Schema Mapping for XML and Relational Data Sharing

Ya-Hui Chang* Feng-Chieh Chiu
Department of Computer Science
National Taiwan Ocean University
yahui@mail.ntou.edu.tw

Wang-Chien Lee
Dept. of Computer Science and Engineering
Penn State University
wlee@cse.psu.edu

Abstract

While XML has emerged as the de facto standard for data representation and exchange on the World-Wide-Web (WWW), relational databases are widely used in enterprises to support critical business operations. Thus, providing interoperability between relational databases and XML data repositories is a very important issue. In this paper, a mapping dictionary in XML format is proposed to capture the necessary information for resolving representational conflicts between relational and XML schemas. Based on this proposal, relational queries can be transformed to XML queries, so that XML data can be easily accessed in SQL. A prototype is built to validate our idea and demonstrate the feasibility of the mapping dictionary proposal.

Keywords: interoperability, mapping dictionary, query transformation, relational database, XML

1 Introduction

XML has emerged as the de facto standard for data representation and exchange on the World-Wide-Web (WWW), while relational databases are widely used in enterprises to support critical business operations. Thus, providing interoperability between relational databases and XML data repositories is a very important issue.

There are existing works addressing the issue of representing XML documents in relational databases [1, 2, 5]. To allow users properly manipulate XML data under such environments, an XML query (e.g., in XQuery) needs to be transformed into SQL. The transformation is a challenge due to the difference between XML and relational schema [5]. On the other hand, due to the growing number of data in XML format, "native XML data repository" also receives a lot of attention [3, 4]. However, the ubiquitous presence of relational databases in the business world has resulted in many applications written in SQL. Thus, there is an obvious need for transforming SQL queries into XML queries.

One critical issue needs to be addressed is the mismatch between XML and Relational schemas. *Representational conflicts* have been used in the literature to represent all possible conflicts between two databases, which are used to store the same information. For example, an obvious conflict between XML and relational schemas lies in their different *structures*. A relational schema is *flat* since no explicit structures exist between relations. A relationship is constructed by joining attribute values. On the contrary, XML has a *nesting* structure where the relationship is clearly defined. For a proper query transformation, the correspondence between a join and a nesting structure will need to be properly presented in order to resolve the representational conflicts.

The contributions of this paper are as follows:

- Representational conflicts: The representational conflicts between relational and XML schemas are identified, which serve as the basis of the mapping dictionary.
- Mapping dictionary: A mapping dictionary (based on the XML format) for resolving the representational conflicts between relational and XML schemas is proposed. Mapping dictionary, specified in a declarative manner, is easy to understand and manipulate.
- Prototype: A prototype utilizing the mapping dictionary for query transformation is built to validate our proposal.

The rest of this paper is organized as follows. In Section 2, examples of relational and XML schemas are given to illustrate representational conflicts. In Section 3, we define the mapping dictionary. The transformation algorithms with illustrative query examples are presented in Section 4. Finally, we summarize this work and point out some future research directions in Section 5.

2 Problem Description

In this section, we present the representational conflicts, *i.e.*, all the possible mismatch, between the rela-

*This work is partially supported by the Republic of China National Science Council under Contract No. NSC 93-2422-H-019-001.

tional schema and the XML schema. The term *schema* and *database* are used interchangeably. We also use the term *table* instead of *relation*, and the term *field* instead of *attribute*, to avoid confusion.

2.1 Sample Schemas

The sample schemas used throughout the paper is presented in this section. The relational schema is illustrated in Figure 1(a). The tables *student* and *course* represent the basic information of students and courses. They define the primary keys *sid* and *cid* respectively, which are denoted using a underline. The table *enroll* identifies the grade of a certain student for a certain course. Note that the field *sid* corresponds to the primary key *sid* of the table *student*; similarly for the field *cid* and the table *course*. One more table *student_phone* is designed to represent the phone numbers of each student. Since each student could have multiple phone numbers, such information is represented in a separate table due to normalization.

To explicitly show the key correspondence between tables, the relational schema is depicted as the *RDB graph*, as shown in Figure 1(b). Each node corresponds to a table, and a directed link points from the table representing the primary key to the table representing the foreign key. For example, the attribute *sid* of the table *enroll* is defined as a foreign key corresponding to the primary key of the table *student*.

The XML schema (or DTD) basically supports nesting relationship or sibling orders between elements. To make the structure of a DTD more easily observed, it is represented as a rooted graph, and named as the *DTD graph*. Figure 2(a) illustrates the DTD graph for the sample XML schema, which represents similar information as in Figure 1. The root of the tree corresponds to the root element of the DTD document, which is the *school* element in this example. The nesting relationship between elements is represented by the relationship of parent/child in the graph. For example, the root element has three sub-elements, *student*, *course*, and *evals*.

If an element is associated with values, called *value elements*, it will be represented by rounded squares, *e.g.*, *phone*; otherwise, the elements will be represented using squares, *e.g.*, *enroll*. On the other hand, if an element is allowed to have multiple occurrences in the same document, called a *repeatable element*, the node will be represented by thicker lines. For example, there could be many *student* element instances, and each *student* could in turn possess many *enroll* element instances. Value elements could be also repeatable. In this example, a *student* element instance could have multiple *phone* element instances.

Attributes are represented using ellipses, and are depicted as the children of the associated element. Attributes are normally used to represent values, but it can also define different types to illustrate certain constraints or relationship. For example, the *course* element defines an attribute *cid*, which has the attribute type ID to function as the identifier of each *course* element instance. On the other hand, the attribute *cid* of the element *enroll* is defined with the type IDREF, which requires each attribute value to be also associated with an ID attribute of some element in the same XML document. Such correspondence is explicitly represented as an arrow pointing from the IDREF attribute to the element which defines the corresponding ID attribute. It can be considered as a reference from one element to another element.

2.2 Representational conflicts

The *representational conflicts* between the relational and XML schemas are classified into several cases and illustrated using the two running schemas as follows:

A. Table-versus-Element conflicts

Relational databases use *tables* as the basic unit to define related information, and the counter part in XML databases is the construct *element*. For example, the information associated with the table *course* in Figure 1, is represented by the element *course* in Figure 2(a). In general, the cardinality of correspondence might be one-to-one, many-to-one, or one-two-many, *etc.*, and there might exist *naming conflicts* when different names are used to represent semantically equivalent objects.

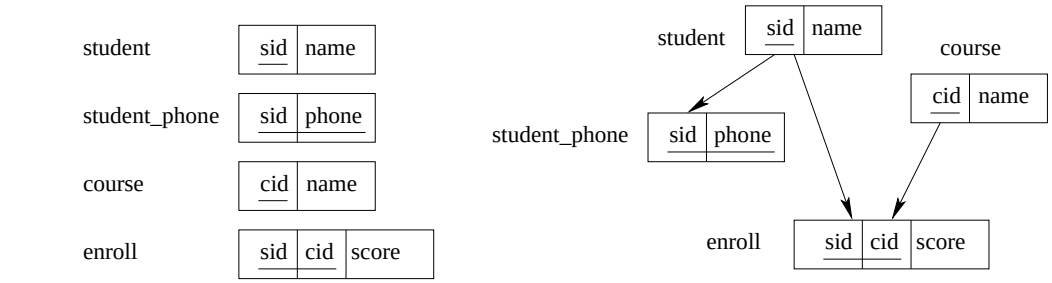
B. Field-versus-Element-or-Attribute conflicts

Values in relational databases are associated with fields, and could be identified or retrieved by the expression *table.field*. In XML databases, data could be represented either by value elements, *e.g.*, the *phone* element, or attributes, *e.g.*, the *sid* attribute. However, since the same name could be defined several times with different meanings in the same XML document, we need to use the *path expression*, which traverses the graph structure from the root to a certain element, instead of only element tags, to avoid confusion. For example, the path expressions of the two occurrences of the element *name* are */school/course/name* and */school/student/name*, respectively. A special symbol "@" is used to designate an attribute, *e.g.*, */school/student/enroll@cid*.

C. Semantic conflicts

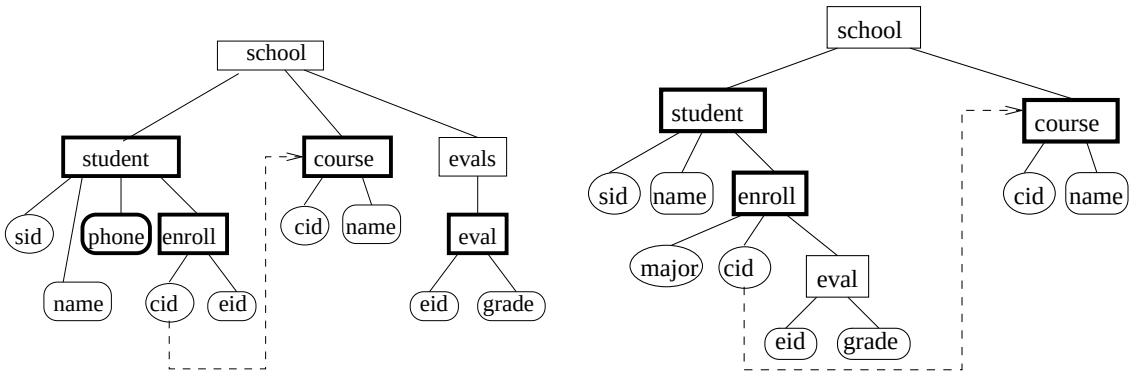
This category refers to mismatch in data values or data types, *etc.* The common data type in relational database is alphanumeric, while that of XML documents is the plain text. Functions which provide transformation between data in the two databases should be defined.

D. Structural conflicts



(a) The relational schema (b) The RDB graph

Figure 1. Sample relational schema



(a) The sample DTD graph (b) The nesting DTD graph

Figure 2. Sample XML schema

To cope with the "flat" structure of tables in relational databases, the relationship between tables is constructed by joining field values, particularly through the primary key/foreign key. This structure could be similarly represented in XML. For example, in Figure 2(a), the elements *student* and *evals* do not have direct structural relationship and the connection is built through the two descendent value elements *//student/enroll/eid* and *//evals/eval/eid*. As a counter example, since the *eval* element is directly nested within the *student* element in Figure 2(b), we could directly retrieve the evaluation records of a certain student through the path expression *//student/enroll/eval*.

For easy identification, the conflicts specified in cases A, B, and C will be referred as the *basic conflict* in the remaining of the paper, to be distinguished with the *structural conflict* in case D.

3 Mapping Dictionary

A mapping dictionary is constructed to resolve representational conflicts between schemas as discussed in Section 2. The definition of the mapping dictionary will be provided in this section, along with sample mapping information based on the two running schemas.

3.1 Resolving Basic Conflicts

The mapping dictionary captures all the mapping information between the relational schema and the XML schema. The mapping dictionary itself is represented based on the XML format due to its powerful modeling constructs. The corresponding DTD graph is shown in Figure 3.

Observe that one attribute and four sub-elements are defined by the root element *MD*. The attribute *Rdb* represents the name of the relational database. The first sub-element *Xdoc* represents the document name or the directory where the XML data is stored. It is allowed to be multiple-occurred to support the case when many XML documents conform to the XML schema. The second sub-element *Table* represents the relevant mapping information for each table in the relational database. The table is indicated by the attribute *Tname*, and the attribute *Txpath* represents the path expression of the element in the XML schema which is semantically equivalent to the target table. By this way, we resolve conflict A in Section 2, *i.e.*, table-vs-element conflicts.

The repeatable element *Table* in turn defines another repeatable element *Field*, which represents the mapping information for each field in the relational database. Similarly, the attribute *Fname* represents the

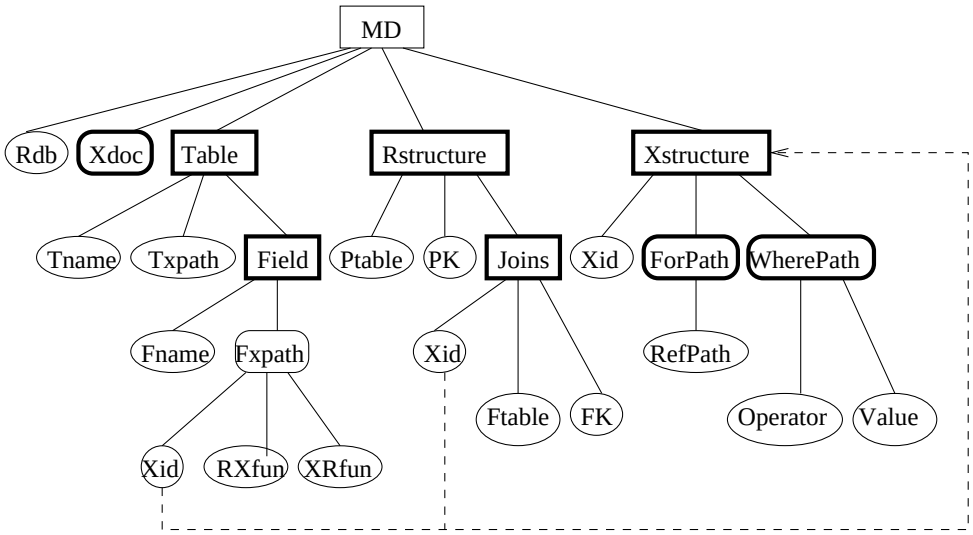


Figure 3. The DTD graph for the mapping dictionary

name of the field, and the value element *Fxpath* represents the path expression of the corresponding value element or attribute in the XML database. Such information resolves conflict B in Section 2, *i.e.*, field-vs-element-or-attribute conflicts. However, since there might exist semantic conflicts (conflict C) between data in two database, we represent the function which transforms relational data to XML data via the attribute *RXfun*, and the reverse transformation function is represented by the attribute *XRfun*.

The following example captures the mapping information for the *student* table in Figure 1 based on the XML schema in Figure 2(a), where the *ItoS* function transforms integers to strings, and the *StoI* function is vice versa:

Example md-1

```
<Table Tname="student" Txpath="/vs/student">
    <Field Fname="sid">
        <Fxpath RXFun="ItoS", XRFun="StoI">
        /vs/student@sid</Fxpath>
    </Field>
    <Field Fname="name" >
        <Fxpath RXFun="_", XRFun="_">
        /vs/student/name</Fxpath>
    </Field>
</Table>
```

An underscore "_" is used when no transformation function is needed. Refer to Figure 3 again. Note that the element *Fxpath* defines an attribute *Xid*. It is used for resolving structural conflicts, as will be discussed in the later sub-section.

3.2 Resolving Structural Conflicts

The remaining sub-elements of the root element, *i.e.*, the elements *Rstructure* and *Xstructure*, will be discussed in this section. They are used to resolve structural conflicts (conflict D) as discussed in Section 2. In short, the information associated with *Rstructure* will be used to identify those joins between tables which convey structural information, and the information associated with *Xstructure* represents the corresponding structure in the XML database.

Recall that the structure of tables is "flat", and the relationship between tables is constructed through joining fields, especially through keys. Therefore, a *Rstructure* element instance is designed to represent all the possible joins for a particular table. The name of the table is denoted by the attribute *Ptable* and its primary key is denoted by the attribute *PK*. Another table to be joined with this table is denoted by the element *Joins*. It is repeatable since a table might have structural relationships with several other tables due to normalization or $n : m$ relationship, *e.g.*, the table *student* in Figure 1. Therefore, each element instance represents one foreign relation (attribute *Ftable*) along with the foreign key (attribute *FK*).

The element *Joins* further defines the attribute *Xid* with the type IDREF to reference a particular *Xstructure* element instance, which is identified by the ID attribute *Xid*. The element *Xstructure* also defines two elements *ForPath* or *WherePath* to provide the structural information in the XML schema corresponding to the join condition in the target relational schema. The element *ForPath* is used when the two elements corresponding to the two joined tables are represented by the *nested* structure, and will be used to construct the FOR clause of the XQuery. The element *WherePath* will be used for unstructured elements, and will be used to construct the WHERE clause of the XQuery. They are illustrated below using examples.

Consider the join statement *student.sid = enroll.sid*. Since the corresponding element *enroll* is nested within the corresponding element *student*, the element

Algorithm BuildMD
Input: rdb //an RDB tree
Output: md //the mapping dictionary
begin
1. md = ( );
2. For each root node r in rdb
3. md = md + TraverseRDB(r);
4. End for;
5. return md;
end

Algorithm MatchNode
Input: n //an RDB node
Output: md //the mapping dictionary
Function
1. Build the information for /MD/Table/Field.
2. Determine if there exists structural conflicts.

Algorithm MatchEdge
Input: n, c //two RDB nodes with parent-child relatinship
Output: md //the mapping dictionary
Function
1. Build the information for /MD/Rstructure.
2. Determine if there exists structural conflicts.

Algorithm TraverseRDB
Input: n //an RDB node
Output: md //the mapping dictionary
begin
1. md = ( );
2. If n.flag == 1
3. Then return md
4. If n is a leaf node Then
5. md = md + MatchNode(n)
6. n.flag == 1
7. return md;
8. End if;
9. For each child c of n
10. If c.flag == 0
11. md = md + TraverseRDB(c);
12. md = md + MatchEdge(n, c);
13. End For;
14. md = md + MatchNode(n);
15. m.flag = 1;
16. return md;
end

Figure 4. The algorithms for constructing the mapping dictionary

ForPath will present the path up to the *nested* element *enroll*. The *Xstructure* element instance is specified as follows:

Example md-2
<Xstructure Xid = "X001">
<ForPath>/vs/student/enroll</ForPath>
</Xstructure>

Consider the other join condition *course.cid = enroll.sid*. Although the corresponding elements *course* and *enroll* are not nested through the parent/child link, they could be considered *nested* through the link IDREF. Therefore, the *Xstructure* element instance will also define a *ForPath* element instance for the structural information. Moreover, the attribute *RefPath* is used to represent the element being referenced by the link, as specified in the following:

Example md-3
<Xstructure Xid = "X002">
<ForPath RefPath="/vs/course">
/vs/student/enroll/id(@cid)</ForPath>
</Xstructure>

We now explain the definition of the element *WherePath*. It is used if the corresponding elements are *un-nested*, and a condition, usually a join statement, needs to be specified in the WHERE clause of XQuery as in SQL. The condition is divided into three parts, and represented by the value element *WherePath* along with its two attributes *Operator* and *Value*. We use the structural conflict incurred by a field as an example to illustrate how to construct the mapping information. Suppose we intend to retrieve the score of a particular student. We could output the field *score* from the *enroll* table by restricting the fields *SID* and *CID* in Figure 1. However, in Figure 2, the attribute *SID* of the element *student* and the attribute *CID* of the element *enroll*, could only get us the element *enroll* and the identifier of the corresponding evaluation record, *i.e.*, *eid*. One more join between *//enroll/eid* and *//eval/eid* is necessary to get the required *grade*. Such information is presented by the *WherePath* element as follows:

Example md-4
<Xstructure Xid = "X004">
<WherePath Operator = "="
Value = "/vs/evals/eval/eid">
/vs/student/enroll/eid </WherePath>
</XStructure>

3.3 Algorithms

We have implemented a set of algorithms (Figure 4) to assist in the construction of the mapping dictionary. The details of the algorithm are omitted due to space limitation. In short, the main algorithm BuildMD will start processing from each root node in the RDB graph, and perform a depth-first-search in the sub-graph. For each node, the algorithm MatchNode will be invoked to find the mapping information for each field. For each edge, the algorithm MatchEdge will be invoked to identify the correspondence for the joining relationship. Note that a node could be pointed by several links, such as the node *enroll* in Figure 1(b). Therefore, each node is associated with a flag, which will be assigned the value "1" when the mapping information has be constructed, as shown in line 6 and line 15 of Algorithm TraverseRDB.

Transformation Module
Input: an SQL query S, the mapping dictionary MD
Output: an XQuery statement X
begin
1. Transform the input S to internal structures:
   Invoke **Algorithm Preprocessor** to output Fobj, Wobj, and Robj corresponding to the FROM, WHERE, and SELECT clauses, respectively.
2. Resolve basic conflicts:
   Invoke **Algorithm Accessor** to get the mapping information under MD/Table and /MD/Table/Field, and represent it in the internal structures.
3. Resolve structural conflicts:
   If **Algorithm Accessor** detects there exists values in Xid, either under /MD/Table/Field/Fxapth or /MD/Rstructure/Joins, get the proper Xstructure element instance with the same Xid, and output the ForPath object and WherePath object if applicable.
4. Update Fobj and Wobj:
   Invoke **Algorithm Joinprocessor** to process all ForPath and WherePath. Update existing Fobj if necessary or create new Fobj or Wobj.
5. Formalize all paths:
   Invoke **Algorithm Formalizer** to process the paths in Wobj, Robj, and Fobj. Create a proper sequence of variable bindings.
6. Construct the output:
   Invoke **Algorithm Constructor** to produce an XQuery based on Fobj, Wobj, and Robj.
end

Figure 5. The algorithms for query transformation

4 Query Transformation

The different syntax of SQL and XQuery is illustrated below by using two sample queries. Suppose the user needs to identify all students who have registered the course "PL", and retrieve the score of this course. The SQL query posed against the relational schema in Figure 1 will be as follows:

```
SELECT student.name, enroll.score
FROM   student, enroll, course
WHERE  course.name = "PL" AND
       student.sid=enroll.sid AND
       course.cid = enroll.cid
```

The XQuery statement which performs the same function as the previous query does, but is appropriate for the XML schema in Figure 2(a), will be as follows:

```
FOR $t1 IN /school/student, $t2 IN $t1/enroll,
    $t3 IN $t2/id(@cid), $t4 IN /school/evals/eval
WHERE $t3/name = "PL" AND $t2/eid = $t4/eid
RETURN $t1/name, $t4/grade
```

An XQuery statement usually consists of three clauses. The FOR clause lists a sequence of variable bindings; the WHERE clause provides restriction on values; the RETURN clause constructs the output. To briefly explain this query, the variable $t1$ considers all students, the variable $t2$ examines all the evaluation records of a student, and the variable $t3$ refers to the course corresponding to this evaluation record. In the WHERE clause, we determine if the course has the name "PL", and retrieve the corresponding evaluation records through the variable $t4$ and the sub-element *eid*. The name of the identified student with the grade of the course "PL" are then returned.

The set of algorithms which could transform an SQL query into an equivalent XQuery statement is shown in Figure 5. Some algorithms are mainly to identify the useful mapping information in the mapping dictionary to meet the requirement of the schema, and others deal with the different syntax between SQL and XQuery to make sure proper constructs are produced. Note that three structural conflicts will be observed in Step 3 for the sample queries. The first one is introduced by the field *enroll.score*, and is resolved by the mapping information presented in **Example md-4**. The second one is introduced by the join condition *student.sid=enroll.sid*. It is used to connect the tables *student* and *enroll*, and corresponds to two nested elements in the XML database. It is resolved based on the mapping information in **Example md-2**. The other join condition *course.cid = enroll.cid* also conveys structural information, and the corresponding elements in the XML databases are nested through the IDREF link. It is resolved based on the mapping information in **Example md-3**. The details of other steps are omitted due to space limitation.

5 Conclusions and Future Research

In this paper, we propose a mapping dictionary to resolve representational conflicts between the relational and the XML schemas. The mapping dictionary, represented in XML, is declarative. A set of algorithms are develop to perform the transformation between SQL and XQuery by accessing the mapping dictionary. A prototype is built to validate our proposal.

As for the next step of this work, we plan to conduct a more comprehensive empirical study by using complicated schemas and queries on our prototype. We also plan to extend the mapping information and algorithms to resolve more representational conflicts such that complicated syntax constructs could also be processed.

References

[1] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From xml schema to relations: A cost-based approach to xml storage. In *Proceedings of the 18th ICDE*, 2002.
[2] D. Florescu and D. Kossmann. Storing and querying xml data using an rdbms. *IEEE Data Engineering Bulletin*, 22(3), 1999.
[3] H. Jagadish et al. Timber: A native xml database. *The VLDB journal*, 11(4), 2002.
[4] J. Naughton et al. The niagara internet query system. *IEEE Data Engineering Bulletin*, 24(2), 2001.
[5] 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 *Proceedings of the VLDB conference*, 1999.