A Model for XML Instance Level Integration

Aldo Monteiro do Nascimento[1], **Carmem S. Hara**[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19.081 – 81.531-990 – Curitiba – PR – Brasil

{aldo, carmem}@inf.ufpr.br

Abstract. *There are two major problems for merging instances from different sources in order to build a datawarehouse: entity identification ambiguity and attribute value conflict. In this paper we propose a data model that facilitates the resolution of value attribute conflicts by explicitly representing them in the integrated schema. In this model, the datawarehouse is an XML tree populated with data imported from one or more XML sources, and nodes are annotated with provenance information. The purpose of annotations is twofold: first, they represent the origin of every element in the datawarehouse. This information is essential for determining the quality and amount of trust one places on the data. Second, they allow the portion of source XML tree used to populate the warehouse to be reconstructed. This capability is important if one needs the original document to compare with new releases from the same source in order to incrementally update the warehouse. Algorithms for populating the warehouse according to the proposed model and for reconstructing the source data are presented. We also report results from an experimental study conducted to determine the impact of the annotations on the size of the warehouse.*

1. Introduction

Cooperation between institutions from the same area and even from distinct areas has created the need for building databases that store large amounts of information on a specific subject. One approach to accomplish this task is to build a *datawarehouse*. A datawarehouse is an integrated repository of data generated from many sources [Pokorný 2002]. Usually, distinct data sources store data in different models and structures. Moreover, data that refer to the same entity in the real world may have different representations in each source. Thus, in order to succeed in combining data, the datawarehouse has to rely on its flexibility.

XML is a semi-structured data model that has become the choice both in data and document management systems because of its capability of representing irregular data while keeping the data structure as much as it exists [Sawires et al. 2005]. XML flexibility allows documents to be extended or reduced to describe content of any size, even when the structure of the document changes. Combining these technologies, datawarehouse and XML, is a step towards obtaining the flexibility that a warehouse application needs. A number of XML datawarehouses have been proposed in the literature. Some address the problem of schema matching and integration [Xyleme 2001, Draper et al. 2001, Rundensteiner et al. 2000], while others concentrate on instance level integration. There are two major problems for combining instances from different sources [Prabhakar et al. 1993]: entity identification ambiguity and attribute value conflict. These are problems that in recent literature have been addressed as *data cleaning*

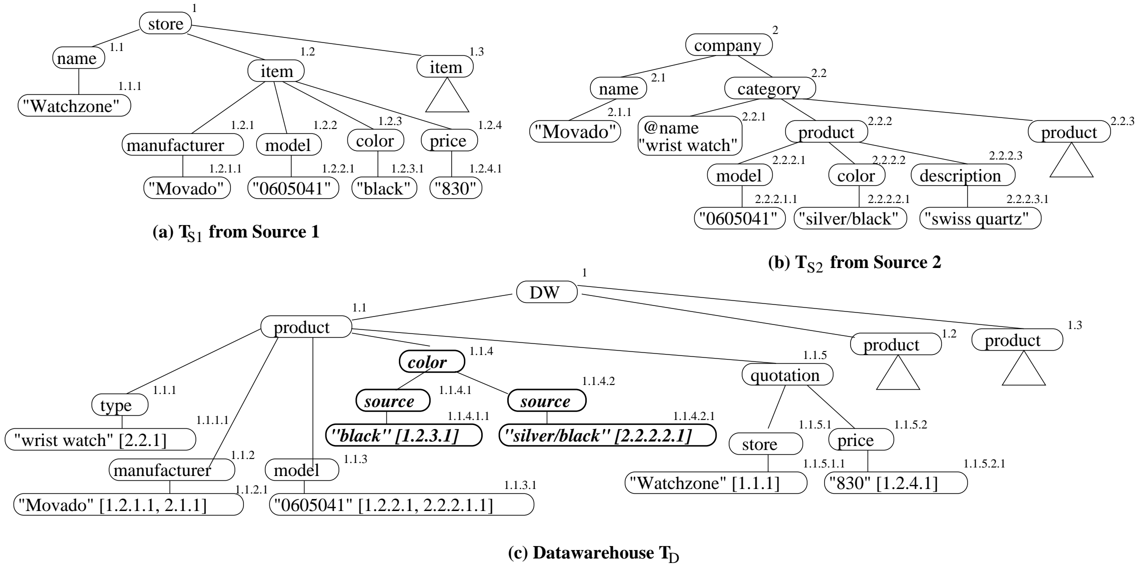
[Rahm and Do 2000]. Entity identification refers to the problem of identifying overlapping data in different sources. It has been the purpose of extensive research on the relational [Lim et al. 1996], entity-relationship [Menestrina et al. 2006], and XML [Poggi and Abiteboul 2005] data models. Attribute value conflict refers to the problem of two or more sources containing information on the same entity or attribute, but with conflicting values. Some existing approaches for addressing this problem include data profiling, data mining, and constraint-based techniques [Prabhakar et al. 1993] [Rahm and Do 2000].

In this paper we propose a data model that facilitates resolution of value attribute conflicts by explicitly representing them in the integrated schema. As an example, consider two data sources on the domain of products, illustrated in Figure 1(a) and (b). The document denoted as `Source 1` contains information on products obtained from an online store, while `Source 2` contains information provided by a manufacturer. A mapping from `Source 1` to a datawarehouse with the structure depicted in Figure 1(c) defines that: `name` of the `store` is mapped to the `store` of a `product`'s `quotation`; an `item` sold by the store is mapped to a `product`, and its subelements `manufacturer, model` and `color` are mapped to `product`'s subelements; the `price` of the `item`, on the other hand, is placed under a `quotation` element. We define a mapping from `Source 2` to the datawarehouse in a similar way. Given that in the datawarehouse we define that whenever two products agree on their `manufacturer` and `model` they refer to the same entity in the real world, the resulting tree is given by Figure 1(c). This can be expressed as XML keys [Buneman et al. 2002a] defined on the datawarehouse. That is, `manufacturer` and `model` are keys for `product`, and therefore whenever two products agree on their values, they should be merged in the datawarehouse.

Observe that the data sources disagree on the color of the product, and this is represented by creating two distinct elements for each value, as highlighted in the figure. The goal is to facilitate the process of solving the attribute conflict problem. Conflicts that have been identified can then be eliminated by a cleaning process.

Nodes in our model are annotated with provenance information. The purpose of the annotations is twofold: first, we would like to be able to determine the origin of every element in the datawarehouse. This information is essential for determining the quality and amount of trust one places on the data [Tan 2007], and can be used as a parameter for solving a conflict. In the example of Figure 1, one can trust that the color provided by a manufacturer is accurate, and thus give priority to this value over the one given by a store. Second, we would like to be able to reconstruct the portion of the XML tree used to populate the warehouse. This reconstruction is important for automating the process of updating the warehouse. More specifically, given the new version of the document and the reconstructed version, it is possible to run a diff algorithm to determine what are their differences. Then, based on the source mapping, the system can determine what are the updates to be applied to the datawarehouse to keep it up to date. Without the reconstruction capability, the original document would have to be locally stored for incrementally updating the warehouse.

Contributions. In this paper we propose a data model for XML instance level integration. It is a first step towards designing a mechanism for automating the process of



(a) T_{S1} from Source 1

(b) T_{S2} from Source 2

(c) Datawarehouse T_D

Figure 1. Merging of two data sources

incrementally updating a datawarehouse based on new versions of source data. We make the following contributions:

- a data model for merging source data based on XML Keys [Buneman et al. 2002a] that explicitly represents value conflicts and allow the source document to be reconstructed based on provenance annotations;
- algorithms for populating a datawarehouse structured according to the proposed model and for reconstructing source data from the warehouse;
- an experimental study to determine the impact of provenance annotations on the size of the datawarehouse.

Organization. The rest of the paper is organized as follows. Section 2 describes a tree model for XML, and XML Keys. Our proposed model, a mapping language for populating the datawarehouse, and algorithms are presented in Section 3. Experimental results are given in Section 4, followed by a discussion on related work in Section 5. Section 6 concludes the paper by presenting some future work.

2. Preliminary Definitions

We begin with the definitions of XML trees and the class of XML keys which will be used throughout the paper.

2.1. XML Tree

An XML document is typically modeled as a node-labeled tree. We assume three pairwise disjoint sets of labels: **E** of element tags, **A** of attribute names, and a singleton set $\{\mathtt{S}\}$ denoting text (PCDATA).

Definition 1 An *XML tree* T is defined to be $T = (source, V, r, id, lab, ele, att, val)$, where (1) $source$ is the XML tree source identification; (2) V is a set of nodes; (3) r is the unique and distinguished root node; (4) id is a function that assigns unique identifiers to nodes in V; given a node v, $id(v)$ returns a vector that represents the path from r to v; (5) lab is a mapping $V \rightarrow \mathbf{E} \cup \mathbf{A} \cup \{\mathtt{S}\}$ which assigns a label to each node in V; a node v in V is called an *element* if $lab(v) \in \mathbf{E}$, an *attribute* if $lab(v) \in \mathbf{A}$, and a text node if $lab(v) = \mathtt{S}$; (6) ele and att are partial mappings that define the edge relation of T: for any node v in V,

- if v is an element then $ele(v)$ is a *list* of elements and text nodes in V and $att(v)$ is a *set* of attributes in V; for each v' in $ele(v)$ or $att(v)$, v' is called a *child* of v and we say that there is a (directed) edge from v to v';
- if v is an attribute or a text node then $ele(v)$ and $att(v)$ are undefined;

(7) val is a partial mapping that assigns a string to each attribute and text node: for any node v in V, if v is an attribute or text node then $val(v)$ is a string, and $val(v)$ is undefined otherwise.

An XML tree has a tree structure, i.e., for each $v \in V$, there is a unique path of edges from root r to v. □

Examples of XML trees are given in Figure 1(a) and (b), where each node v is represented with its identifier ($id(v)$), a label ($lab(v)$) if it is a an element or attribute node, and a value ($val(v)$) if it is an attribute or text node. The encoding adopted by the identifier function id is called *Dewey Order* [Tatarinov et al. 2002], which provides a global node ordering. In our model, we assume that each data source has a distinct $source$ identifier, and that this value is used as the identifier of the root node; that is, $id(r) = source$. In our running example, `Source 1` has identifier 1, so in Figure 1(a), $id(r) = 1$. We define function $origin$ that receives an identifier i of a node in T, and returns its $source$; that is $origin(i) = source(T)$.

2.2. XML Keys

Various forms of key specification have been proposed in the literature [Bray et al. 1998, Fallside 2000]. We use the key specification proposed in [Buneman et al. 2002a] because it provides a method for defining *relative keys*. To define a key three things are specified: 1) the *context* in which the key must hold; 2) a *target* set on which we are defining a key; and 3) the *values* which distinguish each element of the target set.

The path language we adopt is a common fragment of regular expressions [Hopcroft and Ullman 1979] and XPath [Clark and DeRose 1999]:

$$Q \quad ::= \quad \epsilon \quad | \quad l \quad | \quad Q/Q \quad | \quad //$$

where ϵ is the empty path, l is a node label, "/" denotes concatenation of two path expressions (*child* in XPath), and "//" means *descendant-or-self* in XPath. To avoid confusion we write $P//Q$ for the concatenation of P, $//$ and Q. A *path* p is a sequence of labels $l_1/\ldots/l_n$. We denote by $size(p)$ the number of labels in p. A *path expression* Q defines a set of paths, while "$//$" can match any path. We use $\rho \in Q$ to denote that ρ is in the set of paths defined by Q. For example, $/product/quotation/store \in //store$. Given an XML tree T, we denote by $Paths(T)$ the set of paths in T. As an example, let T be the XML tree depicted in Figure 1(a). Then $Paths(T) = \{/, /name, /item, /item/manufacturer, /item/model, /item/color, /item/price\}$.

Following the syntax of [Buneman et al. 2002a] we write an XML key as:

$$\varphi : (Q_1, (Q_2, \{P_1, \ldots, P_p\}))$$

where φ is the key identifier, path expressions Q_1 and Q_2 are the context and target path expressions respectively, and $P_1, ..., P_p$ are the key paths. For the purposes of this paper, we restrict the key paths to be attributes $@A_1, ..., @A_p$ or simple text valued elements. A key is said to be *absolute* if the context path Q_1 is the empty path ϵ, and *relative* otherwise.

Example 1 Consider the XML tree in Figure 1(c). Some keys that can be defined on this tree are:

- $K_1 : (\epsilon, (product, \{manufacturer, model\}))$: in the context of the entire document (ϵ denotes the root), a `product` is identified by its `manufacturer` and `model` number;
- $K_2 : (/product, (quotation, \{store\}))$: within the context of any subtree rooted at a `product` node, a `quotation` is identified by its `store` name.
- $K_3 : (//product, (type, \{\}))$: each `product` has at most one `type`; similarly, $K_4 : (//product, (color, \{\}))$ for the `color` of the `product`, and $K_5 : (//product/quotation, (price, \{\}))$ for the `price` of a `quotation`. □

To define the meaning of an XML key, we use the following notation: in an XML tree T, $n[\![P]\!]_T$ denotes the set of nodes in T that can be reached by following path expression P from node n. We also use $[\![P]\!]_T$ as an abbreviation for $r[\![P]\!]_T$, where r is the root node of the tree. As an example, in Figure 1(b), $[\![//product]\!]_{T_{S2}} = \{2.2.2, 2.2.3\}$, and $2.2.2[\![color]\!]_{T_{S2}} = \{2.2.2.2\}$.

Definition 2 An XML tree T *satisfies* a key $\varphi : (Q_1, (Q_2, \{P_1, \ldots, P_p\}))$, denoted by $T \models \varphi$, iff for any n in $[\![Q_1]\!]_T$ and any n_1, n_2 in $n[\![Q_2]\!]_T$, (1) n_1 and n_2 each has a unique node reached by following path P_i for all $i \in [1, p]$, and (2) if $val(n_1.P_i) = val(n_2.P_i)$ for all $i \in [1, p]$ then $n_1 = n_2$, where $val(n'.P_i)$ denotes the text value associated with the single node in $n'[\![P_i]\!]_T$. □

The implication problem for this class of XML Keys has been addressed in [Davidson et al. 2007]. A set of inference rules $\mathcal{I}$ is defined such that for any set Σ of XML keys and a single key φ, it is possible to determine whether Σ entails φ, denoted as $\Sigma \models \varphi$, if φ can be proved from Σ using $\mathcal{I}$. In particular, the definition for key satisfaction states that every key path is unique under the target node. Thus, from any $(Q_1, (Q_2, \{P_1, \ldots, P_p\})) \in \Sigma$ we can derive $(Q_1/Q_2, (P_i, \{\}))$ for every $i \in [1, p]$. Let us denote this rule as *keyValues*.

Example 2 The XML tree of Fig. 1(c) satisfies all key constraints given in Example 1. Moreover, the following keys can be derived applying the *keyValues* rule. From K_1 we can derive $K_6 : (/product, (manufacturer, \{\}))$ and $K_7 : (/product, (model, \{\}))$; that is, every `product` has a unique `manufacturer` and `model`. Similarly, from K_2 we can derive $K_8 : (/product/quotation, (store, \{\}))$. □

3. Datawarehouse

The datawarehouse is an XML tree populated with data imported from one or more XML sources. We assume that the datawarehouse has a fixed schema, and every element is identifiable via XML keys. Keys defined on the datawarehouse determine how to merge data originated from different sources.

Example 3 Consider the XML keys defined in Example 1, and XML trees depicted in Figure 1. Observe that according to key K_1, in the datawarehouse a `product` element is uniquely identified by its `manufacturer` and `model`. Suppose we define that element `manufacturer` is populated with nodes reached by following path $/item/manufacturer$ from `Source 1`, and nodes reached by path $/name$ from `Source 2`; similarly, element `model` is populated by nodes reached by following path $/item/model$ from `Source 1` and path $/category/product/model$ from `Source 2`. Then whenever these two values coincide they are mapped to the same element in the datawarehouse, as depicted in Figure 1(c). □

In the datawarehouse, leaf nodes are annotated with provenance information. These annotations are important not only to determine the origin of data, but they also allow the portion of source XML tree used to populate the datawarehouse to be reconstructed. Some functionalities that may be implemented for the datawarehouse may require the original document to be available. Thus, the reconstruction capability is important for reducing the amount of storage used by the system. One particular problem in which this requirement exists is the process of incrementally updating of the datawarehouse when new versions of source data becomes available. In this scenario, given the rebuilt source data and the new release, it is possible minimize the number of updates by determining the differences between the two versions.

Definition 3 A datawarehouse populated from a set of source XML trees S is defined to be $D = (T_D, \mathcal{K}_D)$, where (1) T_D is an XML tree with a set of nodes V such that each leaf node $v \in V$ is annotated with a set of pairs (id_n, p), where id_n is the identifier of a node n in a source XML tree $T \in S$ used to populate v, and p is the path in T from the root to n; (2) $\mathcal{K}_D$ is the set of XML keys defined on T_D. Every element node in T_D can be uniquely identified according to keys in $\mathcal{K}_D$ or have a single `S` node as a child. □

As an example, nodes in the datawarehouse depicted in Figure 1(c) have been annotated with node identifiers from `Source 1` and `Source 2`. Paths traversed from the root have been omitted for simplicity. Moreover, the set of keys given by Example 1 in addition to those defined in Example 2 satisfy the key identification requirement. This requirement is similar to the notion of *insertion-friendly* set of keys defined in [Buneman et al. 2002a]. Intuitively, every element is either a simple text element or "keyed" by some values according to K_D.

3.1. Mapping Language

In order to describe how data are extracted from the sources, we define a simple language, which consists of a set of rules $p_D \leftarrow p_s$, where p_D is a path in the datawarehouse, and p_s is a path in the data source.

Example 4 Consider data source `Source 1` and datawarehouse T_D depicted in Figure 1(a) and (c), respectively. A mapping from `Source 1` to T_D can be specified as follows.

$$\begin{array}{ll}
/product & \leftarrow /item \\
/product/manufacturer & \leftarrow /item/manufacturer \\
/product/model & \leftarrow /item/model \\
/product/color & \leftarrow /item/color \\
/product/quotation/store & \leftarrow /name \\
/product/quotation/price & \leftarrow /item/price
\end{array}$$

□

In general, several mappings can be specified to populate a datawarehouse from the same or different sources. Each mapping is defined as a pair $M = (source, R)$, where $source$ is a source identifier, and R is a set of rules. We denote by $order(M)$ the number of rules in R. Observe that the meaning of a mapping is given by the structure of the datawarehouse and the set of keys. That is, a key $\varphi = (Q_1, (Q_2, K))$ determines that whenever two nodes n_1 and n_2 are reached by following a path $p \in Q_1/Q_2$ and they agree on the values of their K subelements, then they should be mapped to the same n_D node in the datawarehouse. Thus, we define the following restriction of a *well-defined* mapping: if $M = (source, R)$ is a mapping defined for datawarehouse $D = (T_D, \mathcal{K}_D)$,

and $\varphi = (Q_1, (Q_2, K))$ is a key in $\mathcal{K}_D$ then for every path $p_D \in Paths(T_D)$ and $p_K \in K$ such that $p_D \in Q_1/Q_2$, there exists a rule $p_D/p_K \leftarrow p'$ in R for some p'. Intuitively, this says that every key in the datawarehouse must be populated from some element in a data source. As an example, the mapping given in Example 4 is well-defined since the key values defined for the datawarehouse are `model` and `manufacturer` for `product`, and `store` for `product/quotation`; moreover, the mapping defines that all key values are populated with data extracted from `Source 1`.

Despite its simplicity, our mapping language allows unnesting and nesting of source data, besides projection, union, Cartesian product, and join on key values. In the following sections we provide algorithms for populating a datawarehouse, and also for reconstructing a source document from the warehouse.

3.2. Populating the Datawarehouse

The algorithm for populating the datawarehouse consists of two steps. First, we build subtrees structured as the datawarehouse and with values extracted from a source. Then, these trees are merged with existing data in the warehouse according to their key values. The pseudo code is presented in Algorithm 1. It takes as input: a datawarehouse $D = (T_D, K_D)$, a mapping $M = (source, R)$, and a source tree T_s. The following data structures are used by the algorithm:

- $sourcePath$: an array of paths $[p_0, p_1, \ldots, p_m]$, where $p_0 = /$, and $p_i, i \in [1, n]$ are paths defined on the source data that are used to populate the warehouse. That is, they are paths on the right-hand side of rules in R. The list is in prefix order: if $p_j = p_i/Q$ for some path Q then $i < j$. The size of the list is given by $order(M)$, which is the number of rules in M.
- $dwPath$: an array of paths $[q_0, q_1, \ldots, q_m]$, where $q_0 = /$ and q_i is a path defined on the datawarehouse populated with path $sourcePath[i]$; that is, $q_i \leftarrow sourcePath[i]$ is a rule in the mapping.
- $sourceNode$: an array of nodes $[n_0, n_1, \ldots, n_m]$ in the data source such that n_i is the last node in $[\![sourcePath[i]]\!]_{T_s}$ visited by the algorithm.

Function `populate` first initializes all nodes in $sourceNode$ with $null$ values, and sets $sourceNode[0]$ to be the root of the source tree T_s. It then calls function `transformAndMerge` to build a subtree with nodes stored in $sourceNode$ and merge it with datawarehouse T_D (Lines 2 to 4). Each invocation of function `transformAndMerge` looks for a node reached by following path $sourcePath[ind]$ in the source tree T_s, and assign it to $sourceNode[ind]$. When nodes are assigned for every element in $sourceNode$ then a subtree T' is built by calling function `createTree` and subsequently merged with the datawarehouse by function `mergeTrees` (Lines 6 to 8). When looking for a node in $[\![sourcePath[ind]]\!]_{T_s}$, we have to restrict the search space to the smallest subtree rooted at a node that has already been stored in $sourceNode$. Recall that $sourcePath$ is prefix-ordered and function `transformAndMerge` considers elements in the array in ascending order of their indices. Then, there exists an $a < ind$ such that $sourcePath[a]$ is the longest prefix of $sourcePath[ind]$ in the array. Moreover, $sourceNode[a]$ is the root of the subtree in which nodes v reached by following path $sourcePath[ind]$ are searched for (Lines 10 to 13). After assigning a node v to $sourceNode[ind]$, function `transformAndMerge` is recursively called for filling up the $sourceNode$ array (Lines 14 to 16).

Function `createTree` builds a tree T' with the structure of the datawarehouse as specified by the mapping, and extract values from nodes in $sourceNode$ to populate leaves in T'. It starts by creating the root node r of T'. Then, for each node v in $sourceNode$, the path $p_D = /l_1/\dots/l_m$ in the warehouse populated with v is obtained from $dwPath$. If T' already contains a node n_a reached by following path $/l_1/\dots/l_a$, $a < m$ then only nodes for path $l_{a+1}/\dots/l_m$ are generated as descendents of n_a (Lines 20 to 25). The last node receives the value extracted from the source tree and provenance data (attribute `@prov`) if it is a leaf node (Lines 26 to 28).

Algorithm 1 Populate Datawarehouse Algorithm

1: **function** populate ($D = (T_D, K_D)$, $M = (source, R), T_s$)
2: initialize $sourceNode$ array with $null$ values;
3: $sourceNode[0] \Leftarrow r(T_s)$;
4: return transformAndMerge ($T_D, 1, sourceNode$);
5: **function** transformAndMerge ($T_D, ind, sourceNode$)
6: **if** $ind > order(M)$ **then**
7: $T' \Leftarrow$createTree ($sourceNode$);
8: return mergeTrees (T_D, T', K_D);
9: **else**
10: let a be the index s.t. $sourcePath[a]$ is the longest prefix of $sourcePath[ind]$;
11: let Q be the path s.t. $sourcePath[ind] = sourcePath[a]/Q$;
12: $anc \Leftarrow sourceNode[a]$;
13: $V \Leftarrow anc[\![Q]\!]_{T_s}$;
14: **for** each node v in V **do**
15: $sourceNode[ind] \Leftarrow v$;
16: $T_D \Leftarrow$ transformAndMerge ($T_D, ind + 1, sourceNode$);
17: **function** createTree ($sourceNode$)
18: create root node r of tree T';
19: **for** $i := 1$ to $order(M)$ **do**
20: **if** $sourceNode[i] \neq null$ and $dwPath[i] \neq null$ **then**
21: let p be the longest prefix of $dwPath[i]$ s.t. $[\![p]\!]_{T'}$ is not empty
22: let n_0 be the single node in $[\![p]\!]_{T'}$
23: let $dwPath[i]$ be $p/l_1/\dots/l_m$
24: **for** $j := 1$ to m **do**
25: create node n_j in T' as a child of n_{j-1} with label l_j;
26: **if** $\nexists p \in dwPath$ s.t. $dwPath[i]$ is a prefix of p **then**
27: $val(n_j) \Leftarrow val(sourceNode[i])$;
28: $n_j.@prov \Leftarrow \{(id(sourceNode[i]), sourcePath[i]\}$;
29: return T';

Example 5 Consider again the mapping given in Example 4 and `Source 1` depicted in Figure 1(a). Array $sourcePath$ generated from this mapping consists of the following values: $[/, /item, /item/model, /item/manufacturer, /item/color, /item/price, /name]$. Similarly, array $dwPath$ contains the following paths: $[/, /product, /product/model, /product/manufacturer, /product/color, /product/quotation/price, /product/quotation/store]$. The content of array $sourceNode$ after the first `item`

of Source 1 is traversed by function `transformAndMerge` is the following: $[1, 1.2, 1.2.2, 1.2.1, 1.2.3, 1.2.4, 1.1]$. Based on these nodes and the structure given by $dwPath$, function `createTree` builds a tree which consists of the following nodes: 1, 1.1, 1.1.2, 1.1.2.1, 1.1.3, 1.1.3.1, 1.1.4, 1.1.4.1.1, 1.1.5, 1.1.5.1, 1.1.5.1.1, 1.1.5.2, 1.1.5.2.1. Here, we use identifiers of nodes in datawarehouse T_D depicted in Figure 1(c). □

Merging XML Trees. The algorithm for merging trees receives two XML trees: the datawarehouse T_D and a source tree T_s that has already been transformed according to the datawarehouse schema. The result is the datawarehouse tree T_D with elements in T_s merged according to XML keys $\mathcal{K}$ defined on D. That is, whenever nodes in T_s and T_D agree on their key values they are merged into a single one in the datawarehouse; otherwise new elements are created. Value conflicts are represented by generating two distinct subelements containing values from both sources.

The pseudocode is given by Algorithm 2. Function `mergeTrees` first checks whether the datawarehouse is empty and if this is the case, the root node is created (Lines 2 and 3). Function `mergeNodes` is then called to extract values from each child of the source's root and insert them in the datawarehouse (Lines 4 to 6). The invoked function receives the datawarehouse T_D, a node $n_D \in V(T_D)$, which is the root of the subtree into which source nodes in the subtree rooted at n_s in $V(T_s)$ are to be inserted, and a set of keys $\mathcal{K}$ that determine when nodes should be merged. The function first checks whether source node n_s is a text or attribute node. If this is the case, function `mergeValues` is called (Lines 9 and 10). Recall that every element in the datawarehouse must have a key according to $\mathcal{K}$. Thus, if n_s is an element node, the function looks for a node in the datawarehouse with key values matching those of n_s. If such a node is found, they are merged and function `mergeNodes` is recursively called for merging their children; otherwise, the source subtree is copied into the datawarehouse (Lines 12 to 21). Function `mergeValues` is called either to create new attribute or text nodes or to compare their values with existing ones, generating nodes that point out value conflicts if they exist. If node n_D in the datawarehouse, under which an attribute node $@a$ or text node `S` is to be created, does not have any children with such label, a new one is created (Line 24 to 28). Otherwise it is checked if the new value agree with an existing one in the datawarehouse. If this is the case, provenance information is inserted in T_D (Lines 30 and 31). If this is not the case, then a value conflict has been detected and a new node with label `source` is created to identify the conflict (Lines 33 to 37).

Example 6 Consider the XML trees in Figure 1 and XML keys defined in Example 1. Suppose that the datawarehouse T_D has already been populated with all the elements in `Source 1`, and `Source 2` is now being considered. Recall that before calling function `mergeTrees`, `Source 2` has already been restructured by function `populate` to adhere to the datawarehouse structure. When processing source `product` node 2.2.2, it is checked whether its key values given by $K_1 : (\epsilon, (product, \{manufacturer, model\}))$ match a node in the warehouse. Since this is the case, it is merge with node 1.1 in T_D. Recursive calls to function `mergeNodes` annotate leaf nodes under `manufacturer` and `model` with provenance information. When considering source node `color` (2.2.2.2), T_D has already been populated with a color node. Since key $K_3 : (//product, (color, \{\}))$ defines that each `product` contains a single subelement `color`, when function `mergeValues` is called, the value in the source does not agree with the one already

Algorithm 2 Merge Algorithm

```
1:  function mergeTrees (T_D, T_s, 𝒦)
2:    if V(T_D) = {}  /* the datawarehouse is empty */ then
3:      create root node r_D of T_D;  else r_D ⇐ r(T_D);
4:    r_S ⇐ r(T_s);
5:    for n ∈ children(r_s) do
6:      T_D ⇐ mergeNodes(T_D, r_D, T_s, n, 𝒦);
7:    return T_D;

8:  function mergeNodes(T_D, n_D, T_s, n_s, 𝒦)
9:    if n_s is a text or attribute node then
10:     T_D ⇐ mergeValues(T_D, n_D, T_s, n_s, path_s);
11:   else
12:     let path_s be the path from the root of T_s to n_s;
13:     let path_D be the path from the root of T_D to n_D;
14:     let P be the path such that path_s = path_D/P;
15:     keyPaths ⇐ {p | 𝒦 ⊨ (Q_1, (Q_2, K)), path_s ∈ Q_1/Q_2, p ∈ K};
16:     if there exists n in n_D⟦P⟧_{T_D} s.t. for all k ∈ keyPaths val(n.k) = val(n_s.k)
        then
17:       for each node n' ∈ children(n_s) do
18:         T_D ⇐ mergeNodes(T_D, n, T_s, n', 𝒦);
19:     else
20:       copy the subtree rooted at n_s to T_D;
21:       insert n_s in children(n_D);
22:   return T_D;

23: function mergeValues(T_D, n_D, T_s, n_s, p)
24:   if n_s is an attribute node such that p = p'/@a then
25:     p_s ⇐ @a;  else p_s ⇐ S;
26:   nodes_D ⇐ n_D⟦p_s⟧_{T_D} ∪ n_D⟦source/p_s⟧_{T_D};
27:   if nodes_D = {}  /* no attribute or value element */ then
28:     copy n_s to T_D and insert it as a child of n_D;
29:   else
30:     if there exists a node n ∈ nodes_D such that val(n) = val(n_s) then
31:       val(n.@prov) ⇐ val(n.@prov) ∪ val(n_s.@prov);
32:     else
33:       create a source labelled node n'_s as a child of n_D;
34:       copy n_s to T_D as a child of n'_s;
35:       if n_D⟦p_s⟧_= {n}  /* this is the first value conflict */ then
36:         create a source labelled node n'_D as a child of n_D;
37:         move n from children(n_D) to children(n'_D);
38:   return T_D;
```

in T_D. As a consequence, `source` conflict subelements are created. Observe that without key K_3, there would be no restriction on the number of colors a `product` may have. Thus, the algorithm would not create a conflict node, but two `color` subelements for `product`, each originated from a given source. □

The overall complexity of the algorithm for populating the warehouse is $O(|T_s|^2|M||D|)$, where $|T_s|$ is the size of the source document, $|M|$ is the total size of paths in mapping rules, and $|D|$ the size of the datawarehouse tree T_D plus the size of keys in K_D [do Nascimento 2008].

3.3. Reconstruction of Data Sources

The reconstruction algorithm rebuilds the portion of the data source used to populate the datawarehouse based only on the annotations stored in the datawarehouse. The pseudo code is presented in Algorithm 3. Function `buildSource` takes datawarehouse D and a source identifier $source$ as parameters. The algorithm first creates the root node of the document source T_s (Line 2), and then computes the set ($Leaves$) of all leaf nodes in the datawarehouse. For each node in this set, it checks whether the annotation $@prov$ contains a pair (id_s, p_s) such that id_s is an identifier of a node in $source$ (Lines 4 and 5). If this is the case, function `createPath` is invoked to generate a new node in T_s reached by following path p_s.

Function `createPath` takes as input: a partially built source tree T; an identifier id_s of the form $j_0.j_1 \dots j_b$; a path $path_s$ in the source document of the form $/l_1/\dots/l_b$, and a leaf $node$ in the datawarehouse populated from a node in $source$. Let this source node be n_s. Then $id(n_s) = id_s$ and, in the original document T_s, $n_s \in [\![path_s]\!]_{T_s}$. Observe that from id_s and $path_s$ it is possible to create all source nodes in the path from the root leading to n_s. That is, this path traverses nodes $n_1, \dots n_b$ such that for each n_i, $id(n_i) = j_0 \dots j_i$ and $label(n_i) = l_i$. Before creating these nodes we check whether some of them have already been created (Lines 9 to 12) and then generate the remaining ones (Lines 13 to 15). The value of the last node in the path is extracted from the node in the datawarehouse ($node$). For keeping the relative order of elements in the source document, the first newly created node is correctly placed in the list of elements of its parent (Line 16 to 18).

Example 7 Consider the reconstruction of `Source 2` from the datawarehouse T_D depicted in Figure 1. Suppose that the reconstruction starts with leaf node 1.1.3.1 in $[\![/product/model/S]\!]_{T_D}$. Observe that $1.1.3.1.@\texttt{prov} = \{(1.2.2.1, /item/model/S), (2.2.2.1.1, /category/product/model/S)\}$. Given that the root node of `Source 2` with identifier 2 has already been created, the following new nodes are inserted in the source tree: 2.2 (a `category` node), 2.2.2 (a `product` node), 2.2.2.1 (a `model` node) and 2.2.2.1.1 (an `S` node). Moreover, the value "0605041" is assigned to the last node. Consider that the next leaf in the datawarehouse to be considered is node 1.1.1.1 in $[\![/type/S]\!]_{T_D}$. Observe that $1.1.1.1.@\texttt{prov} = \{(2.2.1, /category/@name)\}$. Since `category` node 2.2 has already been created, the algorithm creates only a new `@name` node with value "wrist watch". □

The complexity of the reconstruction algorithm is quadratic. It takes at most $O(|T_D||M|)$ time, where $|T_D|$ is the size of the datawarehouse and $|M|$ is the size of paths in the mapping rule that populate T_D from $source$ [do Nascimento 2008].

Algorithm 3 Reconstruction of source document

```
1:  function buildSource (D = (T_D, K_D), source)
2:      V(T_s) ⇐ {r}; id(r) ⇐ source;
3:      Leaves ⇐ {n ∈ V(T_D) | n is a leaf node};
4:      for each node n_D in Leaves do
5:          if (id_s, p_s) ∈ n_D.@prov and origin(id_s) = source then
6:              T_s ⇐ createPath(T_s, id_s, p_s, n_D);
7:      return T_s;

8:  function createPath (T, id_s, path_s, node)
9:      let n_0 be the node in T s.t. id(n_0) is the longest prefix of id_s among identifiers of
        nodes in T;
10:     let id(n_0) be j_0.j_1. ... .j_a;
11:     let id_s be j_0. ... .j_a.j_{a+1}. ... .j_{a+m};
12:     let path_s be /l_1/l_2/ ... /l_{a+m};
13:     for i := 1 to m do
14:         create node n_i in T as a child of n_{i-1} with label l_{a+i};
15:         id(n_i) ⇐ j_0. ... .j_a. ... .j_{a+i};
16:     val(n_m) ⇐ val(node);
17:     if n_1 is an element node then
18:         sort list ele(n_0) according to node identifiers;
19:     return T;
```

4. Experimental Results

One of the goals of annotating nodes with provenance information is to enable reconstruction of the source document, which is necessary for incrementally updating the warehouse based on new versions of source data. It is clear that the same functionality can be achieved if the source document were locally stored along with the datawarehouse. The question we would like to answer is how much storage is saved by keeping annotations instead of the original document. That is, we want to quantify the impact of annotations on the storage cost of our model.

We have conducted an experimental study to compare the size of a datawarehouse generated according to the proposed model with the size of a datawarehouse without annotations plus the size of the sources. We denote the first as *annotated DW* and the latter as *DW+source*. Recall that in our model elements are merged whenever they agree on their key values. Thus, the number of mergings has an impact on the size of the warehouse. More specifically, the difference between the size of an annotated DW and DW+source tends to grow with the number of mergings. For preventing this aspect to affect the experimental results, we have chosen to populate annotated DW from a single data source with no restructurings that could cause element mergings. Moreover, we have defined "complete" mappings; that is, every text value in the source is extracted to populated the warehouse. As a result, the size of annotated DW reported by the experiment represents the *worst case scenario*, in which annotations are stored for every text value in the source and no elements are merged.

The algorithms were implemented in Java using JDOM framework [Hunter 2000]. We used DBLP repository [Ley 1997] as our source data, and have ex-

ecuted the experiment with five documents extracted from DBLP with different sizes. Table 1 presents our results. The first column (*Elements*) shows the number of elements in the data source; the second (*Leaf nodes*) contains the amount of leaf nodes in the source; the size of DW+Source and annotated DW are given in the third and fourth column, respectively; the last column shows the amount of storage saved by annotated DW in comparison with DW+source. In our implementation, an annotation of the form $(ident, p)$ is stored with the node identifier $ident$ encoded as a Dewey Order, and path p represented as a *path identifier*, instead of the complete path in its text form.

Elements	Leaf nodes	(1) DW+Source	(2) Annotated DW	% Saved
451.678	403.996	36MB	27MB	25%
902.879	809.626	72MB	53MB	26%
1.309.532	1.176.649	104MB	77MB	26%
1.659.740	1.488.950	132MB	97MB	26%
1.988.661	1.785.833	160MB	117MB	26%

Table 1. Size of source data and datawarehouse.

Our experiment showed that in general the size of annotated DW is 26% smaller than DW+source. It is worth noticing that in all data sets around 90% of the nodes are leaves. Since in our model only leaves are annotated, this fact imposes a bigger impact on the size of *annotated DW*. Observe that 26% is the minimum amount of storage saved by our model, since in real datawarehouses, data can be filtered out and sources may contain overlapping data. The size of annotated DW can be further reduced by compressing node identifiers. A compression mechanism for an encoding similar to the Dewey Order, which can be applied to our model, has been proposed in [O'Neil et al. 2004].

5. Related Work

There are many proposals for specifying keys for XML documents. Languages for schema specification as DTD, and XML Schema also allow keys to be specified. Keys proposed in [Buneman et al. 2002a] are independent of any schema specification, and allows the definition of keys that identify elements in the entire document and in subtrees. The need for defining hierarchical keys in order to univocally identify elements in each level of the tree has been largely recognized as a requirement for propagating updates through views [Davidson and Liefke 2001, Braganholo et al. 2006].

Several schemes for generating persistent XML node identifiers have been proposed in the literature. In [Marian et al. 2001], persistent identifiers XIDs are proposed to describe changes in XML documents. Three labelling schemes have been presented in [Tatarinov et al. 2002]: global order, local order, and Dewey order. We have adopted the Dewey order because it is the only one that enables source reconstruction when only part of the original document is stored in the warehouse.

Annotations for XML documents have been proposed in several domains [Tan 2007, Buneman et al. 2002b, Simmhan et al. 2005, Buneman et al. 2001]. They may carry not only provenance information, but also notes on corrections and errors that the original data may have. Annotations can also be used for archiving, as proposed in [Buneman et al. 2002b]. Although they adopt the same idea of merging nodes based on XML keys, the model assumes that no changes in the structure of the document are made.

To the best of our knowledge, our datawarehouse model is the first to use node identifiers that combine provenance information with the ability to reconstruct the source data even in the presence of filtering and restructuring.

6. Conclusion

The model proposed in this paper is a first step towards a mechanism for automating the update process of a datawarehouse based on new versions of source XML data. To this end, the model annotates values extracted from each data source with information that allows the portion of the source document used to populate the warehouse to be reconstructed. By running a diff algorithm, we can then compute the changes between the reconstructed document and a new version, and generate the update operations. We have conducted an experimental study to determine the impact of the annotations on the size of the datawarehouse. It showed that our model reduces the storage cost in at least 26%. That is, the size of the annotated datawarehouse is at least 26% smaller than the size of the data source and a non-annotated datawarehouse combined. We believe that this result can be further improved by applying a compression mechanism on the Dewey Order Encoding, which is adopted in our model. Investigating alternative encoding and compression mechanisms is one of our future work. Other issues that need to be investigated include: (1) extensions to the mapping language with selection and function calls (such as split and concatenation of strings); (2) an experimental study to determine the performance of the algorithms for populating the warehouse and reconstructing source documents; (3) definition of cleaning procedures and a mechanism to register the sequence of modifications applied to source data; (4) a complete framework for automating the datawarehouse update process.

References

Braganholo, V., Davidson, S., and Heuser, C. (2006). Pataxó: a framework to allow updates through xml views. *ACM TODS*, 31:839–886.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (1998). *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium (W3C). `http://www.w3.org/TR/REC-xml`.

Buneman, P., Davidson, S., Fan, W., Hara, C., and Tan, W.-C. (2002a). Keys for XML. *Computer Networks*, 39(5):473–487.

Buneman, P., Khanna, S., Tajima, K., and W.C., T. (2002b). Archiving scientific data. Technical report.

Buneman, P., Khanna, S., and Tan, W.-C. (2001). Why and where: A characterization of data provenance. In *Proceedings of ICDT'01*.

Clark, J. and DeRose, S. (1999). XML Path Language (XPath). World Wide Web Consortium (W3C). `http://www.w3.org/TR/xpath`.

Davidson, S., Fan, W., and Hara, C. (2007). Propagating XML constraints to relations. *Journal of Computer and System Sciences (JCSS)*, 73(3):316–361.

Davidson, S. and Liefke, H. (2001). Creating and maintaining curated view databases. In *Knwoledge Discovery and Data Mining in Biological Databases*.

do Nascimento, A. M. (2008). Um modelo para integração de documentos XML em nível de instância. Master's thesis, Universidade Federal do Paraná, Brazil.

Draper, D., HaLevy, A. Y., and Weld, D. S. (2001). The nimble xml data integration system. In *Proceedings of the International International Conference on Data Engineering (ICDE)*.

Fallside, D. C. (2000). *XML Schema Part 0: Primer*. World Wide Web Consortium (W3C). http://www.w3.org/TR/xmlschema-0/.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages and Computation*. Addison Wesley.

Hunter, J. (2000). *JDOM*. http://www.jdom.org.

Ley, M. (1997). *XML Schema Part 0: Primer*. Universitat Trier. http://dblp.uni-trier.de.

Lim, E.-P., Srivastava, J., Prabhakar, S., and Richardson, J. (1996). Entity identification in database integration. *Information Sciences*, 89(1).

Marian, A., Abiteboul, S., Cobena, G., and Mignet, L. (2001). Change-centric management of versions in a xml warehouse. In *Proceeddings of VLDB'2001*, pages 581 – 590.

Menestrina, D., Benjelloun, O., and Garcia-Molina, H. (2006). Generic entity resolution with data confidences. In *Proceedings of the International VLDB Workshop on Clean Databases*, Seoul, Korea.

O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., and Westbury, N. (2004). ORDPATHs: Insert-friendly XML node labels. In *Proceedings of SIGMOD'2004*, pages 903–908, Paris, France.

Poggi, A. and Abiteboul, S. (2005). XML data integration with identification. In *Proceedings of International Workshop on Database Programming Languages (DBPL)*.

Pokorný, J. (2002). Xml data warehouse: Modelling and querying. In *Proceedings of the Baltic Conference (BalticDB&IS)*, pages 267 – 280.

Prabhakar, S., Richardson, J., Srivastava, J., and Lim, E.-P. (1993). Instance-level integration in federated autonomous databases. In *Hawaiian Conference for System Science*.

Rahm, E. and Do, H. H. (2000). Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13.

Rundensteiner, E. A., Koeller, A., and Zhang, X. (2000). Maintaining data warehouses over changing information sources. *Communications of the ACM*, 43(6):57–62.

Sawires, A., Tatemura, J., Po, O., Agrawal, D., and Candan, K. S. (2005). Incremental maintenance of path-expression views. In *Proceedings of SIGMOD'2005*, pages 443 – 454, Baltimore, Maryland.

Simmhan, Y. L., Plale, B., and Gannon, D. (2005). A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31=36.

Tan, W.-C. (2007). Provenance in databases: Past, current, and future. *IEEE Data Engineering Bulletin*, 30(4):3–12.

Tatarinov, I., Viglas, S. D., Beyer, K., Shanmugasundaram, J., Shekita, E., and Zhang, C. (2002). Storing and querying ordered XML using a relational database system. In *Proceedings of SIGMOD'2002*, pages 204–215, Madison, Wisconsing, USA.

Xyleme, L. (2001). A dynamic warehouse for xml data of the web. *IEEE Data Engineering Bulletin*, 24(02).