

RRXS: Redundancy reducing XML storage in relations

Yi Chen¹, Susan Davidson¹, Carmem Hara², and Yifeng Zheng¹
(yicn@cis.upenn.edu, susan@cis.upenn.edu, carmem@inf.ufpr.br, yifeng@cis.upenn.edu)

¹University of Pennsylvania *

²Universidade Federal do Parana, Brazil

Abstract

Current techniques for storing XML using relational technology consider the structure of an XML document but ignore its semantics as expressed by keys or functional dependencies. However, when the semantics of a document are considered redundancy may be reduced, node identifiers removed where value-based keys are available, and semantic constraints validated using relational primary key technology.

In this paper, we propose a novel constraint definition called XFDs that capture structural as well as semantic information. We present a set of rewriting rules for XFDs, and use them to design a polynomial time algorithm which, given an input set of XFDs, computes a reduced set of XFDs. Based on this algorithm, we present a redundancy removing storage mapping from XML to relations called RRXS. The effectiveness of the mapping is demonstrated by experiments on three data sets.

1 Introduction

Over the past several years, formalisms for capturing structural and semantic constraints in XML have been developed. The earliest examples of these were DTDs and schema graphs [20, 14], which capture structural constraints. More recently, keys [7], foreign keys [21] and functional dependencies [3] have been proposed to capture semantic constraints, and various aspects of these proposals have found their way into XMLSchema [21].

There has also been tremendous interest in using relational databases to store XML documents [11, 13, 20, 6], thus leveraging a well-developed technology for data

Research supported by NSF DBI-9975206, NSF IIS-9977408.

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 29th VLDB Conference,
Berlin, Germany, 2003

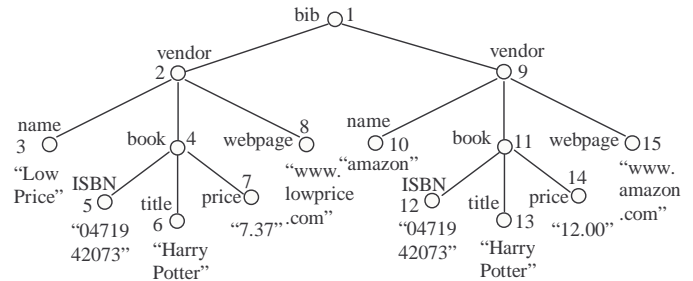


Figure 1: Book vendors document

management and query processing. While most of these proposals use structural constraints to produce the relational mapping, they ignore semantic constraints. The result is that validating updates in the relational representation of an XML document can be very inefficient, since semantic constraints must be checked using stored procedures. In contrast, [9] takes into account KEY and KEYREF constraints [21], and maps these XMLSchema semantic constraints to primary keys and foreign keys in “key relations” in the relational mapping. In this way, XML key and keyref constraints can be efficiently enforced using relational primary and foreign keys. However, it does not consider other types of semantic constraints in XML, such as functional dependencies.

In this paper, we consider the problem of providing a mapping from XML to a relational database taking structural as well as a broad class of semantic constraints into account. As an example, consider the XML tree representation of a book vendors document shown in figure 1. Given this document and our understanding of its semantics, we may wish to state the following constraints:

- C_1 : Each vendor has exactly one name and webpage.
- C_2 : Each vendor is identified by its name.
- C_3 : Each book has exactly one ISBN, title and price.
- C_4 : If two books have the same ISBN, they must have the same title.
- C_5 : Each book sold by a particular vendor is identified by its ISBN.

```

<!ELEMENT bib(vendor*)>
<!ELEMENT vendor(name, webpage, book*)>
<!ELEMENT book(ISBN, title, price)>
<!ELEMENT name(#PCDATA)>
<!ELEMENT webpage(#PCDATA)>
<!ELEMENT ISBN(#PCDATA)>
<!ELEMENT title(#PCDATA)>
<!ELEMENT price(#PCDATA)>

```

Figure 2: Book vendors DTD

The first and third constraints are structural constraints, as captured by the DTD of figure 2. The second constraint is an example of an *absolute* key, where the key is defined over the whole document, and the fifth is an example of a *relative* key, where the key is defined within some context, in the terminology of [7, 21]. The fourth constraint is an example of a functional dependency [3], and cannot be expressed as a key constraint.

Given this structural and semantic information, a good relational design for this document would be *Vendor(name, webpage)*, with $\{name\}$ as primary key; *Book(ISBN, title)*, with $\{ISBN\}$ as primary key; and *Sells(name, ISBN, price)*, with $\{name, ISBN\}$ as primary key.

However, none of the relational storage strategies designed to date would produce this design. For example, the hybrid inlining strategy of [20] would produce *Vendor(ID, name, webpage)*, with $\{ID\}$ as primary key; and *Book(parent-ID, ID, ISBN, title, price)*, with $\{ID\}$ as primary key. Here, *ID* can be thought of as the XML node identifier (id), and *parent-ID* as the node id of the parent node. Similarly, the strategy of [9] would produce *Vendor(VID, name, webpage)*, with $\{VID\}$ and $\{name\}$ as two keys, *Book(VID, ISBN, BID, title, price)*, with $\{VID, ISBN\}$ and $\{BID\}$ as two keys. Although information about XML keys has been captured as primary keys in the relational design, internal node ids are still used and the redundancy associated with book titles has not been eliminated. Furthermore, these two approaches fail to separate the entity *Book* from the relationship *Sell* between entities *Book* and *Vendor*.

The strategy adopted in this paper, RRXS, is to produce a relational design which preserves structural and semantic constraints of the XML data while reducing redundancies. First, we interpret structural constraints, keys and user-defined functional dependencies as special forms of *XML functional dependencies* (XFDs). Using a sound set of rewrite rules for XFDs, we then remove redundant XFDs. Finally, by mapping paths in XFDs to relational attributes we get a set of relational functional dependencies and produce a relational storage for the XML data which preserves the content and structure information of the original XML document (up to order), removes redundancy as indicated by the XFDs, reduces the use of internal node IDs, and allows XFDs to be efficiently en-

forced using relational primary key constraints.

The contributions and outline of this work are:

1. A novel constraint definition, XFDs, that can capture structural constraints, key constraints [7, 21], as well as the functional dependencies of [3] is introduced in section 2;
2. A set of rewriting rules for XFDs is presented in section 3;
3. A polynomial time algorithm to reduce the input set of XFDs is proposed in section 3;
4. A constraint-preserving mapping into relational storage that preserves information and reduces redundancy is discussed in section 4: and
5. Experimental evaluation which shows the effectiveness of RRXS using three real data sets with different features is presented in section 5.

We close by reviewing related work and discussing directions for future research.

2 Functional Dependencies for XML

As in relational databases, functional dependencies for XML (XFDs) are used to describe the property that the values of some attributes of a tuple uniquely determine the values of other attributes of the tuple [1]. The difference lies in that attributes and tuples are basic units in relational databases, whereas in XML data, they must be defined using path expressions.

After giving the definition of paths and XFDs, we show how they can be used to capture structural as well as semantic constraints.

2.1 Paths and XFDs

We adopt an unordered tree model for XML data, where the leaves can be either an element node which has a text value, or an attribute node.

The path language used in XFDs for XML tree navigation allows traversal along the child (/) and descendant (//) axis. In addition, variables can be used to bind to path expressions, and can appear at the beginning of path expressions. Our path language, $XP^{\{/./\}}$, is defined by the following grammar:

$$PL_1 \rightarrow l|PL_1/PL_1|PL_1//PL_1$$

$$PL_2 \rightarrow \epsilon|/PL_1|//PL_1$$

$$PL \rightarrow PL_2|PL_2/value()$$

$$XP^{\{/./\}} \rightarrow PL|\$x/PL$$

where l denotes an XML node label (element tag or attribute name), ϵ denotes the empty path, $value()$ retrieves the value of the context node (only applicable for

a leaf node), and $\$x$ is a variable bound to a path expression in PL .

We call path expressions without variable bindings *variable-free path expressions*. A variable-free path expression is a *simple path* if it does not contain “//”.

Definition 2.1: An XFD ϕ is of the following form:

$$\begin{aligned} & \$v_1 \text{ in } P_1 \{, \$v_2 \text{ in } \$v_1/P_2\} \\ & \$v_{f(1)}/Q_1, \dots, \$v_{f(n)}/Q_n \rightarrow \\ & \$v_{f(n+1)}/Q_{n+1} \dots, \$v_{f(n+m)}/Q_{n+m} \end{aligned}$$

where $f(k) \in \{1, 2\}, 1 \leq k \leq n + m$. Here, the *dependent* variable $\$v_2$ is optional (indicated by curly braces), in which case $f(k) \in \{1\}$. P_1 , and P_2 are expressions in PL , whereas the Q_i 's are simple paths. ■

Two variables are necessary to model relative keys [7]. We limit the number of variables to avoid complexity.

For example, using the variable bindings $\$x$ in //vendor, $\$y$ in //book, $\$z$ in $\$x$ /book, the constraints from the introduction would be expressed as follows:

$$\begin{aligned} C_1: & \$x \rightarrow \$x/\text{name}, \$x/\text{webpage} \\ C_2: & \$x/\text{name}/\text{value}() \rightarrow \$x \\ C_3: & \$y \rightarrow \$y/\text{ISBN}, \$y/\text{title}, \$y/\text{price} \\ C_4: & \$y/\text{ISBN}/\text{value}() \rightarrow \$y/\text{title}/\text{value}() \\ C_5: & \$x, \$z/\text{ISBN}/\text{value}() \rightarrow \$z \end{aligned}$$

Note that we have simplified the example by presenting all variable definitions first, but notice that each XFD involves at most two variables.

An “attribute” for XML, called a *P-attribute*, is defined by a path expression $\$v/Q$ that occurs on the left-hand or right-hand side of some functional dependency. The value of a P-attribute is the set of nodes or values in an XML instance which are obtained by evaluating the simple path Q with respect to the node to which the variable $\$v$ is bound (denoted $\$v[[Q]]$). For example, referring to figure 1, if $\$x$ is bound to node 2, then $\$x[[/\text{name}/\text{value}()]] = \{\text{“LowPrice”}\}$.

The set of P-attributes in an XFD group together values to form a “tuple” for an XML instance, named an *X-tuple*. For example, for C_2 , if $\$x = 2$ then the tuple corresponding to $\{\$x/\text{name}/\text{value}(), \$x\}$ would be (“LowPrice”, 2).

Given an XML instance, the set of all valid variable bindings for an XFD is defined as follows: $\{(\$v_1, \$v_2) \mid \$v_1 \in [[P_1]], \$v_2 \in \$v_1[[P_2]]\}$, where $[[P_1]]$ denotes that P_1 is evaluated with respect to the root of the document. If the XFD has single variable then v_2 is omitted. For example, referring to figure 1, the set of all valid variable bindings for C_5 is $\{(2,4), (9, 11)\}$, while those for C_4 are $\{(4), (11)\}$.

A functional dependency is defined on the P-attributes of an X-tuple, and intuitively must hold on the set of all X-tuples formed by valid variable bindings.

Note that P-attributes are “expandable” in the sense that all variables can be removed. For example, given the previous definitions of $\$x$ and $\$z$ we can expand path expression $\$z/\text{ISBN}$ as follows:

$$\$z/\text{ISBN} = \$x/\text{book}/\text{ISBN} = //\text{vendor}/\text{book}.$$

Definition 2.2: Given a P-attribute p , $\text{expand}(p)$ is defined as the path expression obtained by repeatedly substituting variables with the path expressions they are bound to. ■

To define what it means for an XFD to hold in an XML instance, we test P-attribute equality “=” with respect to the path expression it is bound to. Specifically, we use *node equality* when the P-attribute ends with a label and *value equality* when the P-attribute ends with “value()”. First, we introduce the following notation:

Definition 2.3: For $p \in PL$, $\text{NodePath}(p) = q$, if $p = q/\text{value}()$; and p otherwise. ■

Definition 2.4: Given an XML instance T and an XFD ϕ , ϕ holds in T if and only if

- (1) For every valid variable binding (x_1, x_2) for ϕ , $|x_{f(k)}[[\text{NodePath}(Q_k)]]| = 1, 1 \leq k \leq n + m$; and
- (2) For any two variable bindings (x_1, x_2) and (y_1, y_2) , if $x_{f(k)}[[Q_k]] = y_{f(k)}[[Q_k]], 1 \leq k \leq n$ then $x_{f(k)}[[Q_k]] = y_{f(k)}[[Q_k]], n + 1 \leq k \leq n + m$ ■

We call the first condition the *singleton condition*.

2.2 Types of XFDs

There are two basic types of XFDs: structural and semantic. Structural XFDs are used to capture the tree structure of an XML document and certain types of schema information. Semantic constraints are used to capture deeper knowledge of the data.

2.2.1 Structural XFDs

Since XML documents are trees, each node in the tree except the root has a unique parent node. If a node labeled by l' has a parent labeled by l , the *unique parent* constraint is expressed as:

$$\$x \text{ in } //l, \$y \text{ in } \$x/l', \$y \rightarrow \$x$$

The *unique child* constraint is based on the particular structure of an XML tree, and is used to distinguish between “one” or “many” occurrences of a child label. If a node labeled by l has only one child labeled by l' then:

$$\$x \text{ in } //l, \$x \rightarrow \$x/l'$$

Constraint C_1 is an example of a unique child constraint.

The *leaf node* constraint specifies that leaf nodes have exactly one value. Thus for every leaf node labeled by l we have: $\$x \text{ in } //l, \$x \rightarrow \$x/\text{value}()$

Note that function $value()$ is only defined for leaf nodes.

In our example, nodes labeled by *title* are leaves. The corresponding functional dependency is expressed as: $\$x \text{ in //title, } \$x \rightarrow \$x/value()$

Although producing structural XFDs by hand is tedious, they can be automatically generated by parsing the schema (XML DTD or XML-Schema) of the input XML data. If this schema information is not available, data mining techniques could be used to generate those constraints [11]. There is also an interesting interplay between structural XFDs and semantic XFDs, since the definition of XFDs poses some restrictions on the structure of the data. We will revisit this issue later.

2.2.2 Semantic XFDs

Semantic XFDs include the key constraints of XML-Schema [21] as well as the functional dependencies of [3]. C_2, C_4, C_5 are examples of semantic constraints.

It should be noted, however, that XFDs are not sufficiently general to capture the absolute and relative keys of [7], since P-attributes must evaluate to singleton sets of simple values. In contrast, the keys of [7] can evaluate to arbitrary sets of tree values, and the definition of satisfaction uses non-empty sets of values rather than equality. The reason for this restriction is because we are interested in generating a relational design for the XML document, and first-normal form prevents the use of complex values for attributes.

3 Reducing XFDs

We now turn to the implication problem: Given a set of XFDs, what others can be inferred and how?

Definition 3.1: An XFD $\varphi : X \rightarrow Y$ is *logically implied* by a set of functional dependencies F , written $F \models \varphi$, if and only if φ holds on every instance that satisfies all dependencies in F , that is, φ holds whenever all XFDs in F hold. ■

This problem is typically addressed by finding a set of inference rules, e.g. Armstrong’s Axioms for functional dependencies in relational databases, and proving that they are sound and complete. Compared to the relational counterpart, however, the task of finding such a set of inference rules for XFDs is much more difficult. This is because XFDs are based on path expressions (P-attributes) while relational FDs are defined on attribute names. Path expressions can interact with each other, and variable bindings are involved in the path expressions. We therefore give a set of rules that are sound but not known to be complete, and use the term “rewrite rules” rather than inference rules to make this point explicit.

3.1 Rewrite rules

Rewrite rules for XFDs must reason about variables as well as P-attributes. The first three extend Armstrong’s Axioms (*reflexivity*, *augmentation* and *transitivity*), to use path expressions instead of simple attributes. *Containment* considers the relationship between path expressions. *Singleton-path* exploits structural constraints imposed by the definition of XFDs. The remaining two rules, *variable-move* and *variable introduction and elimination*, handle variable bindings. Before introducing the axioms, we must therefore define path containment.

Definition 3.2: Given $p, q \in XP\{/, //\}$, let $p_{vf} = expand(p)$, $q_{vf} = expand(q)$. Then p is *contained* in q , denoted $p \preceq q$, if and only if:

- (1) Either both p_{vf} and q_{vf} end in “value()” or neither do; and
- (2) For any XML tree T , $\llbracket NodePath(p_{vf}) \rrbracket \subseteq \llbracket NodePath(q_{vf}) \rrbracket$.

Path expression p is *contained* in q w.r.t a DTD D , denoted $p \preceq_D q$, if $p \preceq q$ holds for any XML tree T that conforms to D . ■

Definition 3.3: Path expression p is *equivalent* to q , denoted $p \equiv q$, if and only if $p \preceq q$ and $q \preceq p$.

Path expression p is *equivalent* to q w.r.t a DTD D , denoted $p \equiv_D q$, if and only if $p \preceq q$ and $q \preceq p$ holds for any XML tree T that conforms to D . ■

For example, $//vendor \preceq /bib/vendor$ does not generally hold. However, if an XML tree conforms to the DTD D of figure 2, then $//vendor \preceq_D /bib/vendor$. Since $/bib/vendor \preceq //vendor$ always holds, and this implies $/bib/vendor \preceq_D //vendor$, we conclude that $/bib/vendor \equiv_D //vendor$.

The rewrite rules L for XFDs follow. We assume the variables used are $\$v_1$ and an optional $\$v_2$, where $\$v_1$ in $/P_1$ and $\$v_2$ in $\$v_1/P_2$, unless specified otherwise. We will also frequently use the XFD $\$v \rightarrow \v/Q , which means that “ $\$v/Q$ is known to exist and to be unique, and Q is a simple path.”

1. Reflexivity.

(a) **Variable Reflexivity:** $\$v \rightarrow \v

(b) **Path Reflexivity:**

$$\begin{aligned} & \$v_{f(k)} \rightarrow \$v_{f(k)}/Q_k, \\ & f(k) \in \{1, 2\} \text{ for } 1 \leq k \leq n, \\ & NodePath(Q_k) = NodePath(Q'_k) \\ & \implies \end{aligned}$$

$$\$v_{f(1)}/Q'_1, \dots, \$v_{f(n)}/Q'_n \rightarrow \$v_{f(l)}/Q'_l, 1 \leq l \leq n$$

Note that $NodePath$ is used to allow Q'_i to either add or drop “value()”.

2. Augmentation.

$$\$v_{f(1)}/Q_1, \dots, \$v_{f(n)}/Q_n \rightarrow \$v_{f(n+1)}/Q_{n+1},$$

$$\begin{aligned} & \$v_{f(n+2)} \rightarrow \$v_{f(n+2)}/Q_{n+2}, \\ & f(k) \in \{1, 2\} \ 1 \leq k \leq n+2 \\ \implies & \\ & \$v_{f(1)}/Q_1, \dots, \$v_{f(n)}/Q_n, \$v_{f(n+2)}/Q_{n+2} \rightarrow \\ & \$v_{f(n+1)}/Q_{n+1}, \$v_{f(n+2)}/Q_{n+2}. \end{aligned}$$

3. Transitivity.

$$\begin{aligned} & \$v_{f(1)}/Q_1, \dots, \$v_{f(n)}/Q_n \rightarrow \\ & \$v_{f(n+1)}/Q_{n+1}, \dots, \$v_{f(n+m)}/Q_{n+m}, \\ & \$v_{f(n+1)}/Q_{n+1}, \dots, \$v_{f(n+m)}/Q_{n+m} \rightarrow \\ & \$v_{f(n+m+1)}/Q_{n+m+1}, \dots, \$v_{f(n+m+l)}/Q_{n+m+l}, \\ & f(k) \in \{1, 2\}, 1 \leq k \leq (n+m+l) \\ \implies & \\ & \$v_{f(1)}/Q_1, \dots, \$v_{f(n)}/Q_n \rightarrow \\ & \$v_{f(n+m+1)}/Q_{n+m+1}, \dots, \$v_{f(n+m+l)}/Q_{n+m+l}. \end{aligned}$$

4. Containment.

$$\begin{aligned} & \$v_{f(1)}/Q_1, \dots, \$v_{f(n)}/Q_n \rightarrow \$v_{f(n+1)}/Q_{n+1}, \\ & f(k) \in \{1, 2\}, 1 \leq k \leq n+1 \\ & \$v'_1 \text{ in } P'_1, \$v'_2 \text{ in } \$v'_1/P'_2, \\ & \text{expand}(\$v'_1) \leq \text{expand}(\$v_1), \\ & \text{expand}(\$v'_2) \leq \text{expand}(\$v_2) \\ \implies & \\ & \$v'_{f(1)}/Q_1, \dots, \$v'_{f(n)}/Q_n \rightarrow \$v'_{f(n+1)}/Q_{n+1} \end{aligned}$$

Note that if an XFD only has one variable, $\$v_2$ and $\$v'_2$ are not used.

The intuition behind this rule is that if an XFD holds on a set of X-tuples S , then it holds on any subset of S .

5. Singleton-path.

(a) Defined by XFDs:

$$\begin{aligned} & \$v_{f(1)}/Q_1, \dots, \$v_{f(n)}/Q_n \rightarrow \$v_{f(n+1)}/Q_{n+1} \\ \implies & \\ & \$v_{f(k)} \leftrightarrow \$v_{f(k)}/\text{NodePath}(Q_k) \\ & \text{and } \$v_{f(k)} \rightarrow \$v_{f(k)}/Q_k, 1 \leq k \leq n+1 \end{aligned}$$

(b) Defined by variables:

$$\begin{aligned} & \$v_1 \rightarrow \$v_1/P_2/Q, \\ & \$v_2 \rightarrow \$v_2/Q \\ \implies & \\ & \$v_1 \rightarrow \$v_1/P_2 \end{aligned}$$

The intuition is that, by definition, any P-attribute in an XFD must exist and be unique.

6. Variable-move.

(a) First-Variable-move:

$$\begin{aligned} & \$v_1 \text{ in } P_1/P_2 \ \{, \$v_2 \text{ in } \$v_1/P_3\}, \\ & \$v'_1 \text{ in } P_1 \ \{, \$v'_2 \text{ in } \$v'_1/P_2/P_3\}, \\ & \$v'_1 \rightarrow \$v'_1/P_2, \end{aligned}$$

$$\begin{aligned} & \$v_{f(1)}/Q_1, \dots, \$v_{f(n)}/Q_n \rightarrow \$v_{f(n+1)}/Q_{n+1} \\ \iff & \\ & \$v'_{f(1)}/Q'_1, \dots, \$v'_{f(n)}/Q'_n \rightarrow \$v'_{f(n+1)}/Q'_{n+1} \\ & Q'_k = \begin{cases} P_2/Q_k & f(k) = 1 \\ Q_k & f(k) = 2 \end{cases} \ 1 \leq k \leq n+1 \end{aligned}$$

Curly braces are used to indicate omitted expressions for the one-variable case.

(b) Second-Variable-move:

$$\begin{aligned} & \$v_1 \text{ in } P_1, \$v_2 \text{ in } \$v_1/P_2/P_3, \\ & \$v'_1 \text{ in } P_1, \$v'_2 \text{ in } \$v'_1/P_2, \\ & \$v'_2 \rightarrow \$v'_2/P_3, \end{aligned}$$

$$\begin{aligned} & \$v_{f(1)}/Q_1, \dots, \$v_{f(n)}/Q_n \rightarrow \$v_{f(n+1)}/Q_{n+1} \\ \iff & \\ & \$v'_{f(1)}/Q'_1, \dots, \$v'_{f(n)}/Q'_n \rightarrow \$v'_{f(n+1)}/Q'_{n+1} \\ & Q'_k = \begin{cases} Q_k & f(k) = 1 \\ P_3/Q_k & f(k) = 2 \end{cases} \ 1 \leq k \leq n+1 \end{aligned}$$

Note that this only applies to two-variable XFDs.

The intuition is that variables can be moved along singleton paths to create new XFDs.

7. Variable introduction and elimination.

$$\begin{aligned} & \$v_2/Q_1, \dots, \$v_2/Q_n \rightarrow \$v_2/Q_{n+1} \\ \iff & \\ & \$v'_2 \text{ in } P_1/P_2 \\ & \$v'_2/Q_1, \dots, \$v'_2/Q_n \rightarrow \$v'_2/Q_{n+1} \end{aligned}$$

As an example of these rules, suppose we have:

ϕ_1 : $\$x$ in //vendor, $\$x \rightarrow \x/name ,

ϕ_2 : $\$y$ in //vendor/name, $\$y \rightarrow \$y/\text{value}()$,

Using variable-move with ϕ_2 , we can infer the XFD

ϕ_3 : $\$x$ in //vendor, $\$x/\text{name} \rightarrow \$x/\text{name}/\text{value}()$.

By applying transitivity to ϕ_1 and ϕ_3 , we now get:

ϕ_4 : $\$x$ in //vendor, $\$x \rightarrow \$x/\text{name}/\text{value}()$.

This use of variable-move and transitivity is very common.

As another example, suppose we have:

ϕ_1 : $\$z$ in //book,

$\$z/\text{ISBN}/\text{value}() \rightarrow \$z/\text{title}/\text{value}()$

Using containment, we can infer the following XFD:

ϕ_2 : $\$y$ in //vendor/book,

$\$y/\text{ISBN}/\text{value}() \rightarrow \$y/\text{title}/\text{value}()$.

This is because $\text{expand}(\$y) \leq \text{expand}(\$z)$.

Theorem 3.1: L is sound for XFDs defined over XML data. That is, if an XFD φ can be inferred from a set of XFDs F using L (written $F \vdash_L \varphi$), then for any XML instance in which F holds φ will also hold ($F \models \varphi$). ■

3.2 XFD inference

The mapping algorithm in the next section relies on an input set of XFDs G that are redundancy-reduced. We therefore give a polynomial time algorithm, *infer* (algorithm 1), which given an XFD $\phi : X \rightarrow Y$ and a set of XFDs F , determines whether or not ϕ can be inferred from F using L . Given an initial set of XFDs, we then use this algorithm to detect which XFDs can be eliminated and which ones can be simplified by eliminating P-attributes on their left hand sides, thereby deriving G .

In *infer*, reflexivity is applied in lines 5-6; reflexivity, augmentation and transitivity in lines 26-27; containment in line 19; variable-move in line 22; singleton-path in lines 5, 7 and 22; and variable introduction and elimination in lines 19 and 22. We assume each XFD (1) has a single P-attribute on its right-hand side; and (2) that the variable on the right-hand side either appears on the

left-hand side, or the variables on the left hand side are dependent variables.

As an example of *infer*, suppose we have a set of XFDs $F : \{\phi_1, \phi_2, \phi_3\}$ and an XFD φ :

$\$x$ in //vendor, $\$y$ in $\$x$ /book, $\$z$ in //book

$\phi_1: \$x, \$y/\text{ISBN}/\text{value}() \rightarrow \y

$\phi_2: \$x \rightarrow \x/name

$\phi_3: \$z/\text{ISBN}/\text{value}() \rightarrow \$z/\text{title}/\text{value}()$

$\varphi: \$x, \$y/\text{ISBN}/\text{value}() \rightarrow \$y/\text{title}/\text{value}()$

Using *infer*, we initialize $S = \{\$x, \$y/\text{ISBN}/\text{value}()\}$ since $\$x$ and $\$y/\text{ISBN}/\text{value}()$ appear in ϕ_1 (reflexivity, line 6) and they satisfy the singleton condition. After processing ϕ_1 and ϕ_2 , we add $\$y$ and $\$x/\text{name}$ to S (reflexivity and transitivity, lines 26-27) resulting in the set $S = \{\$x, \$y/\text{ISBN}/\text{value}(), \$y, \$x/\text{name}\}$. When processing ϕ_3 , since //vendor/book \preceq //book, we can infer

$\phi_4: \$z'$ in //vendor/book,

$\$z'/\text{ISBN}/\text{value}() \rightarrow \$z'/\text{title}/\text{value}()$

using containment. Applying variable introduction and elimination (line 19), we have:

$\$y/\text{ISBN}/\text{value}() \rightarrow \$y/\text{title}/\text{value}()$.

Using reflexivity and transitivity (line 26-27), we can now add $\$y/\text{title}/\text{value}()$ to S and conclude that ϕ can be inferred by F using L .

There are two subtleties in algorithm 1: the bound l (line 4), and the marking of P-attributes in S (lines 21 and 24). The motivation behind l is that there are certain constraints (e.g. unique child) that can generate arbitrarily long P-attributes. Consider the following example:

$\$x'$ in //a, $\$x$ in /a

$\phi: \$x' \rightarrow \x'/a

$\varphi_1: \$x \rightarrow \$x/a/a/a$

$\varphi_2: \$x \rightarrow \x/b

Suppose $F = \{\phi\}$ and we want to check if $F \vdash_L \varphi_1$. Initially $S = \{\$x\}$ (reflexivity). By containment, variable move and transitivity, we can expand S to $\{\$x, \$x/a\}$. To see this, let $\$y$ in //a/a. By containment, $\$y \rightarrow \y/a . Using variable move, we get $\$x/a \rightarrow \$x/a/a$, and can therefore add $\$x/a/a$ to S (transitivity). Repeating this set of steps, we can also add $\$x/a/a/a$ to S . At this point we can stop because the target has been reached. However, checking $F \vdash_L \varphi_2$ would cause S to expand infinitely. We therefore need to know when we can safely stop, i.e. we need a bound on the length of paths that are “useful” to add to S .

Since P-attributes are added to S by the input constraint $X \rightarrow Y$ or by transitivity from XFDs in F , it is clear that l must be at least as long as the maximum simple path length in a P-attribute in F or $X \rightarrow Y$. We must also account for the fact that variable-move increases the simple path length of a P-attribute using simple paths in the variable definitions. We therefore define $\text{length}(\$x/P)$ to be the number of labels in $\text{NodePath}(\text{expand}(\$x/P))$.

The motivation behind marking P-attributes in S by (ϕ, v) is to avoid unnecessary recalculations. Observe

that for $s \in S$ and $\phi \in F$, s can be constructively used in ϕ at most twice, once for each variable in ϕ . To see this, suppose that $s = \$x'/R$ matches two different P-attributes $\$x/P$ and $\$x/Q$ in the left hand side of ϕ , each using the same variable. Then $\$x'/R \preceq \x/P and $\$x'/R \preceq \x/Q . Since P and Q are simple path, this means $P = Q$, a contradiction.

Algorithm 1

```

1: function infer
2: input:  $(F, \varphi : X \rightarrow Y)$ 
3: output: True, if  $F \vdash_L \varphi$ ; False, otherwise.
4: let  $l = \max\{\text{length}(\$v/Q) \mid \$v/Q \text{ is a P-attribute of some XFD in } F \text{ or } \varphi\}$ 
5: if  $\forall \$v/Q \in X$  satisfies the singleton condition then
6:    $S = X$ 
7:   if  $\exists \$v/P \in X$  and  $P$  does not end with  $\text{value}()$  then
8:      $S = S \cup \{\$v\}$ 
9:   end if
10: else
11:   return false
12: end if
13: if  $Y \in S$  then
14:   return true
15: end if
16: repeat
17:    $B' = \text{Null}$ 
18:   for each XFD  $\phi : A \rightarrow B \in F$  do
19:     if  $\forall \$v/R_j \in A, \exists \$v'/R_j \in S$  not marked by  $(\phi, \$v)$ , and  $\text{expand}(\$v') \preceq \text{expand}(\$v)$  then
20:        $B' =$  replace each  $\$v$  with  $\$v'$  in  $B$ 
21:       Mark  $\$v'/R_j$  by  $(\phi, \$v)$  if the variable of  $B$  is  $\$v$ , or  $\$v$  is the dependent variable.
22:     else if  $\forall \$v/R_j \in A, \exists \$v'/R'_j \in S$  not marked by  $(\phi, \$v)$ ,  $\text{expand}(\$v'/R'_j) \preceq \text{expand}(\$v/R_j)$ , and the path between  $R'_j$  and  $R_j$  satisfies the singleton condition then
23:        $B' =$  replace each  $\$v$  with  $\$v'$  and adjust the path under  $\$v'$ .
24:       Mark  $\$v'/R'_j$  by  $(\phi, \$v)$  if the variable of  $B$  is  $\$v$ , or  $\$v$  is the dependent variable.
25:     end if
26:     if  $B' \neq \text{Null}$  and  $|B'| \leq l$  then
27:        $S = S \cup \{B'\}$ 
28:     end if
29:     if  $Y = B'$  then
30:       return true
31:     end if
32:   end for
33: until  $S$  does not enlarge
34: return false

```

Now, suppose $\phi : \$v_1/P_1, \dots, \$v_2/P_2 \rightarrow \$v_1/Q$, and s is $\$v'/R$ where $\$v'/R \preceq \v_1/P_1 . Then reusing s to match $\$v_1/P_1$ will not add anything new to S as the conclusion $(\$v'/Q)$ will remain the same. We therefore mark s by $(\phi, \$v_1)$. Similarly, if s matches $\$v_2/P_2$ where $\$v_2$ is the dependent variable, the conclusion will have been fixed by whatever $s' \in S$ was used to match $\$v_1/P_1$. We therefore mark s by $(\phi, \$v_2)$.

Lemma 3.2: Algorithm 1 runs in polynomial time.

Proof: Let n be the number of XFDs in F (this is also the number of P-attributes on the right hand side of XFDs in F) and m be the number of P-attributes in X . The crux of the proof is that $|S| \leq (m + n)l$, and that each XFD in F is used at most $2|S|$ times. Since each use of an XFD marks at least one element in S , there are at most $2(m + n)ln$ iterations of the repeat loop. ■

The complexity of the algorithm also depends on the path language used and the presence (or absence) of a DTD. For PL , there is a linear-time containment algorithm in the absence of DTD [18]. For the same path language in the presence of DTD, [19] shows that the containment problem is polynomial in the overall size of the DTD and path expressions tested.

The (polynomial time) algorithm to compute a reduced set G of a set of XFDs F , *Reduce*, is similar to the relational algorithm for finding the minimum cover of a set of functional dependencies [5], but uses *infer* instead of attribute closure.

In the next section, we will discuss how to map XML data into relations based on the reduced set of XFDs.

4 Constraint preserving relational storage

Our XML-to-relational mapping takes an XML “schema”, i.e. a set of XFDs and an optional DTD, and generates a normalized relational schema as well as the instance transformation program. More formally:

Input: A set of XFDs F , and an optional DTD D .

Output: A target relational schema R with a set of keys K , and a *redundancy reducing, constraint preserving transformation* M .

Redundancy reducing means that redundancy which can be detected by F using L is eliminated in R . As an example, consider the sample XML data in figure 1, and suppose the following XFD holds:

$\$x$ in //book, $\$x/ISBN/value() \rightarrow \$x/title/value()$

This means that if there is a book node with an ISBN value “0471942073” and a title value “Harry Potter”, we can conclude that any other book node with the same ISBN value will have the same title value. When we store the XML tree into relations, this redundancy in the XML document can be removed by creating a relation which stores the relationship between ISBN and title exactly once, and specifies ISBN as the key.

Constraint-preserving means that, for any XML tree T , F hold on T if and only if K hold on $M(T)$.

The transformation M will map an XML tree T which conforms to D and satisfies F to relations $M(T)$ which conform to schema R .

Algorithm 2

1: **function** RRXS

2: **input:** F , optional DTD D

3: **output:** R with K defined, M

4: $G = Equivalence(F, D)$

5: $H = Reduce(G, D)$

6: $I = Shrink(H)$

7: Map each distinct P-attribute p in I to an attribute p_a

8: $M = \cup_{p \in I} (p_a \leftarrow p)$

9: Let A be the set of attributes obtained

10: Map I to functional dependencies I_R over A

11: Generate a 3NF relational schema R over the attribute set A according to I_R

12: **return** R, M

4.1 Schema mapping algorithm: RRXS

The mapping algorithm RRXS is presented in algorithm 2. We assume that every node is assigned a unique node id to guarantee that the parent-child connections between nodes are preserved. However, if there are other means to identify a node (for example, a semantic key value), it is not necessary to keep the id for that node. The removal of redundant node id is based on the following observation: if $X \rightarrow Y$ and $Y \rightarrow X$ then X and Y are functionally equivalent. If Y does not end with “value()”, we can replace Y with X . The first step, *Equivalence* will recognize equivalent XFDs and equivalent elements (an element is a set of P-attributes which appear on the left or right-hand side of an XFD), then group those elements into equivalence classes and output G . In the second step, the reduced set H of G is computed to remove redundant XFDs. Then for each equivalence class, *shrink* removes unnecessary elements, producing the set of XFDs I . During the fourth step, every non-equivalent P-attribute p in I is mapped into a relational attribute p_a to record the ids or values of the nodes reachable by p , and I is mapped into a set of functional dependencies I_R . Finally, a third normal form (3NF) target relational schema R is generated based on I_R . The optional XML schema information D can be used to automatically generate structural XFDs, and also used in the path containment test in the reduced cover algorithm.

4.1.1 Equivalence

The intuition of *Equivalence* is that we try to find relationships between variables and P-attributes to recognize redundant node ids. Equivalence consists of two steps:

(1) If two XFDs ϕ_1 and ϕ_2 satisfy $\phi_1 \implies \phi_2$ and $\phi_2 \implies \phi_1$, then we choose the one that minimizes the number of variables used for a given set of XFDs;

(2) If we have two XFDs $\phi_3: X \rightarrow Y$ and $\phi_4: Y \rightarrow X$ then we group elements X and Y into an equivalence class.

There are several advantages to using equivalence classes rather than the set of XFDs they represent. First, we re-

Algorithm 3

```
1: function Equivalence
2: input:  $F, D$ 
3: output:  $G$ 
4: Construct  $C$  using the unique child and unique parent
   XFDs in  $F$ 
5:  $F' = F -$  the XFDs used to construct  $C$ 
6:  $n = |F'|$ 
7: for  $i = 1$  to  $n$  do
8:   let  $\phi_i$  be  $X \rightarrow Y$ 
9:   if  $\forall X_k \in X, \exists (X'_k \in C \text{ or } X'_k \in \phi_j, i <$ 
      $j \leq n)$  such that:  $\text{expand}(\text{NodePath}(X'_k)) \equiv$ 
      $\text{expand}(\text{NodePath}(X_k))$  then
10:    if  $X \rightarrow Y \iff X' \rightarrow Y'$  then
11:      replace  $\phi_i$  with  $X' \rightarrow Y'$ 
12:    end if
13:  end if
14: end for
15: for  $i = 1$  to  $n$  do
16:   let  $\phi_i$  be  $X \rightarrow Y$ 
17:   if  $((\text{expand}(X) \in PL_2 \text{ and } |X| = 1) \text{ or } \text{expand}(Y)$ 
      $\in PL_2)$  and  $\text{infer}(F, Y \rightarrow X)$  then
18:     put  $X$  and  $Y$  into the same equivalence class  $C_i$ 
19:     remove  $\phi_i$ 
20:   end if
21: end for
22:  $G = (F', C)$ 
23: return  $G$ 
```

duce the number of inferences performed. For example, if Y is in the same equivalence class as X , we know $\text{infer}(F, X \rightarrow Y) = \text{True}$ immediately without computing it twice. We therefore pass equivalence classes into *reduce*. Second, if Y does not end with “value()”, we can use X to represent Y and remove the redundant node ids associated with Y .

Equivalence is presented in algorithm 3. Note that each “unique child” constraint $X \rightarrow Y \in F$ will place X and Y into the same equivalence class, since the “unique parent” $Y \rightarrow X$ is always true. In this algorithm, C is the set of equivalence classes created according to “unique child” and “unique parent”, and F' is the set of XFDs excluding those represented in C .

To illustrate, consider the book-vendor example in figure 3 with its DTD in figure 2. For clarification, we use distinct variable names and put all the variable bindings at the beginning. First, we rewrite the XFDs in figure 3 to minimize the number of variables needed. Since XFD (2) can be rewritten as $\$x/\text{name} \rightarrow \$x/\text{name}/\text{value}()$, $\$n$ is removed. Similarly we rewrite XFDs (3), (4), (5), (6) and remove variable bindings $\$w, \$I, \$t, \p . Then we process each XFD in F' to see if it can create or enlarge equivalence classes. Since $\text{infer}(F, \$x/\text{name}/\text{value}() \rightarrow \$x/\text{name}) = \text{true}$, we enlarge C_1 to $\{\$x, \$x/\text{name}, \$x/\text{webpage}, \$x/\text{name}/\text{value}()\}$ and remove (2). Now XFD (7) can be obtained directly from C_1 , and is removed. When we process XFD (8), since $\text{infer}(F, \$z \rightarrow \$x) = \text{true}$ and $\text{infer}(F, \$z \rightarrow \$z/\text{ISBN}/\text{value}()$

Variable bindings:

```
 $\$x$  in //vendor,    $\$y$  in //book,    $\$z$  in  $\$x/\text{book}$ ,
 $\$n$  in //name,     $\$w$  in //webpage,  $\$I$  in //ISBN,
 $\$t$  in //title,    $\$p$  in //price
```

$F = (F' = \{(1)-(9)\}, C = \{C_1, C_2\})$

“Unique parent” constraints:

(1): $\$z \rightarrow \x

“leaf node” constraints:

(2): $\$n \rightarrow \$n/\text{value}()$ (3): $\$w \rightarrow \$w/\text{value}()$

(4): $\$I \rightarrow \$I/\text{value}()$ (5): $\$t \rightarrow \$t/\text{value}()$

(6): $\$p \rightarrow \$p/\text{value}()$

User-defined semantic XFDs:

(7): $\$x/\text{name}/\text{value}() \rightarrow \x

(8): $\$x, \$z/\text{ISBN}/\text{value}() \rightarrow \z

(9): $\$y/\text{ISBN}/\text{value}() \rightarrow \$y/\text{title}/\text{value}()$

$C_1 = \{\{\$x\}, \{\$x/\text{name}\}, \{\$x/\text{webpage}\}\}$

$C_2 = \{\{\$y\}, \{\$y/\text{title}\}, \{\$y/\text{ISBN}\}, \{\$y/\text{price}\}\}$

Figure 3: XFDs for book vendor example

$) = \text{true}$, we create a new equivalence class $C_3: \{\$z, (\$x, \$z/\text{ISBN}/\text{value}())\}$ and remove XFDs (1) and (8). The resulting set G of XFDs and equivalence classes are shown below. (We omit variable bindings for brevity.)

(3): $\$x/\text{webpage} \rightarrow \$x/\text{webpage}/\text{value}()$

(4): $\$y/\text{ISBN} \rightarrow \$y/\text{ISBN}/\text{value}()$

(5): $\$y/\text{title} \rightarrow \$y/\text{title}/\text{value}()$

(6): $\$y/\text{price} \rightarrow \$y/\text{price}/\text{value}()$

(9): $\$y/\text{ISBN}/\text{value}() \rightarrow \$y/\text{title}/\text{value}()$

$C_1 = \{\{\$x\}, \{\$x/\text{name}\}, \{\$x/\text{webpage}\}, \{\$x/\text{name}/\text{value}()\}\}$

$C_2 = \{\{\$y\}, \{\$y/\text{title}\}, \{\$y/\text{ISBN}\}, \{\$y/\text{price}\}\}$

$C_3 = \{\{\$z\}, \{\$x, \$z/\text{ISBN}/\text{value}()\}\}$

Using *reduce*, we can remove redundant XFDs in G . In our running example, since XFD (4) and (5) can be inferred by other XFDs and C , they are removed. The reduced set of XFDs is the following set H :

(3): $\$x/\text{webpage} \rightarrow \$x/\text{webpage}/\text{value}()$

(6): $\$y/\text{price} \rightarrow \$y/\text{price}/\text{value}()$

(9): $\$y/\text{ISBN}/\text{value}() \rightarrow \$y/\text{title}/\text{value}()$

C_1, C_2, C_3 are not changed.

4.1.2 Shrink

In this step, we remove redundant node ids in the equivalence classes. The idea of shrink is based on the following observation:

Lemma 4.1: Let X and Y be path expressions. If $X \rightarrow Y$ and $Y \rightarrow X$ then X and Y are interchangeable. ■

The lemma is proven using reflexivity and transitivity.

If X and Y are two elements in one equivalence class, then $X \rightarrow Y$ and $Y \rightarrow X$. To minimize node ids, we choose one representative element X , where $|X| = 1$, for each equivalence class C_i , and use X to replace the

```

$x in //vendor, $y in //book, $z in $x/book
(3'): $x/name/value() → $x/webpage/value()
(6'): $y → $y/price/value()
(9'): $y/ISBN/value() → $y/title/value()
(10): $x/name/value(),$z/ISBN/value() → $z
(11): $z → $x/name/value(),$z/ISBN/value()

```

Figure 4: Minimal set of XFDs and variable bindings for book vendor example

occurrences of other singleton elements of C_i which do not end with “value()”. When possible, we choose X which is a semantic key (that is, X ends with “value()”). However, if the semantic key is composed of more than one P-attribute, we use a singleton element X as the representative in order to speed up joins between relations when we store these P-attributes in relations. The shrink algorithm is as follows:

1. If $\exists X = \{\$v/P\} \in C_i$ where P is a simple path ending with “value()”, then use $\$v/P$ as the representative of C_i ; else select $X = \{\$v/P\} \in C_i$, where P is any simple path. Use $\$v/P$ to replace the occurrences of other singleton elements $\$v/Q$ in C_i , where Q does not end with “value()”.
2. Replace $\$v'/Q \in H$ by $\$v'/P$, where $expand(\$v') \prec expand(\$v)$ (containment rule).
3. If $\exists Y \in C_i$ such that $|Y| \geq 1$, then add XFDs $X \rightarrow Y$ and $Y \rightarrow X$ to H .
4. If $\exists Z \in C_i$ such that Z ends with “value()” and $Z \neq X$, then add $X \rightarrow Z$ and $Z \rightarrow X$ to H .

Continuing with our example, we process each equivalence class in turn. From C_1 , we choose $\$x/name/value()$ as the representative and replace all occurrences of $\$x$, $\$x/name$, $\$x/webpage$ with it. Then we choose $\$y$ as the representative of class C_2 , and replace $\$y/title$, $\$y/ISBN$, $\$y/price$ with $\$y$. For C_3 , $\$z$ is the representative and we generate $\$z \rightarrow \$x/name/value()$, $\$z/ISBN/value()$ and $\$x/name/value()$, $\$z/ISBN/value() \rightarrow \z . In this way, we remove all equivalence classes and obtain the set of XFDs and bindings shown in figure 4.

4.1.3 Transformation M

For each distinct P-attribute p in I , we create a new relational attribute p_a . Two P-attributes p and q are distinct if and only if $expand(p) \not\equiv expand(q)$. For P-attribute p in I ending with “value()”, p_a records the values of the nodes reachable by p ; otherwise, p_a records the node ids.

Let the set of attributes created be A . The transformation M is a list of pairs of relational attributes and P-attributes of form $p_a \leftarrow p$.

Returning to our example, we generate a relational attribute for each distinct path expression in I . For clarification, the attribute name generated is the last label in the corresponding path expression. The transformation M is as follows:

```

name ← $x/name/value(),
webpage ← $x/webpage/value(),
bookID ← $y, $z
price ← $y/price/value()
ISBN ← $y/ISBN/value(), $z/ISBN/value()
title ← $y/title/value()

```

Note that since $expand(\$y/ISBN/value()) \equiv expand(\$z/ISBN/value())$, they are mapped to the same relational attribute *ISBN*. Similarly, $\$y$ and $\$z$ are both mapped to attribute *bookID*.

We now map the XFDs to functional dependencies in the target relational schema R by replacing each P-attribute in I with the relational attribute name it maps to. In our example, this yields the following set of functional dependencies I_R :

```

(3'): name → webpage
(6'): bookID → price
(9'): ISBN → title
(10): name,ISBN → bookID
(11): bookID → name,ISBN

```

Finally, we generate a third-normal form relational schema R over A using I_R [5]. In our example, we give each resulting table a name according to its meaning to yield the following:

```

Vendor(name, webpage) with {name} as key.
Book(ISBN, title) with {ISBN} as key.
Sell(name, ISBN, price, bookID) with {name, ISBN}
and {bookID} as two sets of keys.

```

4.2 Instance mapping

The instance mapping takes an XML tree T which conforms to DTD D and satisfies the XFDs F as well as the schema mapping output M , and generates a relational instance $M(T)$ which conforms to schema R .

For example, to populate the relation *Sell*(*name*, *ISBN*, *price*, *bookID*), we first find the source of each attribute using M :

```

name ← $x/name/value(),
ISBN ← $y/ISBN/value(), $z/ISBN/value()
price ← $y/price/value()
bookID ← $y, $z

```

where variables $\$x$, $\$y$ and $\$z$ are as defined in figure 4. The instance mapping can be illustrated by the following XQuery-SQL expression [11]:

```

for $x in //vendor, $z in $x/book, $y in //book
where $y = $z

```

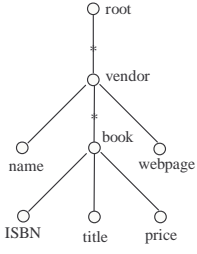
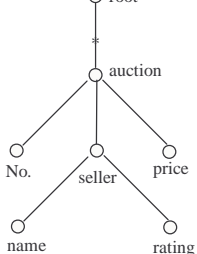
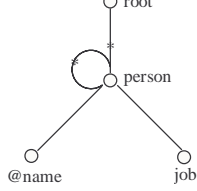
	Book	Auction	Person
DTD			
User	$\$x$ in // vendor	$\$x$ in // auction	$\$x$ in // person
Defined	$\$x/\text{name}/\text{value}() \rightarrow \$x;$ $\$y$ in // vendor, $\$y$ in $\$x/\text{book}$	$\$x/\text{No.}/\text{value}() \rightarrow \$x;$ $\$y$ in // seller $\$y/\text{name}/\text{value}()$	$\$x/\text{@name}/\text{value}() \rightarrow \$x;$
XFDs	$\$y/\text{ISBN}/\text{value}(), \$x \rightarrow \$y;$ $\$z$ in // book $\$z/\text{ISBN}/\text{value}()$ $\rightarrow \$z/\text{title}/\text{value}()$	$\rightarrow \$y/\text{rating}/\text{value}()$	
Hybrid Inlining	Vendor(<u>ID</u> , name, webpage) Book(<u>ID</u> , ISBN, title, price, parentID)	Auction(<u>ID</u> , No., price, name, seller@rating)	Person(<u>ID</u> , @name, job, parentID)
X2R	Vendor(vid, <u>name</u> , webpage) Book(bid, <u>ISBN</u> , title, price, pid)	Auction(aid, <u>No.</u> , price, name, seller@rating)	Person(id, <u>@name</u> , job, pid)
ICDE	Vendor(<u>name</u> , webpage) Book(<u>ISBN</u> , <u>name</u> , title, price)	Auction(<u>No.</u> , price, name, seller@rating)	Person(<u>@name</u> , job)
RRXS	Vendor(<u>name</u> , webpage) Book(<u>ISBN</u> , title) Sell(<u>name</u> , <u>ISBN</u> , price, <u>bookID</u>)	Auction(<u>No.</u> , price, name) Seller(<u>name</u> , rating)	Person(<u>@name</u> , job) Parent(<u>child@name</u> , @name)

Figure 5: Result schemas

insert into *Sell*

values($\$x/\text{name}/\text{value}(), \$y/\text{ISBN}/\text{value}(),$
 $\$z/\text{price}/\text{value}(), \$y)$

In the implementation of the mapping algorithm, we build an automata for each path expression to identify the node ids and values to be stored in attributes. Since all these automatons can run in parallel, the instance mapping can be done by one traversal of the original XML document [15].

Lemma 4.2: An XML document T satisfies the set of XFDs F if and only if the target relations $M(T)$ satisfy the set of functional dependencies I_R . ■

5 Experimental Evaluation

To evaluate RRXS, we compare its performance with hybrid inlining [20]. We do not compare our technique with that of LegoDB[6] since LegoDB focuses on tuning an initial design based on expected queries, and thus represents a different design stage. The comparison is performed in terms of the quality of the resulting schema, the size of the mapped relational instances, and the time to check XFDs mapped to relational integrity checks.

The experiments were performed on three data sets:

book [2], which has repeated *ISBN*, *title* information; *auction* [24], which has several “single-child” constraints; and *person* [23], which is recursive (see the top of figure 5 for simplified versions of the DTDs).

All experiments were conducted on a 1.5GHz Pentium 4 machine with 512MB memory and one hard disk with 7200rpm running windows 2000. The DBMS is DB2 universal version 7.2 using high-performance storage.

Schema Generation. The first experiment tests the quality of the resulting schemas, and is summarized in figure 5. In addition to hybrid inlining, we compare the RRXS schema with that produced by X2R [9], which extends hybrid inlining to capture XML keys, and the technique of [10] (termed ICDE), which represents a 3NF decomposition of a universal relation based solely on XML keys. The difference between our approach and the latter two is the consideration of general XFDs as opposed to XML keys. We observe the following about the schema resulting from our method:

1. Some node ids (*ID*, *parentID*) generated by hybrid inlining are removed. This was possible since in all three schemas, as each instance node can be uniquely identified using value information. Note that X2R does not remove these node ids.
2. User defined XFDs are correctly used to eliminate

<i>Filesize(KB)</i>	259	519	779	1040	1300
<i>Hybrid Inlining</i>	248.85	475.57	702.3	930.1	1157
<i>RRXS</i>	134.07	194.96	255.5	316.4	377.1

Figure 6: Size of relations in KBs

<i>Filesize(KB)</i>	259	519	779	1040	1300
<i>Hybrid Inlining</i>	0.244	0.597875	1.04475	1.444625	19.1455
<i>RRXS</i>	0.16775	0.02645	0.0605	0.304875	0.061235

Figure 7: Time in seconds to check XFDs

redundancies. For example, in the *book* schema, there is a *book* “entity” relation with *ISBN* and *title* information and a *sell* “relationship” relation with *price* information. As another example, in the *auction* schema there is a separate *seller* “entity” relation which captures the *rating* of each seller. Only the seller *name* is used in the *auction* relation. Note that neither X2R nor ICDE removes this redundant information.

3. The strategy works correctly for recursive data. Note that ICDE fails to correctly capture parent information.

Resulting relational instances. The second experiment tests the size of the resulting relational instances (see figure 6). Results for the *book* schema are shown, with the XML instance size varying from 250 KB to roughly 1250 KB. As can be seen, RRXS generates much smaller relational instances than those of hybrid inlining due to redundant information being captured by XFDs. The other data sets have similar results and are omitted.

Constraint checking. The final experiment shows the time needed to enforce the XFDs in the resulting relational design (see figure 7). The time is measured as the difference between the total time to insert tuples one by one into the database with checking turned on, and the total time to insert tuples one by one into the database with checking turned off. Each data point represents the average of 8 runs, after dropping the minimum and maximum values. For hybrid inlining, each semantic XFD is mapped by hand to a triggered procedure. Since in RRXS each semantic XFDs is mapped to a primary key constraint, the checking time is negligible. Again, we only show results for the book schema since the other data sets have similar results.

6 Related Work

Constraints for XML. The notion of keys for XML was introduced in [7] which allows the key value of a node to be a set of trees. Our XFDs can express the keys in [7] when the key value is a single string value. This restriction is influenced by how keys are being defined in XML-

Schema [21], and how they are being used in practice. [3] proposes the concept of functional dependencies for XML. In our work, the XFDs are expressed by variable bindings and support certain regular expressions. [12] considers a broad class of constraints for XML, including XML keys and those implied by a schema (DTD or XML-Schema). XFDs are a subset of the constraints of [12], and borrows the idea of capturing structural and semantic constraints in the same framework.

Reasoning about constraints for XML. [22] proposes a sound and complete set of inference rules for unary functional dependencies for XML defined in [3], where the path expressions are variable-free simple path expressions. [8] gives sound and complete axioms for reasoning over general XML keys defined in [7], and [10] a more efficient set of axioms for a restriction of XML keys. Our work generalizes this to XFDs, but our rewrite rules are not known to be complete.

XML storage in relations. There has been significant interest in mapping XML documents to relations for storage [11, 13, 20, 6, 9]. Though constraints are closely related to the schema design, earlier work [11, 13, 20] only considered the structure of the XML document to design a target relational schema. [6] gives an optimal relational schema in a space of storage mappings according to the content, structure and query workload for some cost model of the XML data. The procedure is analogous to the tuning procedure for physical database design in the relational database. Our approach is analogous to the initial stage of database design (normalization) based on functional dependencies. [9] uses XML key/foreign key information and hybrid inlining to design the target schema design. The storage mapping in this paper extends that work by considering more general constraints, XFDs, and is not tied to hybrid inlining.

Our work borrows from [17, 16] the idea of considering structural and semantic constraints in a uniform framework. However, their work address a much more general question, how to use constraints in the target database to optimize query processing in source. Our work is also related to [3, 4], which consider normal forms for XML data.

7 Conclusions

We have investigated the problem of how to design a normalized relational schema for XML data and how to automate the instance mapping. To achieve this goal, we express structural and semantic XML constraints in a uniform framework (XFDs), and develop a sound set of inference rules to reason about them. A reduced set of the input XFDs is used to guide the design of the target relational schema by translating XFDs to relational functional dependencies and creating a third normal form (3NF) decomposition. In this way, XFDs are mapped to relational keys and efficient relational primary key technology can be used to validate semantic constraints. Additionally, redundant information in the XML document as expressed in XFDs is reduced in the relational design, and the use of node ids is reduced wherever value-based keys exist. Our approach is analogous to the initial stage of relational database design based on functional dependencies. Experiments show that the relational schema generated by RRXS capture the semantics of the data very well.

To achieve information lossless, documents must be completely covered by XFDs.

Since the rewrite rules are not known to be complete, we cannot argue that the algorithm is optimal in reducing redundancy. In future work, we plan to develop a sound and complete set of rules, and consider how the conceptual schema design can be refined according to the query workload.

Acknowledgments. The authors would like to thank Wenfei Fan for his input and advice.

References

- [1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [2] Anonymous. bib. <http://www.cs.wisc.edu/niagara/data/bib/>.
- [3] M. Arenas and L. Libkin. A normal form for XML documents. In *Proceedings of the 21th Symposium on Principles of Database Systems (PODS)*, pages 85–96, 2002.
- [4] M. Arenas and L. Libkin. An Information-Theoretic Approach to Normal Forms for Relational and XML Data. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, 2003.
- [5] C. Beeri and P. A. Bernstein. Computational problems related to the design of normal form relational schemas. *TODS*, 4(1):30–59, 1979.
- [6] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML-Schema to Relations: A Cost-Based Approach to XML Storage. In *ICDE*, 2002.
- [7] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *WWW10*, pages 201–210, 2001.
- [8] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about keys for XML. In *International Workshop on Database Programming Languages (DBPL)*, 2001.
- [9] Y. Chen, S. B. Davidson, and Y. Zheng. Constraint Preserving XML Storage in Relations. In *WebDB*, 2002.
- [10] S. Davidson, W. Fan, C. Hara, and J. Qin. Propagating XML Constraints to Relations. In *ICDE*, 2003.
- [11] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *In Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, pages 431–442, 1999.
- [12] A. Deutsch and V. Tannen. Containment and Integrity Constraints for XPath Fragments. In *KRDB*, 2001.
- [13] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. In *Bulletin of the Technical Committee on Data Engineering*, pages 27–34, September 1999.
- [14] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the 17th International Conference on VLDB*, 1997.
- [15] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *The 9th International Conference on Database Theory (ICDT)*, 2003.
- [16] L. Popa and A. Deutsch and A. Sahuguet and V. Tannen. A Chase Too Far? In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Dallas, USA, May 2000.
- [17] L. Popa and V. Tannen. An Equational Chase for Path-Conjunctive Queries, Constraints, and Views. In *International Conference on Database Theory (ICDT)*, Jerusalem, Israel, January 1999.
- [18] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.
- [19] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *The 9th International Conference on Database Theory (ICDT)*, 2003.
- [20] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *The VLDB Journal*, pages 302–314, 1999.
- [21] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 0: Primer, May 2001. <http://www.w3.org/TR/xmlschema-0/>.
- [22] M. W. Vincent and J. Liu. Completeness and Decidability Properties for Functional Dependencies in XML, 2003. unpublished.
- [23] X-Hive Corporation. census.xml. <http://support.x-hive.com/xquery/index.html>.
- [24] XMARK the XML-benchmark project, April 2001. <http://monetdb.cwi.nl/xml/index.html>.