

# Efficient Discovery of XML Data Redundancies <sup>\*</sup>

Cong Yu  
Department of EECS  
University of Michigan  
congy@eecs.umich.edu

H. V. Jagadish  
Department of EECS  
University of Michigan  
jag@eecs.umich.edu

## ABSTRACT

As XML becomes widely used, dealing with redundancies in XML data has become an increasingly important issue. Redundantly stored information can lead not just to a higher data storage cost, but also to increased costs for data transfer and data manipulation. Furthermore, such data redundancies can lead to potential update anomalies, rendering the database inconsistent.

One way to avoid data redundancies is to employ good schema design based on known functional dependencies. In fact, several recent studies have focused on defining the notion of XML Functional Dependencies (XML FDs) to capture XML data redundancies. We observe further that XML databases are often “casually designed” and XML FDs may not be determined in advance. Under such circumstances, discovering XML data redundancies (in terms of FDs) from the data itself becomes necessary and is an integral part of the schema refinement process.

In this paper, we present the design and implementation of the first system, *DiscoverXFD*, for efficient discovery of XML data redundancies. It employs a novel XML data structure and introduces a new class of partition based algorithms. *DiscoverXFD* can not only be used for the previous definitions of XML functional dependencies, but also for a more comprehensive notion we develop in this paper, capable of detecting redundancies involving set elements while maintaining clear semantics. Experimental evaluations using real life and benchmark datasets demonstrate that our system is practical and scales well with increasing data size.

## 1. INTRODUCTION

Redundant data takes up unnecessary storage, inflates data transfer cost, and can lead to update anomalies. A central goal of database design is to ensure that there are no unintended redundancies. As XML databases have become more common, good design of XML schema has become increasingly important, especially in scientific databases with complex structures. Furthermore, one of the benefits of XML (whether intended or not) is the ease of generating XML data: compared with relational data, XML data can be created by ordinary users (e.g. individual scientists) with minimal training in database schema design. Such casual design of XML

schema is likely to lead to many data redundancies in the resulting XML databases<sup>1</sup>. Discovery of redundancies and schematic constraints based on the data will provide the critical first step for analyzing and refining such schemas.

The notion of functional dependency (FD) plays a central role in defining redundancies [8] in relational databases, and should play a correspondingly important role in XML databases as well. Although similar to their relational counterparts, redundancies in XML data have several distinct features due to the heterogeneous nature of XML data. In consequence, standard relational FD discovery algorithms are both inefficient and insufficient to find all XML FDs. (This is true whether we consider classic relational FD discovery algorithms such as [17], or more recent proposals such as *Dep-Miner*[16], *TANE*[13], and *FUN*[20]). In this paper, we develop a new algorithm *DiscoverXFD*, for efficient discovery of XML FDs and data redundancies.

The notion of XML functional dependency (XML FD), and the related notion of XML normal form, have recently become an important research topic. In [3], Arenas and Libkin adopted a tuple based approach and were the first to formally define XML FD and Normal Form. In [14, 24], the authors took a path based approach and built their XML FD notion in a fashion similar to the XML Key notion proposed in [5]. In this paper, we show that these XML FD notions are insufficient, and propose a *generalized tree tuple* based XML FD notion that can fully capture XML data redundancies with unambiguous semantics. Our algorithm, *DiscoverXFD*, can find all XML FDs in accordance with either of the previous definitions, or according to our new, more general, definition.

Consider the example XML document in Figure 1, which maintains information about books sold at various book stores within a book warehouse, grouped by states. Each store records its contact information and the books it is selling, and for each book, the ISBN, author, title, and price are maintained. Two intuitive constraints, which the example satisfies, are the following: two books with the same ISBN must have the same title and the same set of authors; and likewise, two books with the same set of authors and the same title must share the same ISBN. Both constraints cause some information in the XML document to become redundant (e.g., the title *DBMS* and the set of authors *Ramakrishnan* and *Gehrke* are stored multiple times for ISBN 0072465638, and vice versa). One important characteristic that distinguishes such XML redundancies from their relational counterparts is the involvement of set elements: it is the *set of authors*, rather than an individual author, that are being compared and duplicated. This class of FDs is not covered by the definitions in [3] and [24]. A third constraint is equally interesting: for any two books sold at the same store chain (i.e., stores with the same name), if

<sup>\*</sup>Supported in part by NSF under grant IIS-0438909.

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, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '06, September 12-15, 2006, Seoul, Korea.

Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

<sup>1</sup>Anecdotal examples include some large, heavily used community resources, such as PIR [1].

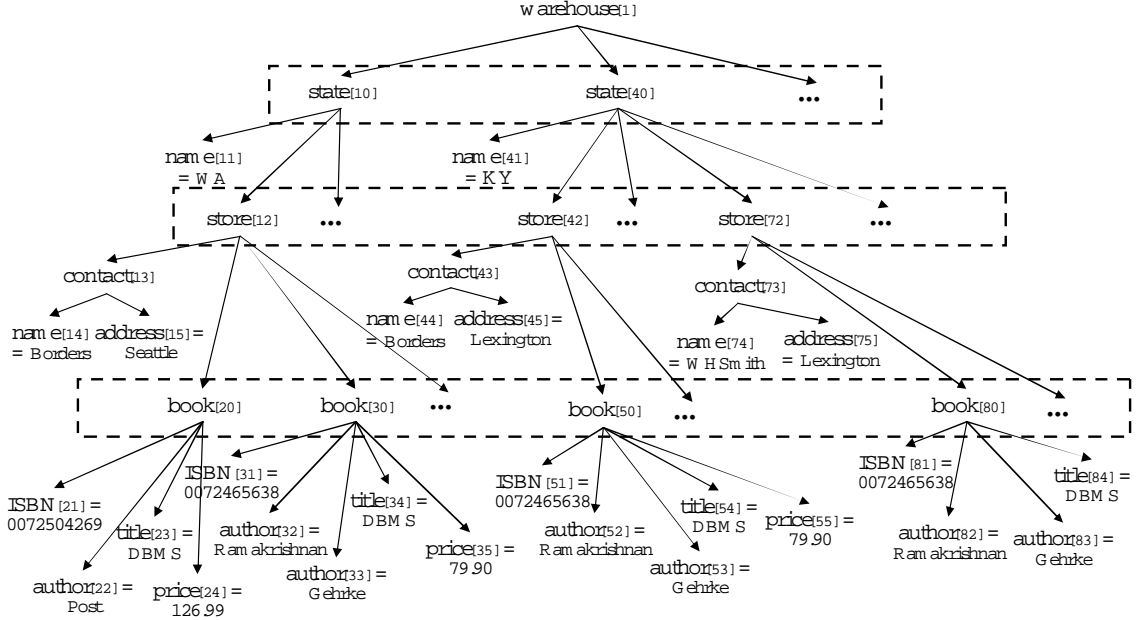


Figure 1: Example XML document (warehouse). Each node is assigned a key in the bracket, and the dashed boxes isolate data elements that correspond to complex set elements in the schema (Figure 2).

they have the same ISBN, they will be sold at the same price. The price 79.90 of book 0072465638, therefore, is stored redundantly for the store chain Borders (one in Seattle and the other in Lexington). Such a redundancy is special in that while it is the books on which the comparison is specified, the constraint actually involves an element (i.e., store name) that is not a descendant of book.

**Main contributions and paper outline:** We make the following main contributions: I. We study the examples of redundancy-indicating constraints in the XML data model (Section 2.2) and show that existing XML FD notions are insufficient for capturing certain XML data redundancies (Section 2.3). II. We propose a new *generalized tree tuple* based XML FD notion, which improves upon the notion introduced in [3]. We show that more XML data redundancies can now be effectively captured by *interesting XML FDs* (Section 3). III. We design and implement the DiscoverXFD system, which employs a new XML data structure and several novel partition-based algorithms that can efficiently discover XML FDs and detect XML data redundancies (Section 4); IV. We demonstrate the scalability and practicality of DiscoverXFD using a benchmark dataset and a variety of real life datasets (Section 5). We first present some necessary background.

## 2. BACKGROUND AND CHALLENGES

### 2.1 Schema and Data Tree

Figure 2 illustrates the schema of the example XML document in Figure 1. It is shown in a nested relational representation [22] that is used as a common data model to represent both relational and hierarchical (XML) schemas. Intuitively, keyword *Rcd* is used to indicate *complex schema elements* (i.e., elements that have child elements, e.g., *contact*), and keyword *SetOf* is used to indicate *set schema elements* (i.e., elements that can have multiple matching data elements sharing the same parent in the data, e.g., *book*). A set element is not necessarily complex: e.g., *author* is a set element with a domain of string (*str*). Formally:

DEFINITION 1 (SCHEMA). A schema is defined to be  $S = \langle E, T, r \rangle$ , where:

- $E$  is a finite set of element labels;
- $T$  is a finite set of element types, and each  $e \in E$  is associated with a  $\tau \in T$ , written as  $(e : \tau)$ ,  $\tau$  has the form:
 
$$\tau ::= \text{str} \mid \text{int} \mid \text{float} \mid \text{SetOf } \tau \mid \text{Rcd}[e_1 : \tau_1, \dots, e_n : \tau_n] \mid \text{Choice}[e_1 : \tau_1, \dots, e_n : \tau_n];$$
- $r \in E$  is the label of the root element, whose associated element type can not be *SetOf*  $\tau$ .

Definition 1 corresponds to the “core” constructs in XML Schema [25]. Types *str*, *int*, and *float* are system defined *simple* types. *Rcd* and *Choice* are *complex* types representing the “all” and “choice” model-groups in XML, respectively<sup>2</sup>. Type *SetOf* is the *set* type associated with elements with *maxOccurs* greater than one in XML Schema. We ignore element order and represent the “sequence” model-group as the *Rcd* type. For simplicity, we treat attributes and elements in the same way, with a reserved “@” symbol to indicate attributes. For mixed-content elements, if there is exactly one textual value, we store it under a distinct new attribute “@value.” Otherwise, we ignore the textual values and treat the mixed-content elements as regular *complex* elements.

A schema element  $e_k$  can be identified through a *path* expression,  $\text{path}(e_k) = /e_1/e_2/\dots/e_k$ , where  $e_1 = r$ , and  $e_i$  is associated with type  $\tau_i ::= \text{Rcd}[\dots, e_{i+1} : \tau_{i+1}, \dots]$  for all  $i \in [1, k - 1]$ . Furthermore, if  $e_k$  is a set element, we call  $\text{path}(e_k)$  a *repeatable path*. We do not consider  $\text{path}(e_k)$  to be a repeatable path if  $e_k$  is not a set element, even if some  $e_i (i < k)$  is a set element. For convenience, we adopt XPath steps “.” (self) and “..” (parent) to form *relative paths* with regard to a given path. For example, if the given path is  $/warehouse/state/store$ , the relative path  $../name$  has the absolute path  $/warehouse/state/name$ .

DEFINITION 2 (DATA TREE). An XML database is defined to be a rooted labeled tree  $T = \langle N, \mathcal{P}, \mathcal{V}, n_r \rangle$ , where:

- $N$  is a set of labeled data nodes, each  $n \in N$  has a label  $e$

<sup>2</sup>While the definitions and algorithms throughout the rest of paper handle both types, we largely omit the “choice” type for the simplicity of discussion.

```

warehouse: Rcd
state: SetOf Rcd
name: str
store: SetOf Rcd
contact: Rcd
  name: str
  address: str
book: SetOf Rcd
  isbn: str
  author: SetOf str
  title: str
  price: str

```

Figure 2: Example schema for Figure 1.

and a node key that uniquely identify it in  $T$ ;

- $n_r \in N$  is the root node;
- $\mathcal{P}$  is a set of parent-child edges, there is exactly one  $p = (n', n)$  in  $\mathcal{P}$  for each  $n \in N$  (except  $n_r$ ), where  $n' \in N, n \neq n', n'$  is called the parent node,  $n$  is called the child node;
- $\mathcal{V}$  is a set of value assignments, there is exactly one  $v = (n, s)$  in  $\mathcal{V}$  for each leaf node  $n \in N$ , where  $s$  is a value of simple type.

In Figure 1, node keys are assigned in pre-order traversal (gaps in the numbering indicate omitted elements), and we use “@key” to refer to the node keys. Parent-child edges are represented as directed lines between two data nodes (with arrow pointing to the child node). Value assignments are represented as equality between the node label and the value. We adopt the notion of conformance as defined in [25] and assume that all given data trees conform to their schemas.

A node (or data element)  $n_k$  is a *descendant* of another node  $n_1$  if there exists a series of nodes  $n_i$ , such that  $(n_i, n_{i+1}) \in \mathcal{P}$  for all  $i \in [1, k-1]$ . Similar to schema elements,  $n_k$  can also be addressed using a *path* expression,  $path(n_k) = /e_1/.../e_k$ , where  $e_i$  are labels of  $n_i$  for all  $i \in [1, k]$ ,  $n_1 = n_r$ , and  $(n_i, n_{i+1}) \in \mathcal{P}$  for all  $i \in [1, k-1]$ . Clearly, it is possible that two distinct data nodes will have the same path (e.g., node 11 and node 41). Furthermore,  $n_k$  is called *repeatable* if  $e_k$  corresponds to a set element in the schema. Finally,  $n_k$  is called a *direct descendant* of node  $n_a$ , if  $n_k$  is a descendant of  $n_a$ ,  $path(n_k) = .../e_a/e_1/.../e_{k-1}/e_k$ , and  $e_i$  is not a set element for any  $i \in [1, k-1]$ . For example, node 21 (ISBN) is a direct descendant of node 20 (book), but not node 12 (store).

In considering data redundancy, it is also important to determine the equality between the “values” associated with two data nodes (as proposed in [5]), instead of comparing their “identities” (as represented by @key):

**DEFINITION 3 (NODE-VALUE EQUALITY).** *Two data nodes  $n_1$  of  $T_1 = \langle N_1, \mathcal{P}_1, \mathcal{V}_1, n_{r1} \rangle$ , and  $n_2$  of  $T_2 = \langle N_2, \mathcal{P}_2, \mathcal{V}_2, n_{r2} \rangle$  are node-value equal (written as  $n_1 =_{nv} n_2$ ) iff:*

- $n_1$  and  $n_2$  both exist and have the same label;
- There exists a set  $M$  of matching pairs: each pair  $m = (n'_1, n'_2)$  indicates that  $n'_1 =_{nv} n'_2$ , where  $n'_1, n'_2$  are child nodes of  $n_1, n_2$ , respectively. All child nodes of  $n_1$  ( $n_2$ ) participate in  $M$  and each child node participates in only one such pair.
- $(n_1, s) \in \mathcal{V}_1$  iff  $(n_2, s) \in \mathcal{V}_2$ , where  $s$  is a simple value.

Intuitively, two data nodes (e.g., node 30 and 50) are node-value equal iff the two subtrees rooted at the two nodes are identical without considering the order among sibling nodes. Because data nodes can also be addressed by paths, we define the path-value equality based on Definition 3:

**DEFINITION 4 (PATH-VALUE EQUALITY).** *Given paths  $p_1$  on  $T_1 = \langle N_1, \mathcal{P}_1, \mathcal{V}_1, n_{r1} \rangle$ , and  $p_2$  on  $T_2 = \langle N_2, \mathcal{P}_2, \mathcal{V}_2, n_{r2} \rangle$ ,  $p_1, p_2$  are path-value equal (written as  $T_1.p_1 =_{pv} T_2.p_2$ ) iff:*

for  $T_1$ , for each  $n_1, n_1 \in N_1, path(n_1) = p_1$ , there exists corresponding  $n_2, n_2 \in N_2, path(n_2) = p_2, n_1 =_{nv} n_2$ , vice versa for  $T_2$ , and the correspondence is one-on-one.

Value equality between two paths is complicated by the fact that a single path can match multiple nodes in a data tree. Definition 4 requires that, for two paths to be considered value equal, each node that is pointed to by one path must have a corresponding node that is pointed to by the other path, where the two nodes are node-value equal.

## 2.2 Example XML Data Redundancies

We now illustrate data redundancies that can be caused by constraints on the XML data and describe the features of those redundancy-indicating constraints. All the examples are based on the data tree in Figure 1.

**CONSTRAINT 1.** *Whenever two books (e.g., nodes 30 and 50) agree on their ISBN values, they will have the same title.*

It is clear that Constraint 1 leads to redundancies if there are two distinct books in the data with the same ISBN value: their titles are redundantly stored. Intuitively, such XML constraints consist of three components. First, *target elements*, which is the set of data elements (e.g., the books) on which the constraints are imposed. Second, *condition elements*, which are the elements (e.g., ISBN) specified in the condition of the constraint. Third, *implication elements*, which are the elements (e.g., title) whose equality is implied if the condition is met. It is worth noting that not all constraints correspond to redundancies. For example, if each distinct book in the data has a unique ISBN value, then Constraint 1 will not result in any redundancy. We will explore the properties of redundancy-indicating constraints later in Section 3.3.

Constraint 1 is straight-forward because both ISBN and title are subelements of book, and each book has exactly one ISBN and one title. However, constraints on XML data can become more complicated. Consider:

**CONSTRAINT 2.** *Whenever two books are on sale at stores with the same name, if they agree on their ISBN values, they will have the same price.*

Again, Constraint 2 indicates redundancies if there exist two distinct books that share the same ISBN value and that are being sold at the same store or at two stores with the same name. More importantly, Constraint 2 illustrates two important features for XML constraints. First, constraints can involve elements from *multiple hierarchies*. In this case, while the target elements are the set of books, the condition elements include not only a descendant element of book (i.e., title), but also a store name element that is neither ancestor nor descendant of book. Second, constraints can involve *missing elements*. Often, either the condition elements or the implication elements can be missing in the data instances. For example, the price of the book node 80 is not recorded. The following constraints illustrate yet another important feature of XML constraints:

**CONSTRAINT 3.** *Whenever two books agree on their ISBN values, they have the same set of authors.*

**CONSTRAINT 4.** *Whenever two books share the same set of authors and the same title, they agree on their ISBN values.*

Constraints 3 and 4 indicate redundancies if there are two distinct books (e.g., book nodes 30 and 50) in the data with either the same ISBN values, or the same title values and the same set of author values. Most importantly, it is not any individual author, but rather the set of authors, that are being compared or redundantly stored because each book has

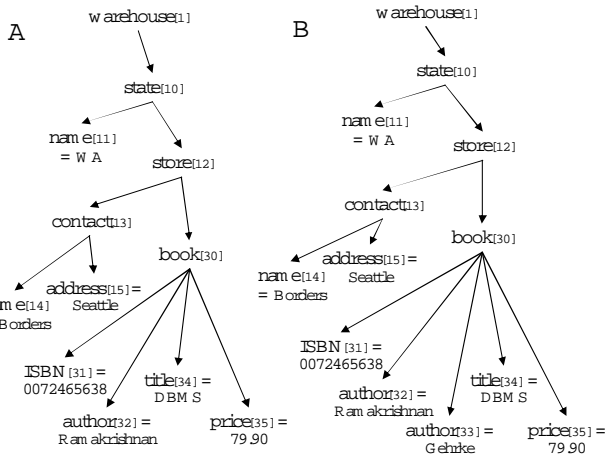


Figure 3: (A) Original tree tuple example, and (B) Generalized tree tuple example (book 30 is the pivot node).

a set of authors. The third important feature of XML constraints, therefore, is the involvement of *set elements*: that each condition or implication element specification can, and often, resolve to a set of elements.

## 2.3 Previous Proposals

The above constraints are essentially intuitive forms of functional dependencies (FDs). To capture redundancies indicated by those constraints, formal definitions of XML FD have been proposed and follow two main approaches: path based approach and tuple based approach. They differ in how the target elements of the constraint are specified: the former implicitly encodes the target elements inside the FD specification, while the latter specifies the target elements independent of each individual FD specification.

**Path based approach:** Proposed by Vincent et al, [24] is representative of the path based approach. An XML FD is of the form:  $\{P_{x_1}, \dots, P_{x_n}\} \rightarrow P_y$ , where  $P_{x_i}$  (also called LHS) are the paths specifying the condition elements,  $P_y$  (also called RHS) is the path specifying the implication element, and the target elements are implicitly specified as the set of elements pointed to by  $P_y$ . For example, Constraint 1 can be expressed as:  $\{\text{/warehouse/ state/store/book/ ISBN}\} \rightarrow \text{/warehouse/ state/store/ book/ title}$ . Without going into details, the semantics of the FD are intuitively defined as the following: for any two distinct title nodes in the data tree, if the ISBN nodes they are associated with have the same value, then the title nodes themselves have the same value. A title node and an ISBN node are associated if they are the descendants of the same book node (book is chosen because its path is the longest common prefix of both title and ISBN). For example, the FD is satisfied in Figure 1 because for any two titles (e.g., title 34 and 54), if their associated ISBNs (e.g., ISBN 31 and 51, respectively) share the same value, they have the same value as well.

**Tuple based approach:** In [3], Arenas and Libkin proposed the first formal XML FD notion based on the concept of *tree tuples*. Instead of specifying target elements from each individual FD as in [24], a set of tree tuples is defined independent of any FD and serves as the target for all FDs. Each tree tuple is a tree constructed by picking exactly one data node from the original data tree for each schema element and projecting away all the other nodes. Figure 3(A) illustrates one such tree tuple. XML FDs are subsequently defined based on

this tree tuple notion and take a form similar to the one in the path based approach. For example, Constraint 2 can be expressed as:  $\{\text{/warehouse/ state/store/contact/ name, /warehouse/ state/store/book/ ISBN}\} \rightarrow \text{/warehouse/ state/store/book/ price}$ . The semantics of the FD is defined as the following: for any two tree tuples, if they have the same values at the nodes specified in the LHS of the FD (i.e., the name and ISBN nodes), they will share the same values at their RHS nodes (i.e., the price nodes). It is worth noting that, if the original XML data tree is viewed as a set of nested relations [4], the set of tree tuples is conceptually equivalent to the set of fully unnested tuples. Compared with path based approach, tuple based XML FD notion has a semantics that is closer to the relational FD notion. It also suggests a natural technique for XML FD discovery: one can convert the XML data into a fully unnested relation and apply existing FD discovery algorithms directly.

**Discussion:** Both [24] and [3] effectively capture multi-hierarchical constraints like Constraint 2. In the former, elements from different hierarchies are associated with each other through the common ancestor node. In the latter, they are connected by belonging to the same tree tuple. Both proposals also adopt a similar semantics for missing elements, which roughly corresponds to the strong satisfaction of FD over incomplete relations as defined in [4].

However, neither notion can effectively capture constraints with set elements. Consider Constraint 3 for Figure 1. The closest form to which it can be expressed under both [24] and [3] is the following:  $\{\text{/warehouse/ state/store/book/ ISBN}\} \rightarrow \text{/warehouse/ state/store/book/ author}$ . It is not difficult to see that the semantics of this FD under either notion are not the same as the semantics of the original constraint. The semantics under [24] are that for any two authors, if they are associated with the same ISBN value, their values are the same. Under this semantics, the FD is violated since book 30 has two authors of different values and the two authors are clearly associated with the same ISBN value. The semantics under [3] are that for any two tree tuples, if their ISBN nodes share the same value, then they have the same value for their author nodes. According to the construction of tree tuple, author 32 and author 33 belong to two different tree tuples. Since the ISBN nodes of the two tuples have the same value while the author nodes of the two tuples differ, the FD is again violated. The original constraint, however, is satisfied in Figure 1: two books with the same ISBN value always have the same *set* of authors. In the next section, we proposed *generalized tree tuple* based XML FD notion that overcomes the semantic limitations the previous notions have in expressing constraints with set elements.

## 3. CAPTURING XML DATA REDUNDANCY

While both tuple and path based approaches are valid ways for defining XML FDs, the tuple based approach has a clearer semantics and is conceptually similar to the relational FD notion. As such, we choose to follow the tuple based approach. In Section 3.1, we introduce the notion of *generalized tree tuple (GTT)*, which improves upon the tree tuple notion in [3]. Based on this new tuple notion, we define the GTT-Based XML FD, as well as the related XML Key. In Section 3.2, we analyze the general form of XML FDs and show that only a subset of such XML FDs are considered *interesting*. Finally, in Section 3.3, XML data redundancy is defined formally.

### 3.1 GTT-Based XML FD

DEFINITION 5 (GENERALIZED TREE TUPLE). A generalized tree tuple of data tree  $T = \langle N, \mathcal{P}, \mathcal{V}, n_r \rangle$ , with regard to a particular data node  $n_p$  (called pivot node), is a tree  $t_{n_p}^T = \langle N^t, \mathcal{P}^t, \mathcal{V}^t, n_r \rangle$ , where:

- $N^t \subseteq N$  is the set of nodes,  $n_p \in N^t$ ;
- $\mathcal{P}^t \subseteq \mathcal{P}$  is the set of parent-child edges;
- $\mathcal{V}^t \subseteq \mathcal{V}$  is the set of value assignments;
- $n_r$  is the same root node in both  $t_{n_p}^T$  and  $T$ ;
- $n \in N^t$  iff 1)  $n$  is a descendant or ancestor of  $n_p$  in  $T$ , or 2)  $n$  is a non-repeatable direct descendant of an ancestor of  $n_p$  in  $T$ ;
- $(n_1, n_2) \in \mathcal{P}^t$  iff  $n_1 \in N^t, n_2 \in N^t, (n_1, n_2) \in \mathcal{P}$ ;
- $(n, s) \in \mathcal{V}^t$  iff  $n \in N^t, (n, s) \in \mathcal{V}$ .

Similar to an original tree tuple, a generalized tree tuple is a data tree projected from the original data tree. However, instead of separating sibling nodes with the same path at all hierarchy levels, a generalized tree tuple has an extra parameter called a *pivot* node, and the separation is done only at subtrees rooted above the pivot node. As a result, ancestor and descendant nodes of the pivot node, as well as all the non-repeatable direct descendant nodes (previously defined in Section 2.1) of those ancestor nodes, are preserved in the tuple. Figure 3(B) illustrates one such generalized tree tuple with node 30 as the pivot node. Note that both author nodes of the book are preserved in the tuple, while in Figure 3(A), only one is kept. Based on the pivot node, we can categorize all generalized tree tuples into tuple classes:

DEFINITION 6 (TUPLE CLASS). A tuple class  $C_p^T$  of the data tree  $T$  is the set of all generalized tree tuples  $t_n^T$ , where  $\text{path}(n) = p$ . Path  $p$  is called the *pivot path*.

For example, the generalized tree tuple in Figure 3(B) belongs to the tuple class  $C_{\text{warehouse/state/store/book}}^T$ .<sup>3</sup> Finally, we introduce the notion of XML FD based on tuple class:

DEFINITION 7 (XML FD). An XML FD is a triple  $\langle C_p, LHS, RHS \rangle$ , often written as  $\{P_{i_1}, P_{i_2}, \dots, P_{i_n}\} \rightarrow P_r$  w.r.t.  $C_p$ , where  $C_p$  denotes a tuple class,  $LHS$  is a set of paths  $(P_{i_i}, i = [1, n])$  relative to  $p$ , and  $RHS$  is a single path  $(P_r)$  relative to  $p$ .

An XML FD holds on a data tree  $T$  (or  $T$  satisfies an XML FD) iff for any two generalized tree tuples  $t_1, t_2 \in C_p$ :

- $\exists i \in [1, n], t_1.P_{i_i} = \perp$  or  $t_2.P_{i_i} = \perp$ , or
- If  $\forall i \in [1, n], t_1.P_{i_i} =_{pv} t_2.P_{i_i}$ , then  $t_1.P_r \neq \perp, t_2.P_r \neq \perp, t_1.P_r =_{pv} t_2.P_r$ . A null value,  $\perp$ , results from a path that matches no node in the tuple, and  $=_{pv}$  is the path-value equality defined in Definition 4.

Because generalized tree tuples can be defined at any hierarchy level, with an appropriate tuple class specification, this new XML FD notion can effectively capture constraints involving set elements. For example, Constraints 3 and 4 can now be expressed as:

FD 3:  $\{./ISBN\} \rightarrow ./author$  w.r.t.  $C_{\text{book}}$

FD 4:  $\{./author, ./title\} \rightarrow ./ISBN$  w.r.t.  $C_{\text{book}}$

with the expected semantics. And the other two example constraints (Constraints 1 and 2) can be expressed as:

FD 1:  $\{./ISBN\} \rightarrow ./title$  w.r.t.  $C_{\text{book}}$

FD 2:  $\{./contact/name, ./ISBN\} \rightarrow ./price$   
w.r.t.  $C_{\text{book}}$ .

**Remarks:** First, missing elements (regarded as being null values) are treated in the same way as in [24], where they are

<sup>3</sup>The superscript is often omitted for brevity. The same for the subscript, which in this case can be abbreviated as *book*.

considered as different from each other and from all other existing elements (i.e., each FD must be strongly satisfied [4]). Second, we note that if we limit tuple classes to only those with pivot paths corresponding to leaf level elements and consolidate all tuple classes into one class, this notion has the same expressive power as the one proposed in [3]. Third, FDs involving set elements only on the RHS can also be captured by incorporating multivalued dependencies (MVD) [10] into the previous tuple based approach. However, in general, FDs involving set elements cannot be captured using MVD. For example, FD 4 can not be expressed using MVD because the set of author values must be considered together. Fourth, previous XML FD notions only consider one element at a time, and hence do not consider order between elements. Since we consider sets of elements, we could consider order between siblings, and treat each collection as a list rather than a set. But then we would miss redundancies where the element order was changed, something we believe is a common occurrence. As such, we have chosen to treat our collections as unordered sets, and to ignore order in XML. The impact of considering order in our system is discussed in Section 4.5.

When the RHS of an XML FD is  $./@key$ , the LHS then uniquely identifies each tuple in  $C_p$  because the pivot node (and hence its key) for each tuple is unique. This naturally leads us to the following XML Key notion:

DEFINITION 8 (XML KEY). An XML Key of a data tree  $T$  is a pair  $\langle C_p, LHS \rangle$ , where  $T$  satisfies the XML FD  $\langle C_p, LHS, ./@key \rangle$ .

This new notion of XML Key shares many similarities with the notion proposed by Buneman et al in [5], which contains a target path (which identifies a set of nodes) and a set of key paths (which uniquely identifies each node in the aforementioned set). In fact, our notion extends their notion by allowing the key paths to be arbitrarily relative to the target path while ensuring the semantics is still valid.

## 3.2 Interesting XML FD

The range of XML FDs expressible under the new notion are quite broad. However, not all expressible FDs are of interest. For example, some FDs may not be interesting because they are trivial or redundant with other FDs.

### 3.2.1 Trivial XML FDs

DEFINITION 9 (TRIVIAL XML FD). An XML FD  $\langle C_p, LHS, RHS \rangle$  is trivial if 1)  $RHS \in LHS$ , or 2) for any generalized tree tuple in  $C_p$ , there is at least one path in  $LHS$  that matches no data node.

The definition of trivial XML FDs partly follows the relational semantics, where an FD is trivial if the LHS contains the RHS, and partly follows the strong satisfaction semantics, where an FD is trivial if the LHS always contains at least one null value. Such a situation can arise, as mentioned in [3], because of the existence of Choice elements. For example, if contact is a Choice element instead of a Rcd element (i.e., it can have either name or address as its child, but not both) in Figure 2, then the XML FD  $\{./contact/name, ./contact/address\} \rightarrow ./@key$  w.r.t.  $C_{\text{store}}$  is trivial since no  $C_{\text{store}}$  tuple will have both LHS nodes.

### 3.2.2 Essential Tuple Classes

THEOREM 1. Given a tuple class  $C_p$ , if  $p$  is not a repeatable path (see Section 2.1), and there exists tuple class  $C_{p'}$ , where













