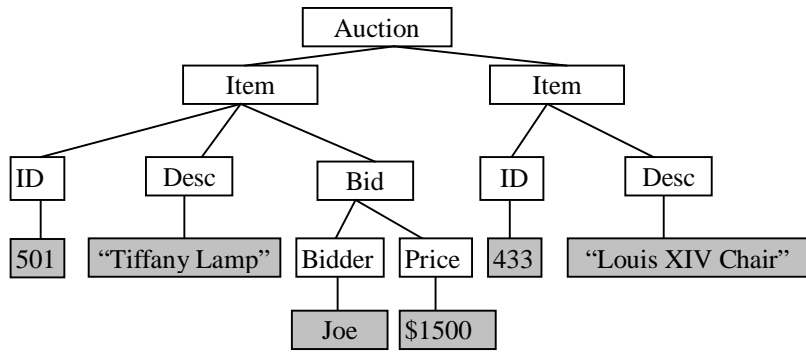


Figure 1 – Auction Status Document

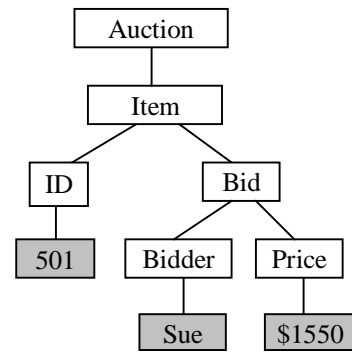


Figure 2 – New Bid

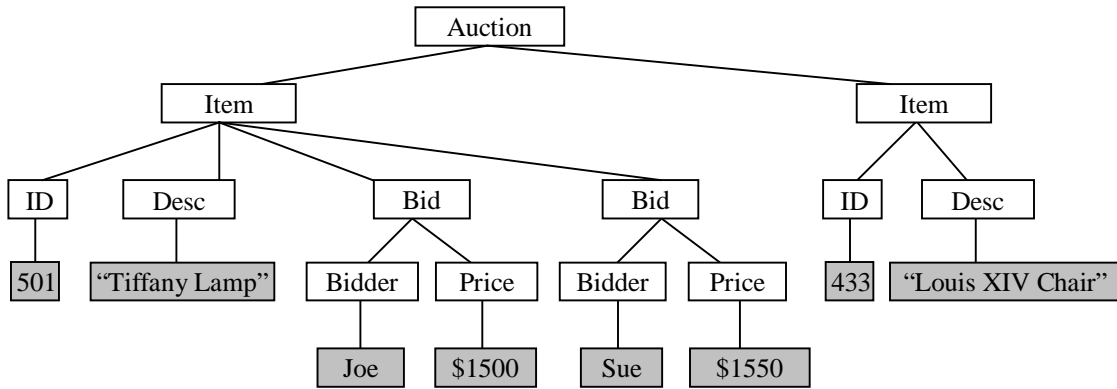


Figure 3 – Merged Document

structures. That paper observed that having the ability to pass information between operators at a finer granularity than a tuple would improve the performance of certain types of queries. Merge can help solve this problem by providing a mechanism for merging the fine-grain information back into the tuple attributes.

Merge is a powerful operator that takes two XML documents and pieces them together to create a result document. The process is controlled by a flexible mechanism called a Merge Template, which specifies how to recursively combine two XML documents. We have implemented a merge-based Accumulate operator in Version 1.0 of the Niagara Internet Query System [12].

The rest of this paper is organized as follows: Section 2 discusses related work, Section 3 describes applications of the Merge Operator, Section 4 presents the Merge Operation and the Merge Template, Sections 5 and 6 contain the heart of the paper – the formal foundation for the Merge operation in Section 5 and the Merge-Lattice Theorem in Section 6, finally, Section 7 concludes.

## 2. Related Work

The Deep Union operator defined by Buneman, et al.[4] is similar to the simplified version of merge used in our theoretical work. Buneman, et al. also describe a path set

notation for representing semi-structured data and a method for describing a semi-structured value as the Deep Union of a set of paths. In the Buneman work, it is assumed that nodes are labelled with their key values and that nodes have unique labels. Merge does not have this requirement. Multiple Merge Templates can be used with the same data set without changing the data itself. This flexibility allows the same data to be categorized differently, without any modifications to the data itself. For example, sellers might wish to have auction bids aggregated by item to enable them to see lists of bids on each item. On the other hand, the security department might wish to aggregate bids by bidder to create a list of items each bidder has bid on in order to detect suspicious activity. We could use the same data set with different Merge Templates to do both aggregations. To do these two aggregations with Deep Union would require two distinct data sets with different labels. In addition, our work differs from the Deep Union work in that we use XML directly, while they use a more restrictive semi-structured model. A related paper by Buneman, et al. [6] uses a modified version of Deep Union and timestamps to archive scientific data.

Liefke and Davidson [9] propose a mechanism for specifying keys for XML documents and mapping those documents into their semi-structured data model. They

also show a one-to-one relationship between path sets and semi-structured values, but do not provide a relationship to a semi-lattice, as we do. In fact, Deep Union and the Deep Update operator proposed by Liefke and Davidson resemble the Union and Join operators of the Verso Algebra [1], which uses a data model based on nested relations. Buneman, et al. propose a definition of keys for XML [5] and proceed to reason about the keys.

The Accumulator described in the introduction is very similar to a materialized view over data streams. There has been a lot of work on views in relational database systems [7]. Research has been done on work on view updates for semi-structured data [2][18]; however, much of that work assumes the existence of OIDs, which are unlikely to be available for XML documents. Tatarinov et al. Proposed operations and language extensions for updating XML data structures [16]; however, the implementation of these operations is over XML stored in a relational database. Babu and Widom [3] propose an architecture for stream processing and use that architecture to classify previous work related to stream processing. This work focuses on relational as opposed to semi-structured data. Finally, several systems have been developed for processing XML or streaming data including Niagara [12], Lore [10] and Tukwila [11][12]. The Merge operation is not available in any of these systems. The Yat system [6] uses XML for data integration. Merge differs from integration in that we aggregate only; we do not address mediation or query reformulation.

### 3. Applications

The Merge operator can be used in two primary ways: as an “accumulator” and as an operator in a query plan. In its function as an accumulator, Merge maintains a continuously updated XML document over a stream of XML data as was described in the introduction. In that example, Merge maintains the Auction Status Document by continually merging new bids, new item information, and so on into the Auction Status Document. The Auction Status Document is called an *accumulator*. An important feature of an accumulator is that it is available to be queried like any other XML document. The Accumulator provides a view over the incoming data stream.

Alternatively, Merge can be used as an operator in a native XML query processing system. In fact, a Merge operator has been implemented in Niagara Version 1.0 [14]. In this system, Merge can be used to replace a series of other operators such as joins and aggregations to provide alternate query processing options. We believe that in certain cases the Merge operator can provide significantly superior performance to the standard set of Niagara operators and we are beginning to test this theory. Niagara [13], like many native XML processing systems such as Yat [6] and Tukwila [12], must unnest data before processing it and then re-nest and construct the XML

result. These systems typically process data by first unnesting it using a scan-like operator for selecting subtrees of an XML document corresponding to a path expression. Then the data is processed using familiar relational operators such as join, select and aggregates. Finally, an operator is used to piece the XML (back) together to construct the result. The reconstruction of the result can be expensive, in particular if the query only touches a small portion of the document deep in the tree. By using the Merge operator in query processing, we believe we can avoid the unnesting and reconstruction costs and improve performance.

For example, consider the Auction example. Assume auction site users are allowed to report suspicious bidders – for example, bidders that are placing bids solely to increase the price of an item. On a regular basis, the security department wants to merge a list of suspicious bidders and the Auction Status Document to flag suspicious bidders and then run a query over the merged document to detect inappropriate activity. To do this kind of processing in a an XML system without an operator like Merge would require that the Auction Status Document be unnested all the way down to the Bidder level, then the Bidders joined with the Suspicious Bidders list, and then the document re-nested (3 levels) to create an Auction Status Document with flagged suspicious bidders. Since the number of suspicious bidders is likely to be a small percentage of the total number of bidders, the un-nest and nest will be very expensive compared to the join and will consume a significant proportion of the processing time. Using Merge, we could retrieve the Auction Status Document, merge in the suspicious bidders and make the resulting document available to the security department without the overhead of the un-nest and nest operations. Note that we do not want to update the actual Auction Status Document since it is visible to the world, rather we temporarily want to merge the suspicious bidders list in and run some queries.

Incrementality is important in considering what type of processing operations to use. If one has a batch of bid data to be added all at once to the Auction Status Document, it might make sense to flatten the Auction Status Document, join with the bid data, and re-nest to produce a new Auction Status Document. However, if the bid data arrives incrementally and there is a need to view the state of the Auction Status Document after each bid has been entered, then there has to be many repetitions of the flatten, join, nest cycle and flatten and nest will dominate the cost. In this case, an operator like Merge may significantly improve performance.

A disadvantage of storing XML in a RDBMS is the conversion and materialization of XML into tuples in the RDBMS. If XML data is to be queried over and over, this may be an acceptable operation. However, if the data to be processed is available as a data stream to be processed on the fly, the conversion cost may be prohibitive. The number of static XML documents on the World Wide

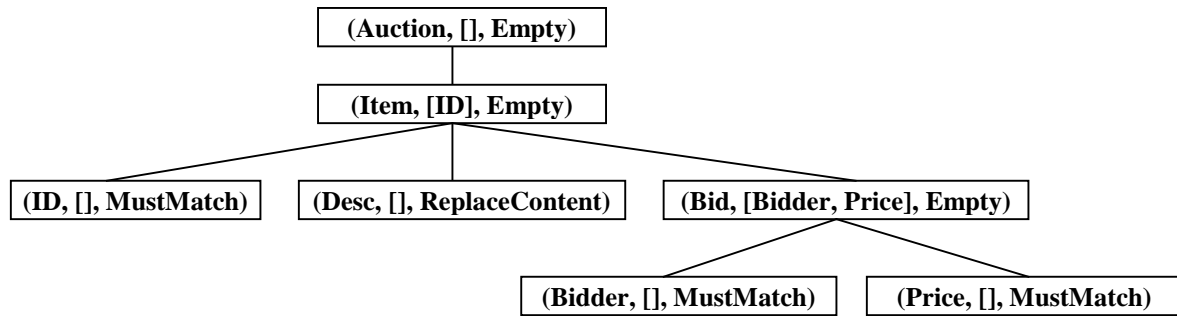


Figure 4 – Logical Representation of a Merge Template

Web is limited; we believe that much of the travelling XML will be in stream format. For example, a stock ticker, baseball play-by-play results and sensor data could be published as XML streams. These streams may be stored for later mining, but a significant part of the processing of these data streams will be done on the fly and materializing streaming data into relational tables may be not optimal. Merge provides an operator which can work on native XML data and avoid the materialization costs. Perhaps more importantly, a significant cost of relational storage of XML is the cost of recreating the XML for the results – this often involves many joins and can be very costly. Merge can process XML data without shredding it into pieces, and avoid the reconstruction costs.

## 4. Merge Operation Description

This section describes the Merge operation. A specification that we have developed, called a Merge Template, controls the Merge operation.

The Merge operator works by stepping through the two documents to be merged and the Merge Template in parallel in a depth-first fashion, merging the two documents element by element. Consider the example in Figure 1 showing a graphical depiction of an XML document representing descriptions of items on auction and bids on those items. Figure 2 shows an XML document for a new bid on the Tiffany Lamp. Figure 3 shows the result when the New Bid is merged with the Auction Status Document. To process this merge, the operator first combines the root Auction elements, next the Item elements are joined on ID and Item 501 from the Auction Description is merged with Item 501 from the New Bid. Next the Bid elements are determined to be distinct because of different bidders, so both bid elements are maintained in the result.

### 4.1 Document Definition

For the purpose of this paper, an XML document is modelled as a tree with two types of nodes: element nodes which contain a name and data nodes which contain a

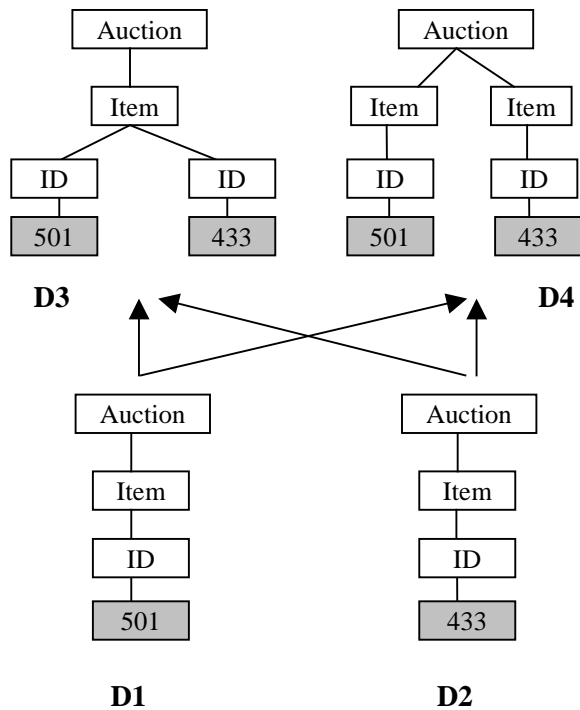
string data value. Data nodes are shaded grey in graphical representations of XML documents. In practice, Merge can handle attributes; however, we leave them out of this paper for ease of presentation. For the purposes of this paper, children nodes are unordered. Handling ordered elements is left to future work.

Before we proceed, we remark briefly on equality. It is important to distinguish between element identity and value equality in XML documents. For the purposes of this paper, we assume that equality (=) applied to an element name or data value refers to (string) value equality, and equality (=) applied to nodes refers to node identity.

### 4.2 Merge Template

The Merge process is guided by a Merge Template, which describes how to combine two documents. A Merge Template is a tree of *Element Merge Templates* (EMTs). An EMT is a triplet containing a *Name*, a *Local Key* and a *Content Combine Function*. Name is a string and sibling EMTs must have distinct names. Local Key is a possibly empty list of path expressions as further described in Section 4.3. The Content Combine Function is a value from an enumeration that specifies a function for combining the *local* content (character data) of two elements. Figure 4 shows a graphical representation of the Merge Template for the Auction example.

Each EMT describes how to produce a (possibly empty) set of elements for the result. For example, the Item EMT specifies how to create Item elements in the result. The Local Key specifies how to match an element of the type associated with the EMT to another element of that type; the Item EMT in Figure 4 indicates that two Items represent the same entity and will be merged if they have the same ID. The Bid EMT indicates that bids are to be matched on Bidder and Price, so all bids are retained for a bidder (except duplicate bids). The Content Combine Function tells how to combine the *local* content of two elements. Several content combine functions are supported including content replace, element replace and various aggregates. The “Empty” Content Combine Function indicates that the element should not have any



**Figure 5 - Lattice Violation: D3 is correct result**

local data. (That is, any additional information should be in subelements.) The children EMTs indicate how to process an element's children. A key feature of Merge is its flexibility. In addition to being used to merge a new bid into the Auction Status Document, this same Merge Template can also be used to merge an item description change into the Auction Status Document, or add a new auction item.

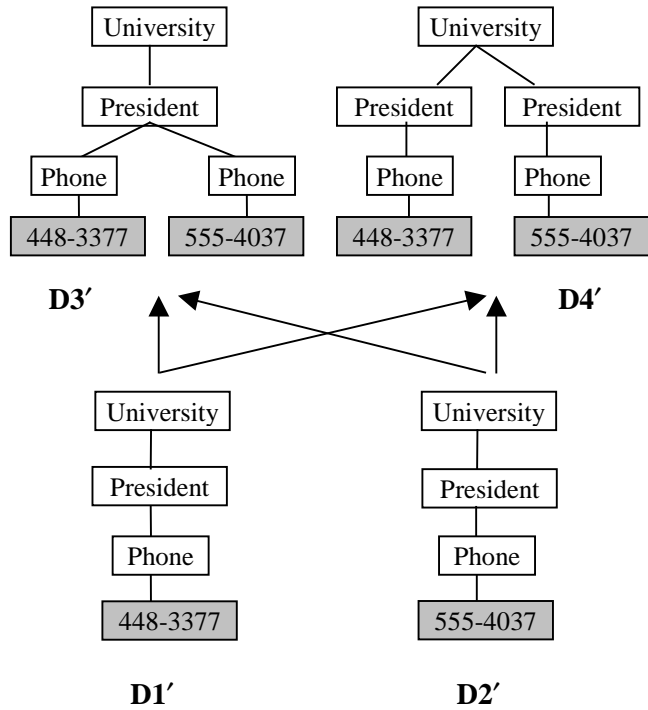
Merge Templates have a representation as XML documents. Each EMT is represented by an EMT element with a Name attribute and Local Key and Content Combine Function sub-elements. The XML document corresponding to the Merge Template in Figure 4 is included as Appendix A. A previous paper [17] describes the Merge operation in more detail.

### 4.3 Description of Local Key

Local Keys are used to determine when two elements represent the same entity. A Local Key is a list of simple paths of the form *name1.name2.name3*. In general, we write a local key as  $[p_1, p_2, \dots, p_n]$  where each  $p_i$  is a simple path. Two elements are deemed equivalent when they have equal values for *all* paths in the local key.

## 5. Theoretical Foundations of Merge

This section provides a formal foundation for the Merge Operation. Studying Merge as a lattice-theoretic operator is important for several reasons. First, we wish to use the lattice-theoretic definition as a basis for our definition of



**Figure 6 – Lattice Violation: D4' is correct result**

the Merge operator, as opposed to our current operational definition. Such a formal definition should aid in proving equalities that can be used for query optimization. Second, understanding the lattice-theoretic properties of Merge can help us understand the conditions under which XML documents can be decomposed and reconstituted without loss of information. We make two contributions. First, we show that the set of XML documents, when suitably constrained, forms an upper semi-lattice of which Merge is the join operation. Second, we present a representation for unordered XML documents as “path sets.” The path set representation is an effective tool for reasoning about XML documents, as will be seen below.

Our theoretical development is restricted to the basic form of the merge operator, which performs structural aggregation. Thus, in the theoretical development, the Content Combine Function is not used. That is, the local data of elements is never combined or changed, but elements can receive new children – as in the Auction example, where the new bid is “inserted” into the Auction Status Document. Integrating aggregates such as max, min and count, which have results with a defined order, into our framework appears feasible. In fact, the implementation of Merge supports multiple Content Combine Functions including several aggregates. We continue to investigate whether other combining forms can be cast into the lattice framework as well. We begin with discussion and motivation.

## 5.1 Discussion and Motivation

Consider Figure 5 which shows two documents, D1 and D2, and two possible results for the merge of D1 and D2, named D3 and D4. In the abstract, either D3 or D4 is a possible result of the merge of D1 and D2; however, document D3 does not make sense in this setting since an item can not have two IDs. Thus the correct result for the merge of D1 and D2 is in fact document D4. If we replaced the names and values in Figure 5 with a different set of names and values, the result could be different, as is shown in Figure 6. The correct result of merging documents D1' and D2' would seem to be D3', since a University should have only one President, but that President might have two phone numbers.

Figures 5 and 6 illustrate the key decision that must be made by the structural aggregation merge operator – whether or not to combine two elements. So, what information is required to make that decision? If we knew, for example, that ID was a key for Item, then we could eliminate D3 and know that D4 was the correct result for the merge of D1 and D2. Simply put, we need to know when two elements should be considered to represent the same entity. The Merge Template provides exactly that information in the form of Local Keys. A Merge Template could tell us that an Item must have a unique ID or that a University can have only one President.

But, what does this all have to do with lattices? For completeness, we recall that a *lattice* is a partially ordered set,  $S$ , such that for any two elements of  $S$ ,  $s_1$  and  $s_2$ , the Greatest Lower Bound (GLB) and Least Upper Bound (LUB) of  $s_1$  and  $s_2$  exist, are unique and are elements of  $S$ . An upper semi-lattice requires only that LUBs exist. Hereafter, we use lattice loosely – when we say lattice it is implied to mean upper semi-lattice.

In Figures 5 and 6, one can see the beginnings of how a lattice of XML documents will be formed. One can interpret the arrows in the figures as corresponding to an intuitive document containment ordering in which document D is contained in document D' if D is a subtree of D' and D and D' have the same root. Under this ordering, D1 and D2 are contained in both of D3 and D4. From a lattice perspective, therein lies the problem. Under this containment ordering D3 and D4 are minimal upper bounds for D1 and D2, but are not comparable, thus no unique LUB exists. To create a unique LUB and take a step towards creating a lattice, we must effectively eliminate D4 from the ordering. We use the Local Keys in a Merge Template for this purpose. Enforcing a set of keys can eliminate documents that cause ambiguous least upper bounds.

## 5.2 Proof Structure

In the rest of this section, we focus on developing the prerequisites needed to prove the Merge-Lattice Theorem. This Merge-Lattice Theorem states that the set of XML

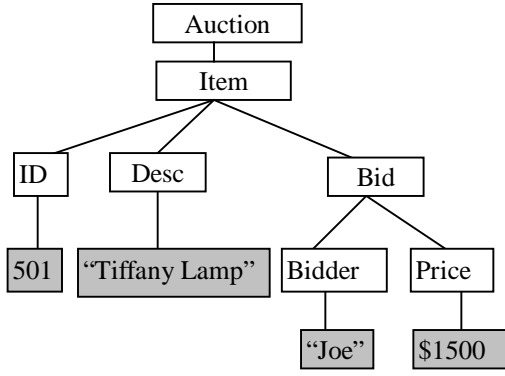
documents with a specific ordering based on a Merge Template is a lattice. The LUB of two documents in this lattice is the Merge of those two documents. In the proof of the Merge-Lattice Theorem, we will represent XML documents as sets of paths (“path sets”) and use the fact that a set of sets under the set containment ordering forms a lattice to show that the set of XML documents under the given ordering is indeed a lattice.

To set up for the Merge-Lattice Theorem proof, we must describe how to represent XML documents as path sets. Path sets are constructed using key values, which are created based on a “compatibility mapping” between a document and a Merge Template. In this section, we first explore the compatibility mapping between a document and a Merge Template, and then proceed to show how the compatibility mapping and Merge Template can be used to create sets of key values for elements and a path set for a document. After we show how to create path sets, we formally define a document ordering and proceed to prove some precursor theorems. The Merge-Lattice Theorem is proved in Section 6.

## 5.3 Compatibility Mapping

A *compatibility mapping* maps elements in a document into a Merge Template. Consider the Auction document shown in Figure 3 and the Merge Template shown in Figure 4 and notice how the structure of elements in the Auction document parallels the structure of the EMTs in the Merge Template. It should be clear that we can create a mapping from the Auction document to the Merge Template in which each element in the Auction document is mapped to an EMT which preserves name and parent-child relationships. We call such a mapping a compatibility mapping and we say that a document is “compatible” with a Merge Template if such a mapping exists.

Formally, an XML document,  $D$ , is *compatible* with a Merge Template  $T$ , if there exists a homomorphism,  $\phi$ , that maps elements in  $D$  to EMTs in  $T$ , such that  $\phi(D.root) = T.root$ , and for every element,  $E$ , in  $D$ ,  $name(E) = name(\phi(E))$ , and  $\phi(parent(E)) = parent(\phi(E))$ . The functions *name* and *parent* are defined in the obvious way. In addition,  $D.root$  refers to the unique root element, or document element, of an XML document and  $T.root$  refers to the unique root EMT of a Merge Template. These conventions are used throughout this paper. Compatibility mappings between a given document and a given Merge Template are unique if they exist. This can be proven inductively (starting at the root of the document and Merge Template) by using the fact that siblings in a Merge Template have unique names and the definition of a compatibility mapping. We proceed to show how compatibility mappings are used to create key values for a document.



```

Auction(ε):ε
Auction(ε).Item(ID:501):ε
Auction(ε).Item(ID:501).ID(ε):501
Auction(ε).Item(ID:501).Desc(ε):TiffanyLamp
Auction(ε).Item(ID:501).Bid(Bidder:Joe,Price:$1500):ε
Auction(ε).Item(ID:501).Bid(Bidder:Joe,Price:$1500).
    Bidder(ε):Joe
Auction(ε).Item(ID:501).Bid(Bidder:Joe,Price:$1500).
    Price(ε):$1500
  
```

Figure 7 – An XML Document and its Associated Path Set

#### 5.4 Key Values

This section provides the basis for path-set creation, by describing how to create local key values and rooted key values for elements in a document. The following section explains how to use these key values to create path sets.

A compatibility mapping,  $\phi$ , associates each element in a document with an EMT in a Merge Template. Recall, the associated EMT contains a Local Key consisting of a possibly empty list of paths. Informally, the Local Key Value for an element,  $E$ , is constructed by evaluating each path in the Local Key over the subtree rooted at  $E$  and creating a list of the results. Let  $patheval(E, D, p)$  be a function which returns the set of values found when the path  $p$  is evaluated over the subtree rooted at  $E$  in document  $D$ . We define an element,  $E$ , to be *key-exact* with respect to an EMT  $\phi(E)$  if for every path,  $p$ , in the local key in  $\phi(E)$ ,  $patheval(E, D, p)$  is a singleton set. In other words, for every path in the local key, that path appears exactly once in the subtree rooted at  $E$ , or if the path appears multiple times, all instances of the path have the same value. In our Auction example, an Item element is key-exact with respect to the Item EMT if all of the Item’s ID sub-elements have the same value. An item with multiple ID children does not make sense in our example; however, the alternative is to define key-exact to allow the local key paths to appear exactly once in the subtree, which is more restrictive than necessary.

Formally, let  $D$  be a document which is compatible with a Merge Template,  $T$ , with compatibility mapping  $\phi$  and let  $E$  be a key-exact element in  $D$  with respect to  $T$  and  $\phi$ . Let  $\phi(E).LocalKey = [p_1, \dots, p_n]$  where each  $p_i$  is a simple path. Then, the local key value of  $E$  is defined as follows:  $lkv(E, D, T, \phi) = patheval(E, D, p_1) \dots patheval(E, D, p_n)$ . If the Local Key is the empty list, then the local key value is the empty string,  $\epsilon$ . Strictly speaking a local key value is a list of sets (or the empty string). Since local key values are defined only on key-exact elements, the local key value is a list of singleton sets and therefore when we write local key values, we generally omit the set notation.

Local key values are concatenated to create *rooted key values*. In fact, a rooted key value is just a list of name-labelled local key values of an element and its ancestors all of which must be key-exact with respect to a Merge Template and compatibility mapping. The rooted key value for an element,  $E$ , in document,  $D$ , with respect to Merge Template,  $T$ , and compatibility mapping,  $\phi$ , or  $rkv(E, D, T, \phi) = rkv(parent(E), D, T, \phi).name(E)(lkv(E, D, T, \phi))$ . If  $parent(E)$  is null,  $rkv(E, D, T, \phi) = name(E)(lkv(E, D, T, \phi))$ .

#### 5.5 Representing XML Documents as Prefix Path Sets

An important contribution of this paper is a representation for XML documents as path sets. In this section, we define our notion of a prefix path set; in the following sections, we elaborate on the relationships between documents and their path sets.

We define a function,  $\rho$ , which maps an element to a prefix path as follows:  $\rho(E, D, T, \phi) = rkv(E, D, T, \phi).value(E)$ . The function  $value(E)$  returns an element’s data or the empty string,  $\epsilon$ , if the element has no data. Note that  $value(E)$  returns only data local to the element, and does not return recursive content. Thus, the prefix path for an element is the element’s rooted key value appended with the element’s value. A document’s path set is the union of the prefix paths for all elements in  $D$ . With a small abuse of notation, we extend the function  $\rho$  to map documents to their path sets, thus  $\rho(D, T, \phi) = \{\rho(E, D, T, \phi) \mid E \in D\}$ . Figure 7 shows an XML document and its associated path set.

Based on the idea that a key should determine value, we define the concept of *key-respecting*. A document,  $D$ , which is key-exact with respect to a Merge Template  $T$  and compatibility mapping  $\phi$  is *key-respecting* with respect to  $T$  if no two elements of  $D$  have the same rooted key value. We extend the concept of key-respecting to path sets, and say that a path set,  $P$ , is *key-respecting* if there do not exist  $p_1$  and  $p_2$  in  $P$  such that  $p_1$  and  $p_2$  differ only in the terminal element value string. It should be clear that if a document is key-respecting, its path set is key-respecting.

In future arguments, we will often want to use functions that are a restriction of  $\rho$  to a particular document. We will use names such as  $\rho_1$  for these functions. It should be clear that if  $D_1$  is key-respecting with respect to a Merge Template, then the restriction of  $\rho$  to  $D_1$  is a bijection between elements in  $D_1$  and paths in  $D_1$ 's path set since paths contain rooted key values. Note that we have defined key-respecting without formally defining rooted keys. We can define a notion of global or rooted keys derived from a Merge Template, but we do not include the material because the results herein can be obtained without it.

## 5.6 Document Containment Ordering

In this section, we discuss document ordering. Our goal is to prove that the set of XML documents is an upper semi-lattice. To prove that this set is an upper-semi lattice, we will define a partial order on the set and then show that under this order, there is a unique least upper bound for every pair of conformant documents.

We have several goals for this ordering. First, the order should mirror the merge of two documents. That is the least upper bound of two documents in this ordering will be shown to be the Merge of those two documents. Second, we wish the ordering to be intuitive. Finally, we must be able to show that this ordering is the same as ordering documents by set-containment on their path sets. This equivalence will allow us to superimpose the lattice structure of set-containment onto the document containment ordering and show that XML documents,

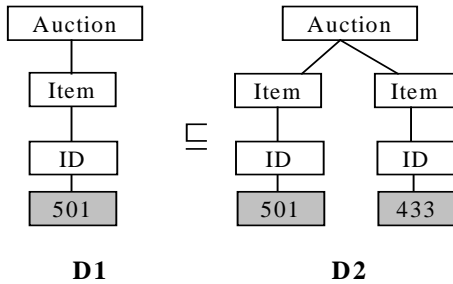


Figure 8 – Document Containment

under our ordering, are a lattice.

We begin by formally defining the document containment order introduced in Section 5.1. Consider Figure 8. It seems clear that document  $D_1$  is contained in document  $D_2$ . More specifically, we say document  $D_1$  is contained in document  $D_2$ , if  $D_1$  is a subtree of  $D_2$  and  $D_1$  and  $D_2$  have the same roots. Formally, let  $D_1$  and  $D_2$  be two documents.  $D_1$  is *contained in*  $D_2$ , or  $D_1 \sqsubseteq D_2$ , if there exists a 1-1 homomorphism  $\lambda$  that maps  $D_1$  into  $D_2$  such that for every element  $E$  in  $D_1$ ,  $name(E) = name(\lambda(E))$ ,  $value(E) = value(\lambda(E))$  and  $\lambda(parent(E)) = parent(\lambda(E))$  and  $\lambda(D_1.root) = D_2.root$ . Note we must require  $\lambda$  to

be 1-1. If not, then documents  $D_3$  and  $D_4$  in Figure 5 would be deemed equivalent, which we want to avoid.

The containment ordering is intuitive and lends itself to creating a semi-lattice of key-respecting documents; however, it can be difficult to work with. Another way to order documents would be to say  $D_1$  is less than  $D_2$  if  $\rho(D_1) \subseteq \rho(D_2)$ . It turns out that this path-set based ordering and the document containment orderings are identical on key-respecting documents. Thus, we have the following theorem and corollary.

**Theorem 1:** *Let  $D_1$  and  $D_2$  be two documents that are key-respecting with respect to a Merge Template  $T$ , with compatibility mappings  $\phi_1$  and  $\phi_2$ , respectively.  $D_1 \sqsubseteq D_2$  if and only if  $\rho(D_1, T, \phi_1) \subseteq \rho(D_2, T, \phi_2)$ .*

**Corollary:** *If  $D_1$  and  $D_2$  are compatible and key-respecting with respect to a Merge Template, then  $D_1$  and  $D_2$  are equal up to ordering of elements if and only if  $\rho(D_1, T, \phi_1) = \rho(D_2, T, \phi_2)$*

*Proof of Theorem 1:*

In the interest of space, we provide a sketch of the proof of the first half of this theorem. The proof of the second half is included.

Let  $D_1, D_2$  be two documents compatible with a Merge Template,  $T$ , with compatibility mappings  $\phi_1$  and  $\phi_2$ . Let  $\rho_1$  and  $\rho_2$  be the restrictions of  $\rho$  to  $D_1$  and  $D_2$ , respectively. Recall that since  $D_1$  and  $D_2$  are key-respecting,  $\rho_1$  and  $\rho_2$  are bijections. In order to improve readability, throughout the proof we write  $\rho(D_1)$  for  $\rho(D_1, T, \phi_1)$  and similarly for  $\rho(D_2)$ .

$\Rightarrow$  **Show:** If  $D_1 \sqsubseteq D_2$  then  $\rho(D_1) \subseteq \rho(D_2)$ . (Sketch)

Assume  $D_1 \sqsubseteq D_2$  and let  $\lambda$  be the containment homomorphism from  $D_1$  to  $D_2$ .

To show  $\rho(D_1) \subseteq \rho(D_2)$ , we let  $p_1$  be an arbitrary path in  $\rho(D_1)$  and show that  $p_1 \in \rho(D_2)$ . To do so, let  $E_1$  be the element in  $D_1$  such that  $\rho(E_1, D_1, T, \phi_1) = p_1$ , and let  $p_2 = \rho(\lambda(E_1), D_2, T, \phi_2)$ . Below, we sketch how to show that  $p_1 = p_2$  and therefore  $p_1 \in \rho(D_2)$ .

The value of  $p_1$  and  $p_2$  can be expressed as follows:

$$\begin{aligned} p_1 &= \rho(E_1, D_1, T, \phi_1) \\ &= rkv(parent(E_1), D_1, T, \phi_1).name(E_1) \\ &\quad (lkv(E_1, D_1, T, \phi_1):value(E_1)) \end{aligned}$$

$$\begin{aligned} p_2 &= \rho(\lambda(E_1), D_2, T, \phi_2) \\ &= rkv(parent(\lambda(E_1)), D_2, T, \phi_2).name(\lambda(E_1)) \\ &\quad (lkv(\lambda(E_1), D_2, T, \phi_2):value(\lambda(E_1))) \end{aligned}$$

Since  $\lambda$  is a containment homomorphism, we know  $name(E_1) = name(\lambda(E_1))$  and  $value(E_1) = value(\lambda(E_1))$ . It remains to show that  $rkv(parent(E_1), D_1, T, \phi_1) = rkv(parent(\lambda(E_1)), D_2, T, \phi_2)$  and  $lkv(E_1, D_1, T, \phi_1) = lkv(\lambda(E_1), D_2, T, \phi_2)$ . To do this, we first show that if  $E_1$  is an element in  $D_1$   $\phi_1(E_1) = \phi_2(\lambda(E_1))$  – this can be proved by structural induction. We use this result to show  $lkv(E_1, D_1, T, \phi_1) = lkv(\lambda(E_1), D_2, T, \phi_2)$ , and then use

induction to prove  $rkv(\text{parent}(E_1), D_1, T, \phi_1) = rkv(\text{parent}(\lambda(E_1)), D_2, T, \phi_2)$ . This completes the sketch.

$\Leftarrow$  **Show:** If  $\rho(D_1) \subseteq \rho(D_2)$  then  $D_1 \sqsubseteq D_2$ .

We assume  $\rho(D_1) \subseteq \rho(D_2)$ , and show there exists a containment homomorphism from  $D_1$  to  $D_2$ . Let  $\lambda$  be a mapping from  $D_1$  to  $D_2$  such that if  $E_1$  is an element in  $D_1$  then  $\lambda(E_1) = \rho_2^{-1}(\rho_1(E_1, D_1, T, \phi_1), T, \phi_2)$  and  $\lambda(\text{null}) = \text{null}$ . We show  $\lambda$  is a containment homomorphism.

First observe that for an arbitrary  $E$  in  $D_1$ ,  $\rho_2(\lambda(E)) = \rho_2(\rho_2^{-1}(\rho_1(E))) = \rho_1(E)$ . This implies that  $\text{name}(\lambda(E)) = \text{name}(E)$ , and  $\text{value}(\lambda(E)) = \text{value}(E)$ , since an element's name and value are contained in its path.

It remains to be shown that  $\lambda(\text{parent}(E)) = \text{parent}(\lambda(E))$ . It will suffice to show that  $rkv(\lambda(\text{parent}(E)), D_2, T, \phi_2) = rkv(\text{parent}(\lambda(E)), D_2, T, \phi_2)$  since  $D_2$  is key-respecting. Since  $\rho_2(\lambda(E)) = \rho_1(E)$  for arbitrary  $E$ , we know  $\rho_2(\lambda(\text{parent}(E))) = \rho_1(\text{parent}(E))$  which implies that  $rkv(\lambda(\text{parent}(E)), D_2, T, \phi_2) = rkv(\text{parent}(E), D_1, T, \phi_1)$ . To finish, we must show  $rkv(\text{parent}(E), D_1, T, \phi_1) = rkv(\text{parent}(\lambda(E)), D_2, T, \phi_2)$ .

We have the following equalities:

$$\begin{aligned} rkv(\lambda(E), D_2, T, \phi_2):value(\lambda(E)) &= \rho_2(\lambda(E)) && \text{(defn } \rho) \\ \rho_2(\lambda(E)) &= \rho_1(E) && \text{(true for arbitrary } E) \\ \rho_1(E) &= rkv(E, D_1, T, \phi_1):value(E) && \text{(defn } \rho) \\ value(E) &= value(\lambda(E)) && \text{(from above)} \end{aligned}$$

Putting these equations together, we can deduce that  $rkv(\lambda(E), D_2, T, \phi_2) = rkv(E, D_1, T, \phi_1)$ . Applying the definition of  $rkv$  to both sides of this equality, we get:

$$\begin{aligned} rkv(\text{parent}(\lambda(E)), D_2, T, \phi_2):name(\lambda(E))(lkv(\lambda(E), D_2, T, \phi_2)) \\ = rkv(\text{parent}(E), D_1, T, \phi_1):name(E)(lkv(E, D_1, T, \phi_1)) \end{aligned}$$

The above equation can only be true if  $rkv(\text{parent}(\lambda(E)), D_2, T, \phi_2) = rkv(\text{parent}(E), D_1, T, \phi_1)$ , as desired. Thus  $\lambda$  is a containment homomorphism and we are done.

□

## 5.7 Merging via Path Sets

To prove the Merge-Lattice theorem in the next section, we must show that any two “mergeable” documents have a unique least upper bound. In this section, we constructively prove the existence of an upper bound. The construction process in many ways parallels the merge process and the upper bound document constructed is indeed the least upper bound, or the merge, of the two documents.

Before we tackle the existence of an upper bound, we need to define two concepts used to compare pairs of documents, key-consistent and mergeable, and we need a notion of path length. Key-consistency implies that two documents do not contain inconsistent information. Consider the Auction document in Figure 1 and a second document that is identical, except that the description

(Desc) for item 501 is “Brass Sextant.” These two documents are *not* key-consistent. Two documents that are compatible and key-respecting with respect to a Merge Template  $T$  are *key-consistent* with respect to  $T$  if the union of their path sets is key-respecting. Two documents,  $D_1$  and  $D_2$ , are *mergeable* if they are key-consistent and  $lkv(D_1.\text{root}) = lkv(D_2.\text{root})$ . Note that the equivalence of local key values implies that the values of the document elements are equal. Finally, we turn to path length. We define the *length* of a prefix path to be the number of steps in the path excluding the terminal element value; the path  $t_1(lkv_1):v_1$  has length one and  $t_1(lkv_1).t_2(lkv_2):v_2$  has length two.

**Theorem 2:** Let  $D_1$  and  $D_2$  be documents that are mergeable with respect to a Merge Template  $T$ . Let  $\phi_1$  and  $\phi_2$  be compatibility mappings from  $D_1$  and  $D_2$  to  $T$ , respectively. Then, there exists a document  $D_3$  compatible with  $T$ , with compatibility mapping  $\phi_3$ , such that,  $D_3$  is key-exact and key-respecting with respect to  $T$  and  $\phi_3$  and  $\rho(D_3, T, \phi_3) = \rho(D_1, T, \phi_1) \cup \rho(D_2, T, \phi_2)$ .

*Proof:* In order to improve readability, throughout the proof we write  $\rho(D_1)$  for  $\rho(D_1, T, \phi_1)$  and similarly for  $\rho(D_2)$  and  $\rho(D_3)$ .

In the first portion of this proof, we inductively construct a document,  $D_3$ . We proceed to prove that  $D_3$  is compatible with  $T$ , is key-exact and key-respecting and  $\rho(D_3, T, \phi_3) = \rho(D_1, T, \phi_1) \cup \rho(D_2, T, \phi_2)$ . As we create  $D_3$ , we will also create a mapping from  $D_3$  to the path set  $\rho(D_1) \cup \rho(D_2)$  called  $\sigma$ . We saw in Section 5.5 how to translate an element into a path. In the construction of  $D_3$ , we do this process in reverse to create one element in  $D_3$  for each path in  $\rho(D_1) \cup \rho(D_2)$ . For a path  $t_1(lk_1).t_2(lk_2)\dots t_n(lk_n):v_n$ , we convert that path into an element with name  $t_n$ , value  $v_n$  and parent with rooted key value  $t_1(lk_1)\dots t_{n-1}(lk_{n-1})$ . Note that strictly speaking we do not know that the parent's rooted key value exists, however this intuition is correct. Finally, to prove that  $D_3$  is compatible with  $T$ , we compose mappings between  $D_3$ ,  $\rho(D_1) \cup \rho(D_2)$ ,  $D_1$  and  $D_2$  and  $T$  to create a compatibility mapping for  $D_3$ . We proceed with the formal proof.

**Base Case:** Let  $t_1$  and  $v_1$  be the name and value of  $D_1.\text{root}$ . Note that these are also the name and value of  $D_2.\text{root}$ , since  $D_1$  and  $D_2$  are mergeable. Create a root element for  $D_3$  with name  $t_1$  and value  $v_1$ . Since  $D_1.\text{root}$  and  $D_2.\text{root}$  have the same name, value and local key value, there exists a unique path,  $p$ , of length 1 in  $\rho(D_1) \cup \rho(D_2)$ . Path  $p = t_1(lk_1):v_1$  where  $lk_1$  is the local key value of  $D_1.\text{root}$  and  $D_2.\text{root}$ . Set  $\sigma(t_1(lk_1):v_1)$  equal to the newly created document element for  $D_3$ .

**Inductive Step:** Assume that elements have been constructed for all paths of length  $n-1$  or less. Let  $p = t_1(lk_1).t_2(lk_2)\dots t_n(lk_n):v_n$  be a path of length  $n$  in  $\rho(D_1) \cup \rho(D_2)$ . There must exist an element  $E_1$  in  $D_1$  or  $D_2$  such that  $\rho(E_1, D_1, T, \phi_1) = p$  or  $\rho(E_1, D_2, T, \phi_2) = p$ . Without

loss of generality, assume  $E_1$  is an element in  $D_1$ . We know that  $\rho(\text{parent}(E_1), D_1, T, \phi_1) \in \rho(D_1)$ . Now  $\rho(\text{parent}(E_1), D_1, T, \phi_1) = \text{rkV}(\text{parent}(E_1), D_1, T, \phi_1):v = t_1(lk_1) \dots t_{n-1}(lk_{n-1}):v$  for some string value  $v$ . Therefore, the path  $t_1(lk_1) \dots t_{n-1}(lk_{n-1}):v \in \rho(D_1) \cup \rho(D_2)$  and has length  $n-1$ . Let  $E_3 = \sigma(t_1(lk_1) \dots t_{n-1}(lk_{n-1}):v)$ . Create a child of  $E_3$ , with name  $t_n$  and value  $v_n$ . Set  $\sigma(p)$  equal to the newly created element. We have shown how to construct an element for an arbitrary path of length  $n$  and we are done with construction of  $D_3$ . We note that  $\sigma$  is a bijection because we create exactly one element for each path in  $\rho(D_1) \cup \rho(D_2)$ .

Now that  $D_3$  has been constructed, we must show that  $D_3$  is compatible with  $T$  and key-exact and key-respecting with respect to  $T$ .

**Compatibility:** We create a mapping,  $\phi_3$ , from  $D_3$  to  $T$  and show it meets the necessary criteria. We first consider the mapping of the root of  $D_3$  and then consider the rest of the elements. We map the root of  $D_3$  to the root of the Merge Template, so  $\phi_3(D_3.\text{root}) = T.\text{root}$ . We know that  $\text{name}(D_3.\text{root}) = \text{name}(D_1.\text{root}) = \text{name}(T.\text{root}) = \text{name}(\phi_3(D_3.\text{root}))$  and  $\text{parent}(D_3.\text{root}) = \text{null} = \text{parent}(T.\text{root}) = \text{parent}(\phi_3(D_3.\text{root}))$  as desired.

To create  $\phi_3$  for the rest of the elements in  $D_3$ , we compose mappings from  $D_3$  to  $\rho(D_1) \cup \rho(D_2)$ , from  $\rho(D_1) \cup \rho(D_2)$  to  $D_1$  and  $D_2$ , and from  $D_1$  and  $D_2$  to  $T$ . More specifically, we use  $\sigma^{-1}$  to map  $E_3$  to a path in  $\rho(D_1) \cup \rho(D_2)$ , next we map that path to  $D_1$  or  $D_2$  using  $\rho_1^{-1}$  or  $\rho_2^{-1}$ , and finally map the element from  $D_1$  or  $D_2$  to  $T$  using  $\phi_1$  or  $\phi_2$ . This composition gives us a mapping from an element in  $D_3$  to an EMT in  $T$ . Let  $E_3$  be an element in  $D_3$  other than  $D_3.\text{root}$ . Let  $\phi_3(E_3) = \phi_1(\rho_1^{-1}(\sigma^{-1}(E_3)))$  if  $\sigma^{-1}(E_3)$  is in the domain of  $\rho_1^{-1}$ , else,  $\sigma^{-1}(E_3)$  is in the domain of  $\rho_2^{-1}$  and we let  $\phi_3(E_3) = \phi_2(\rho_2^{-1}(\sigma^{-1}(E_3)))$ . We must show  $\phi_3$  meets the requirements for a compatibility mapping. Without loss of generality, assume  $\phi_3(E_3) = \phi_1(\rho_1^{-1}(\sigma^{-1}(E_3)))$ . We have  $\text{name}(E_3) = \text{name}(\rho_1^{-1}(\sigma^{-1}(E_3))) = \text{name}(\phi_1(\rho_1^{-1}(\sigma^{-1}(E_3)))) = \text{name}(\phi_3(E_3))$ . The first equality is due to the method of construction of  $D_3$ , the last two equalities are algebraic manipulation.

It remains to be shown that  $\phi_3(\text{parent}(E_3)) = \text{parent}(\phi_3(E_3))$ . First, let  $\sigma^{-1}(E_3) = t_1(lk_1) \dots t_{n-1}(lk_{n-1}):v_n$ . From the method of construction of  $D_3$ , we know that  $\text{parent}(E_3) = \sigma(t_1(lk_1) \dots t_{n-1}(lk_{n-1}):v)$  for some value  $v$ . Since  $\sigma$  is a bijection, we can apply  $\sigma^{-1}$  to both sides of the previous equation to obtain  $\sigma^{-1}(\text{parent}(E_3)) = t_1(lk_1) \dots t_{n-1}(lk_{n-1}):v$ . Since  $t_1(lk_1) \dots t_{n-1}(lk_{n-1}):v$  is in  $\rho(D_1) \cup \rho(D_2)$ , there exists an element in  $E_1$  (due to the WLOG assumption above) with rooted key value  $t_1(lk_1) \dots t_{n-1}(lk_{n-1})$ . This element is  $\rho_1^{-1}(\sigma^{-1}(\text{parent}(E_3)))$  and we have  $\text{rkV}(\rho_1^{-1}(\sigma^{-1}(\text{parent}(E_3)))) = t_1(lk_1) \dots t_{n-1}(lk_{n-1})$ . Similarly, we have  $\text{rkV}(\rho_1^{-1}(\sigma^{-1}(E_3))) = t_1(lk_1) \dots t_{n-1}(lk_{n-1})$ . From our understanding of rooted key values, we simply take off one step to get to a parent's rooted key value and now

directly from the previous equation, we have  $\text{rkV}(\text{parent}(\rho_1^{-1}(\sigma^{-1}(E_3)))) = t_1(lk_1) \dots t_{n-1}(lk_{n-1})$ . Putting these two equations together, we get  $\text{rkV}(\rho_1^{-1}(\sigma^{-1}(\text{parent}(E_3)))) = t_1(lk_1) \dots t_{n-1}(lk_{n-1}) = \text{rkV}(\text{parent}(\rho_1^{-1}(\sigma^{-1}(E_3))))$ . Two elements with the same rooted key value are equal, and so  $\rho_1^{-1}(\sigma^{-1}(\text{parent}(E_3))) = \text{parent}(\rho_1^{-1}(\sigma^{-1}(E_3)))$ . Using this equation, the definition of  $\phi_3$  and simple algebra, we get  $\phi_3(\text{parent}(E_3)) = \phi_1(\rho_1^{-1}(\sigma^{-1}(\text{parent}(E_3)))) = \phi_1(\text{parent}(\rho_1^{-1}(\sigma^{-1}(E_3)))) = \text{parent}(\phi_1(\rho_1^{-1}(\sigma^{-1}(E_3)))) = \text{parent}(\phi_3(E_3))$ , as desired. It should be clear that  $\rho_3(D_3)$  is indeed  $\rho(D_1) \cup \rho(D_2)$ . Also, note that  $\sigma$  is the same function as  $\rho_3^{-1}$ .

**Key-exact:** Intuitively,  $D_3$  is key-exact since  $D_1$  and  $D_2$  are. Let  $E_3$  be an element in  $D_3$ . Let  $\rho_1$ ,  $\rho_2$  and  $\rho_3$  be the restrictions of  $\rho$  to  $D_1$ ,  $D_2$  and  $D_3$ , respectively. Let  $p$  be a path relative to  $E_3$  required for  $E_3$ 's local key. We know there exists  $E_1$  in  $D_1$  or  $D_2$  such that  $\rho_1(E_1) = \rho_3(E_3)$  or  $\rho_2(E_1) = \rho_3(E_3)$ . WLOG assume  $E_1$  in  $D_1$ . There must exist an element at the end of  $p$  in  $E_1$  in  $D_1$ , call that element  $F_1$ . Now,  $\sigma(\rho_1(F_1))$  is an element in  $D_3$  that is a descendant of  $E_3$  and has relative path  $p$  from  $E_3$ .

Let  $F_3 = \sigma(\rho_1(F_1))$ , let  $F_3'$  be a descendant of  $E_3$  also having relative path  $p$  from  $E_3$ . We must show  $\text{value}(F_3) = \text{value}(F_3')$ . We know  $\sigma^{-1}(F_3') = \sigma^{-1}(E_3).p'$ , where  $p'$  is a prefix path suffix. This implies that  $\sigma^{-1}(E_3):v$  is in  $\rho(D_2)$  for some value  $v$ , which implies that  $\rho_2^{-1}(\sigma^{-1}(E_3))$  is an element in  $D_2$ . We focus on  $\rho_2^{-1}(\sigma^{-1}(E_3))$  as opposed to the analogue for  $D_1$  since this is the difficult case. If we have  $F_3' = \sigma(\rho_1(F_1'))$  for some  $F_1'$  in  $D_1$ , we know  $\text{value}(F_3') = \text{value}(F_3)$  since  $D_1$  is key-exact. So, let  $E_2 = \rho_2^{-1}(\sigma^{-1}(E_3))$ . Since  $\rho_1(E_1) = \rho_2(E_2)$  this implies that  $\text{rkV}(E_1, D_1, T, \phi_1) = \text{rkV}(E_2, D_2, T, \phi_2)$ . Let  $t_1 \dots t_n$  be the path from the root of  $D_1$  to  $E_1$ , since  $\text{rkV}(E_1, D_1, T, \phi_1) = \text{rkV}(E_2, D_2, T, \phi_2)$ , we know that  $t_1 \dots t_n$  is the path from the root of  $D_2$  to  $E_2$  (names are contained in rooted key values), and since EMTs can't have siblings with the same names, we have  $\phi_1(E_1) = \phi_2(E_2)$ . This fact implies that the local key (not local key value) for  $E_1$  and  $E_2$  is the same. Further, since the rooted key values of  $E_1$  and  $E_2$  are the same, we also know that  $\text{lkV}(E_1, D_1, T, \phi_1) = \text{lkV}(E_2, D_2, T, \phi_2)$ . Let  $p_1 \dots p_m$  be the local key for  $E_1$  and  $E_2$ . Now  $\text{lkV}(E_1, D_1, T, \phi_1) = \text{patheval}(E_1, D_1, p_1) \dots \text{patheval}(E_1, D_1, p_m)$ . Similarly,  $\text{lkV}(E_2, D_2, T, \phi_2) = \text{patheval}(E_2, D_2, p_1) \dots \text{patheval}(E_2, D_2, p_m)$ . Thus for  $i = 1$  to  $m$ ,  $\text{patheval}(E_1, D_1, p_i) = \text{patheval}(E_2, D_2, p_i)$ . Since  $p = p_i$  for some  $i$ , we know  $\text{value}(F_3) = \text{value}(F_3')$  as desired. Thus,  $D_3$  must be key-exact.

**Key-respecting:** Intuitively,  $D_3$  is key-respecting since  $D_1$  and  $D_2$  are key-consistent. Let  $E_1$  and  $E_2$  be elements in  $D_3$  such that  $\text{rkV}(E_1, D_1, T, \phi_1) = \text{rkV}(E_2, D_2, T, \phi_2)$ . We must show we show  $E_1 = E_2$  (they are the same element). Let  $\rho_3$  be the restriction of  $\rho$  to  $D_3$ . Recall  $\rho_3$  is a bijection.  $\rho_3(E_1) = \text{rkV}(E_1, D_1, T, \phi_1):value(E_1)$ ,  $\rho_3(E_2) = \text{rkV}(E_2, D_2, T, \phi_2):value(E_2)$ . Since both of these paths are

in  $\rho(D_3) = \rho(D_1) \cup \rho(D_2)$  and since  $\rho(D_1)$  and  $\rho(D_2)$  are key-consistent,  $\rho_3(E_1)$  and  $\rho_3(E_2)$  must have identical terminal element value strings, so  $\rho_3(E_1) = \rho_3(E_2)$ . Since  $\rho_3$  is a bijection,  $E_1 = E_2$ , as desired.  $\square$

## 6. Merge-Lattice Theorem

We finish by stating and proving the Merge-Lattice Theorem. We prove that the set of all XML documents forms an upper semi-lattice under the partial ordering specified below. Merge is defined to be the join operation of this semi-lattice. Recall that an upper semi-lattice is a partially ordered set,  $S$ , such that for any two elements of  $S$ , the LUB of  $s_1$  and  $s_2$  always exist, are unique and are elements of  $S$ . The LUB operation is also called the Join operation of the lattice.

**Merge-Lattice Theorem:** *Let  $T$  be a Merge Template. The set of all XML documents plus  $TOP$  forms an upper semi-lattice under the following partial order. Let  $D_1$  and  $D_2$  be arbitrary XML documents: if  $D_1$  and  $D_2$  are mergeable with respect to  $T$ , then  $D_1 \leq D_2$  if and only if  $D_1 \sqsubseteq D_2$ , else  $D_1$  and  $D_2$  are not comparable.  $TOP$  is an object that is defined to be strictly greater than all documents;  $D < TOP$  for all documents  $D$ .*

*Proof:* Let  $D_1$  and  $D_2$  be two documents. We must show that there exists a unique least upper bound (LUB) for these two documents with respect to the order defined above. In the cases where  $D_1$ ,  $D_2$  and  $D_3$  are compatible with  $T$ , let  $\phi_1$ ,  $\phi_2$  and  $\phi_3$  represent the compatibility mappings between  $D_1$ ,  $D_2$  and  $D_3$  and  $T$ , respectively. As before, we write  $\rho(D_1)$  for  $\rho(D_1, T, \phi_1)$  and similarly for  $\rho(D_2)$  and  $\rho(D_3)$ .

Case 1: Either  $D_1$  or  $D_2$  is not compatible or key-respecting with respect to  $T$ . Without loss of generality, assume  $D_1$  is not compatible or key-respecting with respect to  $T$ . Thus,  $D_1$  is not mergeable with any document and is therefore not comparable to any document, so  $TOP$  is the unique LUB for  $D_1$  and  $D_2$ .

Case 2:  $D_1$  and  $D_2$  are compatible and key-respecting with respect to  $T$ , but are not key-consistent. Clearly  $TOP$  is an upper bound for  $D_1$  and  $D_2$ , we claim  $TOP$  is the unique LUB of  $D_1$  and  $D_2$ . For contradiction, assume there exists a document,  $D_3$ , such that  $D_1 \leq D_3$ ,  $D_2 \leq D_3$  and  $D_3 < TOP$ . By Theorem 1,  $\rho(D_1) \subseteq \rho(D_3)$  and  $\rho(D_2) \subseteq \rho(D_3)$ . Since  $D_1$  and  $D_2$  are not key-consistent,  $\rho(D_1) \cup \rho(D_2)$  is not key-respecting and there exists  $p_1 \in \rho(D_1) \subseteq \rho(D_3)$  and  $p_2 \in \rho(D_2) \subseteq \rho(D_3)$  such that  $p_1$  and  $p_2$  are equal except for conflicting terminal element value strings. Let  $\rho_3$  be the restriction of  $\rho$  to  $D_3$ ;  $\rho_3$  is a bijection. Now,  $\rho_3^{-1}(p_1)$  and  $\rho_3^{-1}(p_2)$  are elements in  $D_3$  with different values, but the same rooted key value. Hence,  $D_3$  is not key-respecting with respect to  $T$  and therefore not comparable to any document, contradicting the assumption that  $D_1 \leq D_3$  and  $D_2 \leq D_3$ . Thus,  $TOP$  is the unique LUB for  $D_1$  and  $D_2$ .

Case 3:  $D_1$  and  $D_2$  are compatible and key-consistent with respect to  $T$ , but are not mergeable.  $D_1$  and  $D_2$  are not comparable, so  $TOP$  forms an upper bound for these two documents. But is  $TOP$  the unique least upper bound? For contradiction, assume there exists a document,  $D_3$ , such that  $D_1 \leq D_3$ ,  $D_2 \leq D_3$  and  $D_3 < TOP$ . By Theorem 1, we have  $\rho(D_1) \subseteq \rho(D_3)$  and  $\rho(D_2) \subseteq \rho(D_3)$ . Let  $\rho(D_1.\text{root}) = t_1(lk_1):v_1$  and  $\rho(D_2.\text{root}) = t_2(lk_2):v_2$ . Since  $D_1$  and  $D_2$  are not mergeable, it must be the case that  $lk_1 \neq lk_2$  and  $\rho(D_1.\text{root}) \neq \rho(D_2.\text{root})$ . However, both of these paths must be in  $\rho(D_3)$  and valid path sets can not have two paths of length one, so no such  $D_3$  can exist and  $TOP$  must be the unique LUB of  $D_1$  and  $D_2$ .

Case 4:  $D_1$  and  $D_2$  are mergeable with respect to  $T$ . Let  $D_3$  be the document such that  $\rho(D_3) = \rho(D_1) \cup \rho(D_2)$ . Theorem 2 guarantees  $D_3$  exists and is compatible and key-respecting with respect to  $T$ . Since  $\rho(D_1) \subseteq \rho(D_3)$ , by Theorem 1, we know  $D_1 \leq D_3$ . Similarly,  $D_2 \leq D_3$ . Thus,  $D_3$  is an upper bound for  $D_1$  and  $D_2$ . We show  $D_3$  is the least such upper bound. Let  $D_4$  be a document such that  $D_1 \leq D_4$  and  $D_2 \leq D_4$ . We must show  $D_3 \leq D_4$ . By Theorem 1 we know  $\rho(D_1) \subseteq \rho(D_4)$  and  $\rho(D_2) \subseteq \rho(D_4)$ . Thus,  $\rho(D_3) = \rho(D_1) \cup \rho(D_2) \subseteq \rho(D_4)$  and by Theorem 1,  $D_3 \leq D_4$ , so  $D_3$  must be the unique LUB of  $D_1$  and  $D_2$ .  $\square$

We have thus provided a formal definition of the Merge operation as a lattice-join.

## 7. Conclusion

The function of the Merge operation is to take two similarly-structured XML documents and merge them together to create a new result document. Merge is useful for allowing fine-grained transfers between operators and for creating aggregates over data streams. We have shown that Merge is equivalent to the join operation of a semi-lattice. In addition, we have presented a useful representation of XML documents as prefix path sets. A Merge operator has been implemented in the Niagara Internet Query System and in the near future, we plan to test its performance. In addition, work is necessary to allow Merge to effectively handle ordered XML documents. We plan to proceed to investigate Merge in the various contexts such as partial evaluation and accumulation as discussed in the Introduction.

## Acknowledgements

We would like to acknowledge the help we have received from Kristin's dissertation committee: David DeWitt, Jeffrey Naughton and Raghu Ramakrishnan as well as the useful discussions and communications we have had with the members of the University of Pennsylvania database group. Funding for this work was provided by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908 and by NSF ITR award IIS0086002.

## References

- [1] S. Abiteboul, N. Bidoit. Non First Normal Form Relations: An Algebra Allowing Data Restructuring. *Journal of Computer Science and Systems*, 33(3), 361-393, December 1986.
- [2] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, J. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. In *Proceedings of the 1998 VLDB Conference*, New York, NY, August 1998.
- [3] S. Babu and J. Widom. Continuous Queries over Data Streams. In *SIGMOD Record*, Sept 2001.
- [4] P. Buneman, A. Deutsch, and W.C. Tan. A deterministic model for semi-structured data. In *Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israel, January 1999.
- [5] P. Buneman, S. Davidson, W. Fan, C. Hara, W.C. Tan. Keys for XML. In *Tenth International World Wide Web Conference (WWW10)*, Hong Kong, May, 2001.
- [6] P. Buneman, S. Khanna, K. Tajima, W.C. Tan. Archiving Scientific Data. In *Proceedings of the 2002 ACM SIGMOD Conference*, Madison, WI, June, 2002.
- [7] V. Christophides, S. Cluet, and J. Simeon. On Wrapping Query Languages and Efficient XML Integration. *Proceedings of the 2000 ACM SIGMOD Conference*, Dallas, TX, May 2000.
- [8] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2):3-18, June 1995.
- [9] H. Liefke and S. B. Davidson. View Maintenance for Hierarchical Semistructured Data. *2nd International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2000)*, London-Greenwich, United Kingdom, September 2000.
- [10] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54-66, September 1997.
- [11] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, D. Weld. An Adaptive Query Execution System for Data Integration. *Proceedings of the 1999 ACM-SIGMOD Conference*, Philadelphia, PA, June 1999.
- [12] Z. G. Ives, A. Y. Halevy, D. S. Weld. Integrating Network-Bound XML Data. *IEEE Data Engineering Bulletin* 24(2):27-33, June 2001.
- [13] J. Naughton, D. DeWitt, D. Maier, et al. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24(2): 27-33, June 2001.
- [14] Niagara Internet Query Engine. Version 1.0. Available from: <http://www.cs.wisc.edu/niagara/>.
- [15] J. Shanmugasundaram, K. Tufte, D. DeWitt, D. Maier, J. Naughton. Architecting a Network Query Engine for Producing Partial Results. *Workshop on the Web and Databases (WebDB)*, May 2000.
- [16] I. Tatarinov, Z. Ives, A. Halevy, D. Weld. Updating XML. In *Proceedings of the 2001 ACM SIGMOD Conference*, Santa Barbara, CA, May 2001.
- [17] K. Tufte, D. Maier. Accumulation and Aggregation of XML Data. *IEEE Data Engineering Bulletin*, 24(2):34-39, June 2001.
- [18] Y. Zhuge, H. Garcia-Molina. Graph Structured Views and Their Incremental Maintenance. *International Conference on Data Engineering (ICDE'98)*, Orlando, FL, February 1998.

## Appendix A

This appendix contains the XML document corresponding to the the Merge Template shown in Figure 4.

```
<?xml version = "1.0"?>
<!DOCTYPE DocMergeTemplate MergeTemplate.dtd>
<DocMergeTemplate MergeType= "outer">
<EltMergeTemplate Name = "Auction">
  <Empty>
  <EltMergeTemplate Name = "Item">
    <LocalKey> <PathNode>
      <Path> ID </Path>
      <Content ValueType = "Integer"/>
    </PathNode> </LocalKey>
  <Empty/>
  <EltMergeTemplate Name = "ID">
    <MustMatch/>
  </EltMergeTemplate> <!-- End ID -->
  <EltMergeTemplate Name = "Desc">
    <ShallowContent Function = "replace"/>
  </EltMergeTemplate> <!-- End Desc -->
  <EltMergeTemplate Name = "Bid">
    <LocalKey> <PathNode>
      <Path> Bidder </Path>
      <Content ValueType = "String"/>
    </PathNode>
    <PathNode>
      <Path> Price </Path>
      <Content ValueType= "Float"/>
    </PathNode> </LocalKey>
  <Empty/>
  <EltMergeTemplate Name = "Bidder">
    <MustMatch/>
  </EltMergeTemplate> <!--End Bidder -->
  <EltMergeTemplate Name = "Price">
    <MustMatch/>
  </EltMergeTemplate> <!-- End Price -->
</EltMergeTemplate> <!-- End Bid -->
</EltMergeTemplate> <!-- End Item -->
</EltMergeTemplate> <!-- End Auction -->
```