

# Deriving Relation Keys from XML Keys

Qing Wang<sup>†</sup>

Hongwei Wu<sup>†</sup>

Jianchang Xiao<sup>†</sup>

Aoying Zhou<sup>†</sup>

Junmei Zhou<sup>‡</sup>

<sup>†</sup>Department of Computer Science and Engineering  
Fudan University,  
220 Handan Rd., Shanghai, China  
Email: {qingwang,hwwu,jcxiao,ayzhou}@fudan.edu.cn

<sup>‡</sup>Shanghai R&D Institute of ZTE Corporation,  
396 Guilin Rd., Shanghai, China  
Email: zhou.junmei@zte.com.cn

## Abstract

Much work on XML data was around storage and querying and did not consider constraints of XML, especially keys. Since constraints have been proposed in many papers for XML, much research work on constraints has been being done. In this paper, we consider an important class of constraints, XML keys, and try to find the relationship between XML keys and relation keys. Given XML data whose semantics are represented in XML keys, we use a simple transformation language to express a transformation from the XML data to a relational database. Then we provide a formal definition of a relation tree that represents a well-formed rule on a relation written in the transformation language. After all rules that make up the transformation have been written, we show the way how XML data is transformed into relations. Finally, we present an algorithm for deriving relation keys from XML keys. These keys should be specified on the transformed relations or their compatibility with the predefined schema should be verified.

*Keywords:* XML key, relation tree

## 1 Introduction

Over the past several years, XML (Bray, Paoli & Sperberg-McQueen 1998) has been popular as a prime data exchange format on the web. Because there are many features (such as self-description, strong capacity of expressing) that XML holds, it has been being used widely in many fields. In practical applications, a data provider always stores its data in relational databases in advance and publishes the relational data in XML when necessary (Fernandez, Tan & Suciu 2000, Shanmugasundaram, Shekita, Barr, Carey, Lindsay, Pirahesh & Reinwald 2001, Shanmugasundaram, Kiernan, Shekita, Fan & Funderburk 2001, Oracle n.d., Intelligent Systems Research n.d.); on the other side, a data consumer imports XML data on the web into relational databases to store.

In some cases, XML, including the content and the structure, is mapped into relations by some methods (Shanmugasundaram et al 1999, Florescu & Kossmann 1999). Then, when a query of XML data (Chamberlin et al 2001, Abiteboul, Quass, McHugh, Widom & Wiener 1997) is submitted, it is mapped into a SQL query and executed (Zhou, Lu, Zheng, Liang, Zhang, Ji & Tian 2001, Manolescu, Florescu & Kossmann 2001). In other cases, some relational databases with actual data have been created.

Copyright ©2003, Australian Computer Society, Inc. This paper appeared at Fourteenth Australasian Database Conference (ADC2003), Adelaide, Australia. Conferences in Research and Practice in Information Technology, Vol. 17. Xiaofang Zhou and Klaus-Dieter Schewe, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

One of tasks for a data consumer is to specify transformations from XML data to relational databases, import XML data on the web and append it to them. It may be more concerned about the content and the semantics instead of the whole structure at this time.

A recognized problem with XML is that it is only syntax and does not carry the semantics of the data. However, in relational databases, the semantic constraints have been proved useful in making a good design, update anomaly prevention, efficient storage and access, and query optimization, etc.

In response to the problem, a lot of papers about XML constraints (Buneman, Fan, Siméon & Weinstein 2001, Buneman, Davidson, Fan, Hara & Tan 2001a, Fan, Schwenzer & Wu 2001, Fan & Siméon 1999, Thompson et al 2001) have been published to bring constraints to XML. Among them, the key constraint proposed in the paper of (Buneman, Davidson, Fan, Hara & Tan 2001a) plays an important role and it is independent of DTDs (Fan & Libkin 2001) or XML-Schema (Thompson et al 2001).

Although XML keys are proposed, the relationship between XML keys and relation keys is still unknown. Sometimes problems may appear to a data consumer. For example, while it imports XML data into a relational database with an existing design, conflicts may occur when the relational schema  $\mathbf{R}$  is incompatible with the semantic constraints of the XML data.

```
<db>
  <book isbn="7-111-07526-9">
    <title>Networks</title>
    <author>Larry Peterson</author>
    <chapter number="1">
      <name>Foundation</name>
      <section number="1"><name>TCP/IP</name></section>
      <section number="2"><name>ATM</name></section>
    </chapter>
    <chapter number="9">
      <name>Applications</name>
    </chapter>
  </book>
  <book isbn="7-111-06710-X">
    <title>Networks</title>
    <author>Shary Zhou</author>
    <chapter number="1">
      <name>Introduction</name>
    </chapter>
    <chapter number="9">
      <name>Applications</name>
    </chapter>
  </book>
</db>
```

In order to illustrate the problem, let us take an example first. Suppose the XML document above is being exchanged and the schema of one relation is predefined as `book-chapter(isbn, title,`

number, name). The relation after transformation is shown in figure 1.

If the data consumer specifies (title, number) as the key of the relation in advance, violations will emerge when importing because two distinct tuples agree on the key (“Networks”, “1”). However, if the data consumer specifies (isbn, number) in advance as the key, transformation can be done successfully.

isbn	title	number	name
7-111-07526-9	Networks	1	Foundation
7-111-07526-9	Networks	9	Applications
7-111-06710-X	Networks	1	Introduction
7-111-06710-X	Networks	9	Applications

Figure 1: A sample relation

In relational data, mapping constraints through views has been well-studied. But in XML data, few papers have considered mapping constraints. The paper of (Lee & Chu 2000) presents an algorithm CPI to derive constraints from DTDs instead of XML keys. The paper of (Davidson, Fan & Hara 2001) proposes a simple transformation language and develops a PTIME algorithm to determine whether a given FD can be reasoned about by a set of XML keys as long as the transformation is given. Based on the idea, the algorithm can be naturally generalized to derive a complete set of FDs from a set of XML keys. But the algorithm for computing a complete set of FDs has a high complexity and some restrictions.

In our paper, we develop a heuristic algorithm to compute a complete set of relation keys instead of FDs, given a transformation from XML data to a relational database of schema  $\mathbf{R}$  and a set of XML keys. It can be used to derive relation keys from the semantic constraints of the XML data, and then determine whether the predefined schema  $\mathbf{R}$  of the relational database is compatible or not before the XML data is imported.

**Contributions.** In short, the main contributions of the paper are the following:

- We present the definition of a relation tree which is constructed to represent a well-formed rule on a relation  $R$ . It can be used in the process of populating relations and the process of deriving keys later.
- Given a transformation  $\sigma$  expressed in the transformation language, we show how XML data is imported into the relational database in practice. The transformed relations may violate the basic definition of a relation because of “set-valued attributes” in XML data. Deriving keys is meaningless to this case.
- We present an algorithm for deriving relation keys from XML keys based on relation trees.

**Organization.** The remainder of the paper is organized as follows. Section 2 reviews some basic conceptions about XML keys and constraints. Section 3 introduces the transformation language. Some conceptions about relation trees, as well as the process of populating relations, are also described in this section. Section 4 illustrates algorithms to determine the problem of deriving relation keys. At last we summarize this paper in section 5.

## 2 XML keys and constraints

First of all, we start by reviewing the XML tree model and keys for XML, as well as path expressions that

are used to define XML keys.

### 2.1 XML tree model

The format of XML data is hierarchical and an XML document can be modeled as a tree (Apparao et al 1998). Figure 2 shows the tree model of the sample document. An XML document (or an XML tree) has three types of nodes: element node ( $E$  node), which has a label only; attribute node ( $A$  node), which has a label and carries text; text node ( $S$  node), which just carries text. We use  $\mathbf{E}$  to denote the set of element labels and use  $\mathbf{A}$  to denote the set of attribute labels. A singleton set  $\{\mathbf{S}\}$  denotes text (PCDATA).

**Definition 2.1:** An XML Tree is defined to be  $T = (V, lab, ele, att, val, id, root)$ , where

- $V$  is a finite set of nodes;
- $lab$  maps each node  $v$  to  $\mathbf{E} \cup \mathbf{A} \cup \{\mathbf{S}\}$ . And  $v$  is an element iff  $lab(v) \in \mathbf{E}$ ;  $v$  is an attribute iff  $lab(v) \in \mathbf{A}$ ;  $v$  is a text node iff  $lab(v) = \mathbf{S}$ ;
- $ele$  maps each element node  $v$  to a list of element nodes and text nodes  $[v_1, \dots, v_n]$  in  $V$ . If  $v$  is an attribute node or a text node,  $ele$  is undefined. In essence,  $ele(v)$  defines the element and text children of the element node  $v$ ;
- $att$  maps each element node  $v$  to a set of attribute nodes  $\{v_1, \dots, v_m\}$  in  $V$ . If  $v$  is an attribute node or a text node,  $att$  is undefined.  $att(v)$  defines the attribute children of the element node  $v$ ;
- $val$  maps each attribute node or text node  $v$  to a string. For element nodes,  $val$  is undefined;
- $id$  assigns a unique identity to each node;
- $root$  is a distinguished and unique node. It denotes the root of  $T$  and  $root \in V$ . We use  $r$  to denote the label of  $root$ , i.e.,  $lab(root) = r$ .

□

**Definition 2.2:** Two nodes  $v_1$  and  $v_2$  are *value equal*, denoted by  $v_1 =_v v_2$ , iff the following conditions are satisfied:

- $lab(v_1) = lab(v_2)$ ;
- if  $v_1, v_2$  are  $A$  nodes or  $S$  nodes,  $val(v_1) = val(v_2)$ ;
- if  $v_1, v_2$  are  $E$  nodes, then 1) for any  $a_1 \in att(v_1)$ , there exists  $a_2 \in att(v_2)$  such that  $a_1 =_v a_2$  and vice versa; and 2) if  $ele(v_1) = [v_{11}, v_{12}, \dots, v_{1n}]$ ,  $ele(v_2) = [v_{21}, v_{22}, \dots, v_{2n}]$  and for all  $i \in \{1, 2, \dots, n\}$ ,  $v_{1i} =_v v_{2i}$ .

□

In the tree shown in figure 2, the node of **chapter 9** under the first book is value equal to the node of **chapter 9** under the second book.

Note that  $=_v$  (*value equality*) is used to distinguish from  $=$  (*node equality*). We say two nodes  $v_1$  and  $v_2$  are node equal, denoted by  $v_1 = v_2$ , iff they have the same node identity.

### 2.2 Path expressions

Because the regular expressions (Abiteboul et al. 1997, Clark & DeRose 1999) are so powerful and complicated that they are not suitable for defining XML keys, they are made a little simplification to satisfy the demand.

Two types involved here are shown in the following table.  $PL_s$  denotes *simple path expression*, and  $PL$  denotes *regular path expression*.

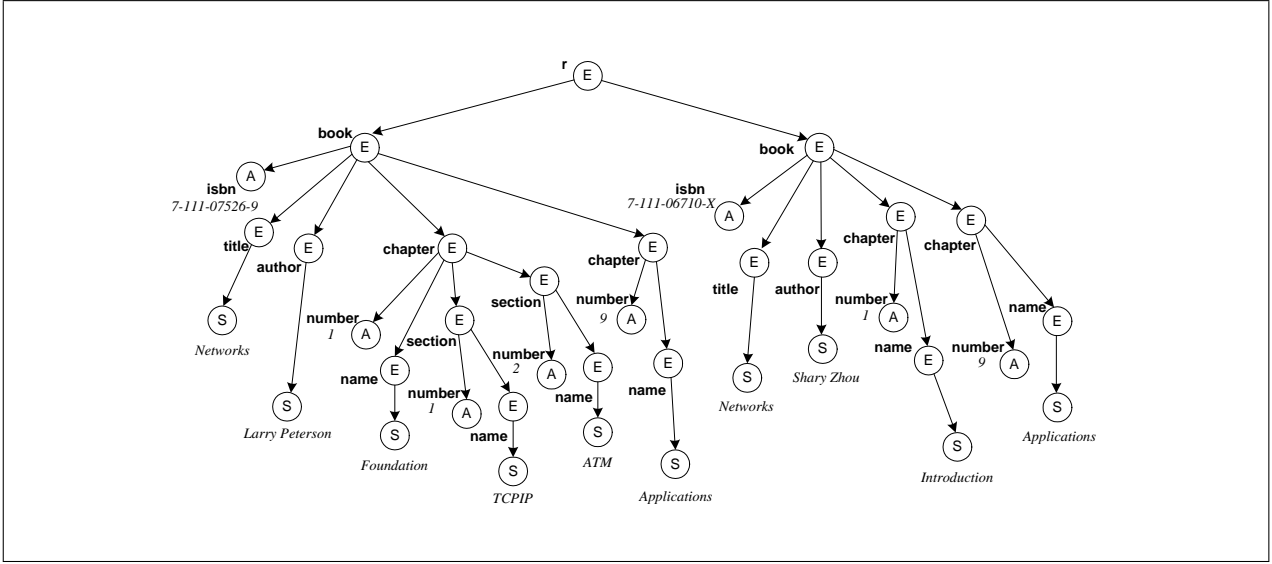


Figure 2: An XML tree

Path expression	Syntax
$PL_s$	$\rho ::= \epsilon \mid l.\rho$
$PL$	$P ::= \epsilon \mid l \mid P.P \mid _*$

where  $l$  is a label in  $\mathbf{E} \cup \mathbf{A} \cup \{\mathbf{S}\}$ ,  $\epsilon$  denotes an empty path,  $.$  denotes concatenation of two path expressions, and  $_*$  is a combination of wild card matching any node label and Kleene closure.

A path defined by a simple path expression is called a *simple path*. `book.chapter.name` is a simple path.

A path defined by a regular path expression is called a *regular path*. For example, `book._*.name` is a regular path. Especially, a simple path can be regarded as a regular path. For example, `book.chapter.name` is not only a simple path but also a regular path.

We use  $\rho \in P$  to denote that  $\rho$  is in the regular path defined by regular path expression  $P$  of  $PL$ . For example, `book.chapter.name`  $\in$  `book._*.name`. Extremely, `book.chapter.name`  $\in$  `book.chapter.name`.

We say that node  $v_2$  in  $T$  is *reachable* from node  $v_1$  by following path  $\rho$  of  $PL_s$ , denoted by  $T \models \rho(v_1, v_2)$ , iff  $v_1 = v_2$  if  $\rho = \epsilon$ , and if  $\rho = \rho'.l$ , then there exists node  $v$  in  $T$  such that  $T \models \rho'(v_1, v)$  and  $v_2$  is a child of  $v$  with label  $l$ .

Similarly, node  $v_2$  in  $T$  is *reachable* from node  $v_1$  by following path  $P$  of  $PL$ , denoted by  $T \models P(v_1, v_2)$ , iff there exists a path  $\rho \in P$  such that  $T \models \rho(v_1, v_2)$ .

Let  $v$  be a node in  $T$ . We use  $v[\rho]$  to denote the set of nodes in  $T$  that can be reached by following the path expression  $\rho$  from  $v$ . Namely,  $v[\rho] = \{v' \mid T \models \rho(v, v')\}$ . Moreover,  $v[P] = \{v' \mid T \models P(v, v')\}$ .

The *value intersection* of  $v_1[P]$  and  $v_2[P]$ , denoted by  $v_1[P] \cap_v v_2[P]$ , is defined by:

$$v_1[P] \cap_v v_2[P] = \{(z, z') \mid \exists \rho \in P, z \in v_1[\rho], z' \in v_2[\rho], z =_v z'\}.$$

$v_1[P] \cap_v v_2[P]$  consists of node pairs that are value equal and are reachable by following the same simple path in the regular path defined by regular path expression  $P$  starting from  $v_1$  and  $v_2$ , respectively.

### 2.3 XML keys

**Definition 2.3:** A *key constraint*  $\varphi$  for XML is an expression of the form

$$(Q, (Q', \{P_1, \dots, P_k\})),$$

where  $Q, Q'$  are expressions of  $PL$  and  $P_i$  are expressions of  $PL_s$  such that for all  $i \in \{1, \dots, k\}$ ,  $Q.Q'.P_i$  is a valid path expression.  $Q$  is called the *context path*,  $Q'$  is called the *target path*, and  $P_1, \dots, P_k$  are called the *key paths*.  $\square$

If  $Q = \epsilon$ , it is an absolute key and can be abbreviated to  $(Q', \{P_1, \dots, P_k\})$ . Otherwise, it is a relative key.

A key  $\varphi = (Q, (Q', \{P_1, \dots, P_k\}))$  specifies as follows: (1) the context path  $Q$ , starting from the root of an XML tree  $T$ , identifies a set of nodes  $\llbracket Q \rrbracket$ ; (2) for each node  $v \in \llbracket Q \rrbracket$ ,  $\varphi$  defines an absolute key  $(Q', \{P_1, \dots, P_k\})$  that is to hold in the subtree rooted at  $v$ ; specifically,

- the target path  $Q'$  identifies a set of nodes  $v[\llbracket Q' \rrbracket]$  in the subtree, referred to as the *target set*;
- the key paths  $P_1, \dots, P_k$  identify nodes in the target set. That is, for each  $v' \in v[\llbracket Q' \rrbracket]$ , the values of the nodes reached by following the key paths from  $v'$  uniquely identify  $v'$  in the target set.

**Definition 2.4:** Let  $\varphi = (Q, (Q', \{P_1, \dots, P_k\}))$  be a key. An XML tree  $T$  *satisfies*  $\varphi$ , denoted by  $T \models \varphi$ , iff for any  $v$  in  $\llbracket Q \rrbracket$  and any  $v_1, v_2$  in  $v[\llbracket Q' \rrbracket]$ , if for all  $i \in \{1, \dots, k\}$  there exists a path  $\rho \in P_i$  and nodes  $x \in v_1[\rho], y \in v_2[\rho]$  such that  $x =_v y$ , then  $v_1 = v_2$ . That is,

$$\left( \bigwedge_{1 \leq i \leq k} v_1[P_i] \cap_v v_2[P_i] \neq \emptyset \rightarrow v_1 = v_2 \right) \quad \square$$

Suppose  $\varphi = (_*.book, (chapter, \{number\}))$  is one of XML keys of figure 2. Roughly speaking, for any `book` in  $\llbracket _*.book \rrbracket$  and any two `chapters` of its, if they agree on the `number`, they will be the same chapter. Through verification, we know  $T \models \varphi$ . If we specify another key  $\varphi' = (_*.chapter, \{number\})$  of  $T$ ,  $T$  will not satisfy  $\varphi'$  because there exist two distinct `chapters` that agree on the `number`.

### 3 Transformation language from XML to relational databases

We review the transformation language from XML to a relational database in this section. Simple as the language is, it is powerful to specify many common transformations in applications. And it has been

proved that some extensions to the language are so powerful that deriving relation keys from XML keys becomes undecidable.

### 3.1 Transformation language

Let  $x$  be a node in an XML tree  $T$ . We introduce another extension of  $val$ . The value function of  $x$ , denoted by  $value(x)$ , returns a string representing the pre-order traversal of the subtree rooted at  $x$ . More precisely,  $value(x)$  is defined as follows:

1. If  $x$  is an  $A$  node or an  $S$  node, then  $value(x) = val(x)$ ;
2. If  $x$  is an  $E$  node, then  $value(x)$  is a string “( $| lab(x_1) | \text{“.”} | value(x_1) | \text{“,”} | \dots | lab(x_n) | \text{“.”} | value(x_n) | \text{“,”} | lab(a_1) | \text{“.”} | value(a_1) | \text{“.”} | \dots | lab(a_m) | \text{“.”} | value(a_m) | \text{“} \text{”}$ )”, where  $ele(x) = [x_1, \dots, x_n]$ ,  $att(x) = \{a_1, \dots, a_m\}$ , and “ $|$ ” denotes string concatenation.

We restrict that attributes are alphabetically ordered by their labels. For example, if the node  $x$  is the second `chapter` element under the first `book` element, then  $value(x) = (name : (S : Applications), number : 9)$ .

In the following sections, we regard  $id$  as an especial case of the function  $value$  by imagining that there exists an attribute `#id` under node  $x$  and  $id(x)$  is equal to  $value(x.\#id)$ , regardless of whether  $x$  is an element, an attribute or a text node.

**Definition 3.1:** A rule path  $X$  is defined as follows: (1)  $X \leftarrow Y.P$  where  $Y$  is also a rule path,  $P$  is a path expression of  $PL$  and  $Y.P$  is a valid path expression; (2)  $x_r$ , which is a distinguished variable and stands for the root of an XML tree, is a rule path.  $\square$

We say that a rule path  $X$  is *well-formed* iff  $X$  is *connected* to the root  $x_r$ ; that is,  $X \leftarrow x_r.P$ , or  $X \leftarrow Y.P$  and  $Y$  is connected to the root  $x_r$ .

For example,  $x_r.book._*.chapter$  is a well-formed rule path.

Actually, a well-formed rule path  $X$  identifies a set of nodes, denoted by  $X^*$ . Each node in  $X^*$  satisfies the path expression  $X$ , i.e., (1) if  $X \leftarrow x_r.P$ , then  $X^* = \llbracket P \rrbracket$ ; (2) if  $X \leftarrow Y.P$ , then  $X^* = y\llbracket P \rrbracket$  where  $y \in Y^*$ . For example, in figure 2,  $x_r.book._*.chapter^*$  is the set including the two `chapter` subelements of the first book and the two `chapter` subelements of the second book.

**Definition 3.2:** Let  $X$  be a well-formed rule path. A rule on an attribute  $l$  of a relation  $R$  is  $rule(R, l) = l : value(X) | l : id(X)$ , where  $value(X) = [value(x_1), \dots, value(x_n)]$ ,  $id(X) = [id(x_1), \dots, id(x_n)]$  and  $X^* = \{x_1, \dots, x_n\}$ .  $\square$

For example, a list [7-111-07526-9, 7-111-06710- $X$ ] can be obtained by computing the rule  $rule(book, isbn) = isbn : value(X)$ , where  $X = x_r._*.book.isbn$ .

**Definition 3.3:** A rule on a relation  $R$  is  $Rule(R) = \{rule(R, l) | l \in att(R)\}$ . In  $Rule(R)$ , some rule paths are used to extract data from XML for the attributes, referred to as *valuable rule paths*. Others are used to navigate the XML data, referred to as *temporary rule paths*. And for each attribute  $l \in att(R)$ ,  $rule(R, l)$  fills the attribute of the relation with the actual data.  $\square$

A rule on a relation  $R$  is *well-formed* iff

1. for any  $X_1 \leftarrow Y.P_1$  and  $X_2 \leftarrow Y.P_2$  in the rule,  $P_1$  and  $P_2$  do not share a common prefix, i.e., there exist no  $PL$  expressions  $P, P'_1, P'_2$  such that  $P_1 = P.P'_1$  and  $P_2 = P.P'_2$  and  $P \neq \epsilon$ ; moreover, for any  $X \leftarrow Y.P$ ,  $P$  is a simple path (without “ $_*$ ”) unless  $Y$  is  $x_r$ ; in addition, for any  $X_1 \leftarrow$

$Y.P_1$  and  $X_2 \leftarrow Y.P_2$  in the rule, where  $X_1$  and  $X_2$  are distinct, it is not allowed that  $P_1$  and  $P_2$  are not simple paths, and  $Y$  is  $x_r$ ;

2. for any  $rule(R, l_1)$  and  $rule(R, l_2)$ , if  $l_1$  and  $l_2$  are different attributes, they are different.

The first condition ensures that it groups two nodes under a common ancestor into the same tuple in a relation; and moreover, it does not group “unrelated” data in the same relation to preserve the structure of the data. Adding the restriction  $_*$  is to exclude the nodes which are “far away” from each other. The second condition prevents populating different attributes of a relation with the same values, which does not make much sense in practice.

For example, the above relation `book-chapter` in figure 1 could be specified as follows:

```
Rule(book-chapter) = { isbn : value(X2), title : value(X3),
number : value(X5), name : value(X6) }
X1 ← x_r._*.book, X2 ← X1.isbn,
X3 ← X1.title.S, X4 ← X1.chapter,
X5 ← X4.number, X6 ← X4.name.S
```

In  $Rule(book-chapter)$ ,  $X_1$  and  $X_4$  are temporary rule paths while others are valuable rule paths.

**Definition 3.4:** Given a relational schema  $\mathbf{R}$ , a transformation from XML data to a relational database of schema  $\mathbf{R}$  is  $\sigma = (Rule(R_1), \dots, Rule(R_n))$ , where  $R_1, \dots, R_n$  are the relations in the relational database.  $\square$

The transformation from XML data to a relational database of schema  $\mathbf{R}$  is composed of a collection of rules from XML data to relations in the relational database. A transformation  $\sigma$  is *well-formed* iff each rule on the corresponding relation is well-formed. We consider well-formed transformations only.

### 3.2 Relation tree model and process of populating relations

**Definition 3.5:** Let  $Rule(R) = \{rule(R, l) | l \in att(R)\}$  be a well-formed rule on  $R$ . A relation tree is defined to be  $RT = (V', x_r, x_b, label, parent, children, leaf, attribute)$ , where

- $V'$  is a finite set of nodes;
- $x_r$ , which is a distinguished and unique node, stands for the root of the relation tree;
- $x_b$ , which is a distinguished and unique node, stands for the unique child of  $x_r$ . The path from  $x_r$  to  $x_b$  may be a regular path (with “ $_*$ ”);
- $label$  maps each node  $v'$  to a path expression. Specially,  $label(x_r) = r$ . If  $v'$  is  $x_b$ ,  $label$  maybe maps it to a regular path expression  $P$  of  $PL$ . Otherwise,  $label$  maps  $v'$  to a simple path expression  $\rho$  of  $PL_s$ ;
- $parent$  maps each non-root node  $v'$  to its parent node. For the root  $x_r$ ,  $parent$  is undefined;
- $children$  maps each interior node  $v'$  to a set of its children. Specially,  $children(x_r) = \{x_b\}$ . For leaves,  $children$  is undefined;
- $leaf$  maps each node  $v'$  to a boolean value. It returns true iff  $v'$  is a leaf. Otherwise, it returns false;
- $attribute$  maps each leaf  $v'$  to a string  $l$  which denotes one of attributes of the relation  $R$ . For interior nodes, it is undefined.

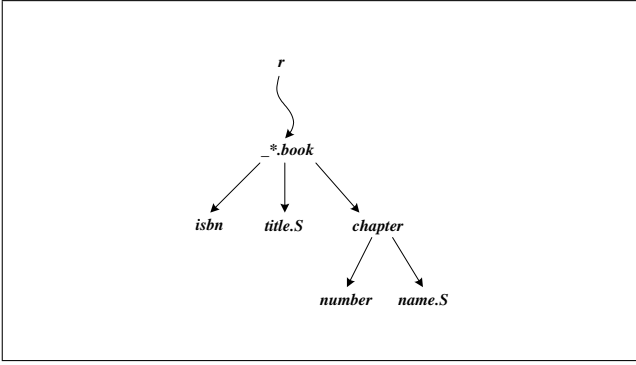


Figure 3: A relation tree

□

We say that node  $v'_2$  in  $RT$  is *reachable* from node  $v'_1$  by following path  $P$  of  $PL$ , denoted by  $RT \models P(v'_1, v'_2)$ , iff  $v'_1 = v'_2$  if  $P = \epsilon$ , and if  $P = P'.P''$ , then there exists node  $v'$  in  $RT$  such that  $RT \models P'(v'_1, v')$ ,  $v'_2 \in children(v')$  and  $label(v'_2) = P''$ . It is worth noting that  $P$ ,  $P'$  and  $P''$  are just made up of the labels of the nodes in  $RT$  through concatenation here (except  $\epsilon$ ). That is,  $P$ ,  $P'$  and  $P''$  are all simple path expressions unless they have passed node  $x_b$ .

Given a well-formed rule  $\text{Rule}(R)$  on a relation  $R$ , there always exists a relation tree that represents the rule. The relation tree of  $\text{Rule}(\text{book} \rightarrow \text{chapter})$  above is given in figure 3. In some degree, the relation tree  $RT$  is a rule to simplify an XML tree  $T$  into a simple one  $T'$  by just preserving the nodes in  $T$  which are “useful” in  $RT$ .

**Definition 3.6:** Let  $T$  be an XML tree and let  $RT$  be the relation tree. For any node  $v$  in  $T$ , there always exists one and only one simple path  $\rho$  such that  $T \models \rho(\text{root}, v)$ . For any node  $v'$  in  $RT$ , there also exists one regular path  $P$  such that  $RT \models P(x_r, v')$ . If  $\rho \in P$ , we say  $v'$  is the *corresponding* node of  $v$ , denoted by  $map(v) = v'$  where  $v \in V$  and  $v' \in V'$ . Contrarily, the set of the *corresponding* nodes of  $v'$  is defined as  $map^{-1}(v') = \{v \mid map(v) = v'\}$ .  $v$  is one of the corresponding nodes of  $v'$  iff  $v \in map^{-1}(v')$ . □

For any two nodes  $v_1$  and  $v_2$  in the set  $map^{-1}(v')$  where  $v'$  is a node in  $RT$ , they have the same label in  $T$ , i.e.,  $lab(v_1) = lab(v_2)$  because  $v'$  stands for the same type of nodes.

Next we show how XML data is imported into relations in practical applications. Given an XML tree  $T$  and a well-formed rule  $\text{Rule}(R)$  on a relation  $R$ ,  $R$  can be populated in the following way:

1. construct the relation tree  $RT$  to represent the rule  $\text{Rule}(R)$ ;
2. change  $T$  into  $T'$  according to  $RT$  by preserving the nodes in  $T$  that are used in  $RT$  only;
3. traverse  $T'$  in the post-order, for any node  $v$  in  $T'$ , (1) if there exists a leaf  $v'$  in  $RT$  such that  $v \in map^{-1}(v')$ , then extract the actual data from  $v$  by  $value(v)$ , store the value into a temporary relation which contains one attribute  $attribute(v')$  and name the relation as  $lab(v)$ ; (2) or else, if  $v$  is one of the corresponding nodes of  $x_b$ , if  $R$  exists, then execute the operation  $union \cup$  between  $R$  and the relation  $lab(v)$  and name the result relation as  $R$  again; if  $R$  does not exist, then rename the relation  $lab(v)$  as  $R$ ; (3) or else, if  $v$  has only one child, then rename the child relation whose name is the label of the child as  $lab(v)$ . If  $v$  has more than one child, execute the

operation *union*  $\cup$  between the children relations which have the same name at first, and then execute the operation *cross product*  $\times$  between the children relations which have different names and name the final relation as  $lab(v)$ ;

4. if the root of  $T'$  is reached, the process terminates.

In fact, the relation is shown in figure 1 for the above example.

Because there exist “set-valued attributes” in XML data and the data is stored through redundancy, it cannot be avoided that two same tuples appear in the transformed relation though it is seldom. In the case, the relation violates the basic definition of a relation and it is impossible to specify keys on the relation. For example, in the XML tree of figure 2, if there are two **title** nodes with the same value “Networks” under the first **book** node, there exist at least two same tuples (e.g. (“7-111-07526-9”, “Networks”, 1, “Foundation”)) in the relation. We ignore this meaningless case in the paper.

## 4 Deriving relation keys from XML keys

### 4.1 Problem statement

As mentioned above, there exists a transformation  $\sigma$  from XML data to a relational database of schema  $\mathbf{R}$  where for any XML Tree  $T$ ,  $\sigma(T)$  is an instance of  $\mathbf{R}$  which is a collection of transformed relations. The problem is, given a transformation  $\sigma$  from XML data to a relational database of schema  $\mathbf{R}$ , and an XML tree  $T$  satisfying a set  $\Sigma$  of XML keys, whether the complete set  $\Gamma$  of relation keys can be found such that  $\sigma(T)$  satisfies  $\Gamma$ . More specifically, the problem can be stated as:

- INPUT: A transformation  $\sigma$  from XML data to a relational database of schema  $\mathbf{R}$ , a set  $\Sigma$  of XML keys.
- OUTPUT: A complete set  $\Gamma$  of relation keys.

For any XML tree  $T$ , if  $T$  satisfies  $\Sigma$ ,  $\sigma(T)$  will satisfy  $\Gamma$ .

### 4.2 Algorithm description

As mentioned early,  $\sigma(T)$  contains some relations which are transformed according to rules in the transformation  $\sigma$ . In order to illustrate the problem simply, we start to analyze a rule  $\text{Rule}(R)$  in  $\sigma$  only. The algorithm for deriving relation keys on  $R$  from  $\Sigma$  is shown in figure 4.

The whole algorithm is a bottom-up approach. The algorithm works as follows: at first, special XML keys  $(\_*, \{\#id\})$  and  $(\_*, (\#id, \{\}))$  should be added into  $\Sigma$  since any node can be identified by its **#id** and the **#id** of each node is unique according to the definition of an XML tree. For the rule  $\text{Rule}(R)$ , the relation tree  $RT$  is constructed to represent the rule. Then  $RT$  is traversed in the post-order. Suppose the node visited currently is  $v'$ . An XML key  $(P(x_r, parent(v')), (label(v'), s))$  is constructed to check (see **Algorithm constructkey** shown in figure 5 for details). If it is implied by  $\Sigma$  (Buneman, Davidson, Fan, Hara & Tan 2001b), then the set  $s$  of the key paths is added into a variable  $v'.keys$ , where the key set  $v'.keys$  of  $v'$  is used to cache the key paths and it is an empty set initially.  $s$  is computed in the following way:

1. If  $v'$  is an interior node, then for each  $c' \in children(v')$ , if  $c'.keys$  is  $\{\{\}\}$ ,  $label(c')$  is added into a set  $S'$  ( $\{\}$  initially); or else, each set in

```

Algorithm derivation( $\Sigma$ , Rule( $R$ ))
begin
1.  $\Sigma := \Sigma \cup \{(-*, \{\#id\}), (-*, (\#id, \{\}))\}$ ;
2.  $RT := \text{ConstructRT}(\text{Rule}(R))$ ;
3.  $v' := \text{GetFirstNode}(RT)$ ;
4. while ( $v' \neq x_r$ ) do
5.   begin
6.      $S := \{\{\}\}$ ;
7.     if (leaf( $v'$ )) then
8.        $S' := \{\epsilon\}$ ;
9.     else
10.    begin
11.       $S' := \{\}$ ;
12.      for each  $c' \in \text{children}(v')$  do
13.        if ( $c'.keys \neq \{\{\}\}$ ) then
14.          candidate( $c', S$ );
15.        else
16.           $S' := S' \cup \{\text{label}(c')\}$ ;
17.        end;
18.      power( $S'$ );
19.      check( $v', S, S', RT$ );
20.      if ( $v'.keys = \{\}$ ) then
21.        return  $\{\}$ ;
22.      if ( $\text{Succ}(v', RT) = x_r$ ) then
23.        begin
24.           $\Gamma := \{\}$ ;
25.          for each  $s \in v'.keys$  do
26.             $\Gamma := \Gamma \cup \{\text{back}(s, \text{Rule}(R))\}$ ;
27.          end;
28.           $v' := \text{Succ}(v', RT)$ ;
29.        end;
30.      return  $\Gamma$ ;
end.

```

Figure 4: An algorithm for deriving relation keys from XML keys

- $S(\{\{\}\})$  initially), referred to as the *compulsory* candidate set of  $v'$ , is combined with each set in  $c'.keys$  (see **Algorithm candidate** shown in figure 6 for details). If  $v'$  is a leaf, add an empty path  $\epsilon$  into the set  $S'$  only;
- each path in  $S'$  is combined with one another and  $S'$  is changed into a power set, referred to as the *optional* candidate set of  $v'$  (see **Algorithm power** shown in figure 7 for details);
  - $s$  is computed by combining each set in  $S$  with each set in  $S'$ , where each set  $s'$  in  $S'$  should be picked up one by one according to the cardinality of  $s'$  in the ascending order (see **Algorithm check** shown in figure 8 for details). But not all of sets in the optional candidate set should be selected into  $s$  because if  $k$  is a key of a relation, then for each  $k' \supseteq k$ ,  $k'$  is also a key, which does not make much sense.

```

Algorithm constructkey( $v', s, RT$ )
begin
1.  $w := \text{parent}(v')$ ;
2.  $key := (P(x_r, w), (\text{label}(v'), s))$ ;
3. return  $key$ ;
end.

```

Figure 5: An algorithm for constructing an XML key

Additionally, given a rule  $\text{Rule}(R)$ , the function  $\text{ConstructRT}$  is to construct the relation tree  $RT$  to represent the rule. Given a tree  $RT$ , the function

```

Algorithm candidate( $c, S$ )
begin
1.  $S_1 := \{\}$ ;
2. for each  $k \in c.keys$  do
3.   begin
4.     for each  $path \in k$  do
5.        $path := \text{label}(c).path$ ;
6.     for each  $s \in S$  do
7.       begin
8.          $s_1 := s \cup k$ ;
9.          $S_1 := S_1 \cup \{s_1\}$ ;
10.      end;
11.    end;
12.   $S := S_1$ ;
13. return;
end.

```

Figure 6: An algorithm for updating  $S$

```

Algorithm power( $S'$ )
begin
1.  $S_0 := \{\}$ ;
2. for any  $S \subseteq S'$  do
3.    $S_0 := S_0 \cup \{S\}$ ;
4.  $S' := S_0$ ;
5. return;
end.

```

Figure 7: An algorithm for computing the power set of  $S'$

$\text{GetFirstNode}$  is to get the first node in the post-order traversal. Given a tree  $RT$  and a node  $v'$  in it, the function  $\text{Succ}$  returns the next node of  $v'$  in the post-order traversal. And given a rule  $\text{Rule}(R)$  and a set of key paths, the function  $\text{back}$  returns the set of attributes mapped from the key paths. We do not describe these functions in this paper.

For example, a transformation  $\sigma$  from the XML data in figure 2 to a relational database is specified as follows:

$$\sigma = (\text{Rule}(\text{book-chapter}), \text{Rule}(\text{chapter-section}))$$

$$\text{Rule}(\text{book-chapter}) = \{\text{isbn} : \text{value}(X_2), \text{title} : \text{value}(X_3), \text{author} : \text{value}(X_4), \text{cid} : \text{id}(X_5), \text{cnumber} : \text{value}(X_6), \text{cname} : \text{value}(X_7)\}$$

$$\begin{aligned} X_1 &\leftarrow x_r..*.book & X_2 &\leftarrow X_1.isbn \\ X_3 &\leftarrow X_1.title.S & X_4 &\leftarrow X_1.author.S \\ X_5 &\leftarrow X_1.chapter & X_6 &\leftarrow X_5.number \\ X_7 &\leftarrow X_5.name.S \end{aligned}$$

$$\text{Rule}(\text{chapter-section}) = \{\text{cid} : \text{id}(Y_1), \text{sid} : \text{id}(Y_2), \text{snumber} : \text{value}(Y_3), \text{sname} : \text{value}(Y_4)\}$$

$$\begin{aligned} Y_1 &\leftarrow x_r..*.book.chapter & Y_2 &\leftarrow Y_1.section \\ Y_3 &\leftarrow Y_2.number & Y_4 &\leftarrow Y_2.name.S \end{aligned}$$

Then the corresponding relation trees are given in figure 9. A set  $\Sigma$  of XML keys is specified as follows:

- $(_*.book, \{\text{isbn}\})$   
Within the context of the entire tree, any book element is identified by its **isbn** attribute;
- $(_*.book, (_*.chapter, \{\text{number}\}))$   
Within the context of the subtree rooted at any book element, any **chapter** element is identified by its **number** attribute;
- $(_*.chapter, (\text{section}, \{\text{number}\}))$   
Within the context of the subtree rooted at any

```

Algorithm check( $v', S, S', RT$ )
begin
1.  $v'.keys := \{\}$ ;
2. for each  $s \in S$  do
3.   begin
4.      $n := 0$ ;
5.      $S_0 := S'$ ;
6.     for each  $s' \in S_0$  and  $|s'| = n$  do
7.       begin
8.          $k := \text{constructkey}(v', s \cup s', RT)$ ;
9.         if  $(\Sigma \models k)$  then
10.        begin
11.           $v'.keys := v'.keys \cup \{s \cup s'\}$ ;
12.          for any  $s'' \in S_0$  and  $s' \subseteq s''$  do
13.             $S_0 = S_0 - \{s''\}$ ;
14.          end;
15.           $n := n + 1$ ;
16.        end;
17.      end;
18.    return;
end.

```

Figure 8: An algorithm for checking

chapter element, any section element which is under the chapter element immediately is identified by its number attribute;

- $(\_*.book, (isbn, \{\}))$ ,  $(\_*.book, (title, \{\}))$ , and  $(\_*.book, (author, \{\}))$

Within the context of the subtree rooted at any book element, the isbn attribute, the title element and the author element of the book element are unique;

- $(\_*.chapter, (number, \{\}))$   
 $(\_*.chapter, (name, \{\}))$

Within the context of the subtree rooted at any chapter, the number attribute and the name element are unique;

- $(\_*.section, (number, \{\}))$   
 $(\_*.section, (name, \{\}))$

Within the context of the subtree rooted at any section, the number attribute and the name element are also unique.

- $(\_*, (S, \{\}))$

For any element, if it has text children, the number of its text children is at most one.

The algorithm finds the first node (node isbn) in the post-order traversal of the left relation tree and then constructs an XML key:  $(\_*.book, (isbn, \{\}))$ . Because the key can be implied by  $\Sigma$ ,  $\{\}$  is added into the key set of the node isbn. Next, the algorithm visits the next node (node title.S) and then it constructs a key:  $(\_*.book, (title.S, \{\}))$ . Similarly,  $\{\}$  is added into the key set of the node title.S.

When node chapter is visited, there is only one element  $\{\}$  in the compulsory candidate set of the node chapter because each key set of its children is  $\{\{\}\}$ , and the optional candidate set of the node chapter contains  $2^3$  sets:

$$\{\}, \{\#id\}, \{number\}, \{name.S\}, \{\#id, number\}, \{\#id, name.S\}, \{number, name.S\}, \{\#id, number, name.S\}.$$

First an XML key  $(\_*.book, (chapter, \{\}))$  is constructed for checking. But  $\Sigma$  can not imply it. Secondly, another key  $(\_*.book, (chapter, \{\#id\}))$  is constructed. It can be implied by  $\Sigma$  and thus any super set of  $\{\#id\}$  in the optional candidate set need not be picked up for checking. Namely,  $\{\#id, number\}$ ,  $\{\#id, name.S\}$ ,  $\{\#id, number, name.S\}$  are removed

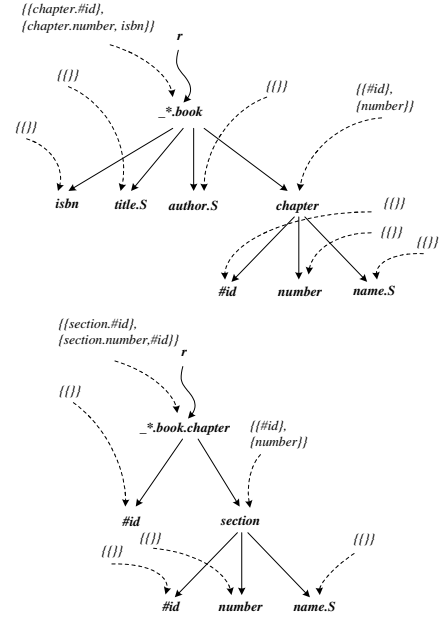


Figure 9: Relation trees

from the optional candidate set of the node chapter. If the algorithm continues like this way, at last the key set of the node chapter is figured out to be  $\{\{\#id\}\}$ .

When node  $\_*.book$  is visited, its compulsory candidate set contains two sets:

$$\{chapter.\#id\}, \{chapter.number\}$$

and its optional candidate set contains  $2^3$  sets:

$$\{\}, \{isbn\}, \{title.S\}, \{author.S\}, \{isbn, title.S\}, \{isbn, author.S\}, \{title.S, author.S\}, \{isbn, title.S, author.S\}.$$

Because the key  $(\epsilon, (\_*.book, \{chapter.\#id\}))$  can be implied,  $\{chapter.\#id\}$  can be added into the key set of the node  $\_*.book$ . Through trials, the key path  $chapter.number$  can not identify the book elements and isbn should be included. That is, the key  $(\epsilon, (\_*.book, \{chapter.number, isbn\}))$  can be implied. Finally, the key set of the node  $\_*.book$  is  $\{\{chapter.\#id\}, \{chapter.number, isbn\}\}$ .

At this time, two keys  $cid$  and  $(cnumber, isbn)$  on the relation book-chapter are computed by the algorithm:

$$\begin{array}{ll} cid \rightarrow isbn, & cid \rightarrow title, \\ cid \rightarrow author, & cid \rightarrow cnumber, \\ cid \rightarrow cname, & (cnumber, isbn) \rightarrow title, \\ (cnumber, isbn) \rightarrow author, & (cnumber, isbn) \rightarrow cid, \\ (cnumber, isbn) \rightarrow cname. & \end{array}$$

Likewise, the keys on the relation chapter-section are  $sid$  and  $(cid, snumber)$ :

$$\begin{array}{ll} sid \rightarrow cid, & sid \rightarrow snumber, \\ sid \rightarrow sname, & (cid, snumber) \rightarrow sid, \\ (cid, snumber) \rightarrow sname. & \end{array}$$

To some cases, if the given set of XML keys is not enough, the set of relation keys is empty. As an example, if an XML key  $(\_*.book, (title, \{\}))$  is absent, the algorithm can not decide whether the title element under a book element is unique or not. So it can not decide  $cid \rightarrow title$  and  $(cnumber, isbn) \rightarrow title$ . The

relation book-chapter has no key finally. In figure 4, **Algorithm derivation** terminates and  $\Gamma$  is empty.

**Proposition 4.1:** *Given an XML tree  $T$  satisfying a set  $\Sigma$  of XML keys and a well-formed rule  $Rule(R)$  on a relation  $R$ , (1) each set in  $\Gamma$  is a candidate key and excluding any attribute in the candidate key makes it not be a key any longer; (2) for any key specified on  $R$ , there exists a set in  $\Gamma$  that is a subset of the key.*  $\square$

*Proof sketch:* By induction the following conclusions can be proved: (a) given a node  $v'$  in the relation tree  $RT$  and a set  $s$  ( $s \in v'.keys$ ) of key paths where each path is from a child of  $v'$  along to a leaf descendant of  $v'$ , for any node  $v''$  which is either a descendant of  $v'$  or  $v'$ , if there are always some key paths in  $s$  such that they can identify  $v''$  within the context of  $parent(v'')$ , then a set  $Y$  of attributes, which is composed of attributes mapped from key paths in  $s$  by using the function **back**, can determine any attribute in the temporary relation  $lab(v)$  ( $v \in map^{-1}(v')$ ); (b) given a node  $v'$  in the relation tree  $RT$  and a set  $Y$  of attributes where each attribute is mapped from a leaf in  $RT$ , if  $Y$  determines any attribute in the temporary relation  $lab(v)$ , then for any node  $v''$  which is either a descendant of  $v'$  or  $v'$ , there always exist some key paths in  $s$  such that they can identify  $v''$  within the context of  $parent(v'')$ , where  $s$  contains those key paths mapped to the attributes in  $Y$ .

After executing the operation union and renaming, the key of the former relation is still the key of the new relation. Thus the conclusions are sound to the operation union and renaming and then we analyze the effect of the operation cross product only.

(a) For a leaf  $v'$  in  $RT$ , if the key path to identify  $v'$  within the context of  $parent(v')$  is empty, the key of the temporary relation  $lab(v)$  is  $attribute(v')$  because the relation contains only one attribute; for an interior node  $v'$  in  $RT$ , for any child  $v''$  of  $v'$ , if there always exist some attributes in  $Y$  such that the set containing those attributes is a key of the temporary relation  $lab(u)$  ( $u \in map^{-1}(v'')$ ), then  $Y$  can determine each attribute in the temporary relation  $lab(v)$  after “ $\times$ ” is executed on the children relations.

(b) For a leaf  $v'$  in  $RT$ , the conclusion is sound because the temporary relation  $lab(v)$  has only one attribute. For an interior node  $v'$  in  $RT$ , if there exists a node  $v''$  in the subtree rooted at  $v'$  such that any set of key paths selected from  $s$  cannot identify  $v''$  within the context of  $parent(v'')$ , then for the set of those attributes which are in  $Y$  and the temporary relation  $lab(u)$  ( $u \in map^{-1}(v'')$ ), any subset cannot become a key of the relation  $lab(u)$ . No matter which operation is executed to create the new relation  $lab(v)$ ,  $Y$  cannot be a key of the relation  $lab(v)$ . Thus, the node  $v''$  does not exist.

According to the above conclusions, the proposition is provable. Additionally, as for (1), any super set of the candidate key is neglected in the algorithm, so the candidate key is minimal and excluding any attribute in the candidate key will make it not be a key any longer.  $\square$

## 5 Conclusion

The key constraint is an essential and critical part of databases, especially in relational databases. As a type of semi-structured data, it is not intact because XML data does duty as relational databases without semantics.

With the development of XML, constraints are proposed in many papers. This paper is mainly to deal with how to map XML keys to relation keys when

importing XML data into a relational database, given a transformation expressed in the transformation language. An algorithm for deriving relation keys from XML keys is described in this paper. If the relational schema has not been predefined before, the set of relation keys must be considered in the schema. If the schema has been predefined, appending data will not succeed unless the set of relation keys does not violate the predefined schema.

The definition of an XML key implicates that descendant nodes identify ancestor nodes while sibling nodes cannot be identified by sibling nodes. Because of the limitation, deriving functional dependencies from XML keys is more difficult and complicated. To make a better algorithm without trial and failure is a part of future work. On the other hand, the algorithm just returns the set of relation keys and does not evaluate the transformation. To make a good transformation automatically according to XML keys is another part of future work.

## References

- Abiteboul, S., Quass, D., McHugh, J., Widom, J. & Wiener, J. L. (1997), ‘The Lorel query language for semistructured data’, *International Journal on Digital Libraries* 1(1), 68–88.
- Apparao et al, V. (1998), ‘Document Object Model (DOM) Level 1 Specification’, W3C Recommendation. <http://www.w3.org/TR/REC-DOM-Level-1/>.
- Bray, T., Paoli, J. & Sperberg-McQueen, C. M. (1998), ‘Extensible Markup Language (XML) 1.0’, W3C Recommendation. <http://www.w3.org/TR/REC-xml/>.
- Buneman, P., Davidson, S., Fan, W., Hara, C. & Tan, W. (2001a), Keys for XML, in ‘WWW’10’.
- Buneman, P., Davidson, S., Fan, W., Hara, C. & Tan, W. (2001b), Reasoning about keys for XML, in ‘Proc. of Int’l Workshop on Database Programming Languages (DBPL)’.
- Buneman, P., Fan, W., Siméon, J. & Weinstein, S. (2001), ‘Constraints for semistructured data and XML’, *SIGMOD Record*.
- Chamberlin et al, D. (2001), ‘XQuery 1.0: An XML Query Language’, W3C Working Draft. <http://www.w3.org/TR/xquery>.
- Clark, J. & DeRose, S. (1999), ‘XML Path Language (XPath)’, W3C Working Draft. <http://www.w3.org/TR/xpath>.
- Davidson, S., Fan, W. & Hara, C. (2001), Propagating XML keys to relations, Technical Report MS-CIS-01-33, University of Pennsylvania.
- Fan, W. & Libkin, L. (2001), On XML integrity constraints in the presence of DTDs, in ‘Proc. of ACM Symposium on Principles of Database Systems (PODS)’, pp. 114–125.
- Fan, W., Schwenzer, P. & Wu, K. (2001), Keys with upward wildcards for XML, in ‘The 12th International Conference and Workshop on Database and Expert Systems Applications’.
- Fan, W. & Siméon, J. (1999), Integrity constraints for XML, in ‘PODS’00’, pp. 23–34.
- Fernandez, M. F., Tan, W. & Suciú, D. (2000), SilkRoute: Trading between relations and XML, in ‘Proc. of Int’l World Wide Web Conf. (WWW)’.

- Florescu, D. & Kossmann, D. (1999), 'A performance evaluation of alternative mapping schemes for storing XML data in a relational database', Tech. Report no. 3684, INRIA.
- Intelligent Systems Research (n.d.), 'XML from databases: ODBC2XML'.  
<http://www.intsysr.com/odbc2xml.htm>.
- Lee, D. & Chu, W. W. (2000), Constraints-preserving transformation from XML document type definition to relational schema, *in* 'Proc. of Int'l Conf. on Conceptual Modeling (ER)'.
- Manolescu, I., Florescu, D. & Kossmann, D. (2001), 'Pushing XML queries inside relational databases', Tech. Report no. 4112, INRIA.
- Oracle (n.d.), 'Using XML in Oracle internet applications'.  
[http://technet.oracle.com/tech/xml/info/htdocs/otnwp/about\\_xml.htm](http://technet.oracle.com/tech/xml/info/htdocs/otnwp/about_xml.htm).
- Shanmugasundaram et al, J. (1999), 'Relational databases for querying XML documents: Limitations and opportunities', *VLDB Journal* pp. 302–314.
- Shanmugasundaram, J., Kiernan, J., Shekita, E. J., Fan, C. & Funderburk, J. (2001), Querying XML views of relational data, *in* 'Proc. of Int'l Conf. on Very Large Databases (VLDB)', pp. 261–270.
- Shanmugasundaram, J., Shekita, E. J., Barr, R., Carey, M. J., Lindsay, B. G., Pirahesh, H. & Reinwald, B. (2001), 'Efficiently publishing relational data as XML documents', *VLDB Journal* **10**(2-3), 133–154.
- Thompson et al, H. (2001), 'XML Schema', W3C Working Draft.  
<http://www.w3.org/XML/Schema>.
- Zhou, A., Lu, H., Zheng, S., Liang, Y., Zhang, L., Ji, W. & Tian, Z. (2001), VXMLR: A Visual XML-Relational Database System, *in* 'Proc. of Int'l Conf. on Very Large Databases (VLDB)'.