Representing Multiple Mappings between Relational and XML Schemas to Support Interoperability[*]

Ya-Hui Chang and Chia-Zhen Lee
Department of Computer Science
National Taiwan Ocean University
Email: yahui@mail.ntou.edu.tw

Abstract

Providing interoperability between relational databases and XML databases has been an important research issue. In this paper, we propose a set of mappings to represent the correspondence between the relational schema and the XML schema. Particularly, we consider the case of multiple mappings between value, collection, and structure constructs. Based on the mapping information, relational queries and XML queries could be transformed to each other and information could be therefore shared. We have built a prototype and experimental results validate the proposed approach.

Keywords: schema mapping, query transformation, XML schema, relational schema

1 Introduction

XML has emerged as the de facto standard for data representation and exchange on the World-Wide-Web, while relational databases are widely used in enterprises to support critical business operations. Providing a convenient way to access data in the two formats is thus a very important issue.

One of the main approach is to represent XML documents in relational databases [1, 2, 5], where users pose XQuery statements against the XML view, and queries are transformed into SQL which are executed in the underlying relational databases. On the other hand, native XML data repositories have received a lot of attention [3, 4], and there is a need to transform the SQL statements coded in existing applications into XQuery to get the data represented in the new format.

In this paper, we propose a general framework, where SQL and XQuery statements can be easily transformed to each other. The main challenge to achieve this goal is to properly represent the mapping between the relational schema and the XML schema, where representational conflicts exist. Take the *structure* constructs as an example. A relational schema is usually considered as *flat*, since no explicit structures exist between relations and the relationship is constructed by joining attribute values. On the contrary, the relationship between XML data could be directly represented through the *nesting* structure. The correspondence between a join and a nesting structure will need to be presented. Moreover, we consider the possibility of multiple mappings between the different constructs in two schemas, which is usually neglected to simplify the transformation process.

The contributions of this paper could be summarized as follows:

- Classification of the representational conflicts: We consider the *value*, *collection*, and *structure* constructs represented in the relational and XML schemas, and discuss the possible sources of representational conflicts.
- Specification of the mapping of schemas: We design a set of mappings to represent the correspondence between different constructs of the relational schema and the XML schema. The type of mappings is represented to help the selection among multiple choices.
- Design of the transformation algorithms: We have designed a set of algorithms, which utilize the mapping information to perform query transformation between the most common type of SQL and an equivalent XQuery statement. Experimental results show the feasibility and efficiency of the proposed approach.

The remaining of this paper is organized as follows. In Section 2, we describe how to represent relational and XML schemas, and formulate the problem to solve. In Section 3, we define a set of mappings between different schema constructs. Transformation algorithms along with examples are

[*]This work is partially supported by the Republic of China National Science Council under Contract No. NSC 96-2221-E-019-048-.

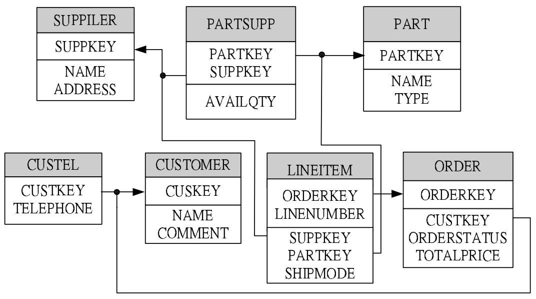


Figure 1. The sample relational schema rdb

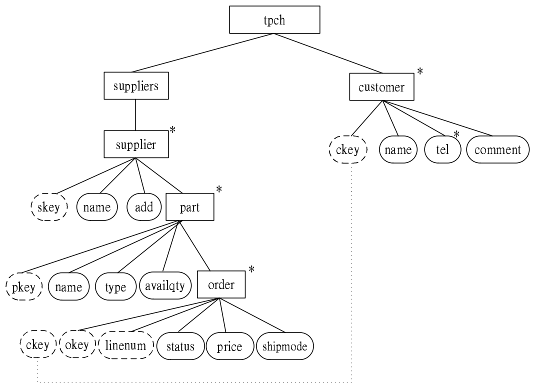


Figure 2. The sample XML schema xdb

presented in Section 4, and experiments are described in Section 5. Finally, we conclude this paper with a brief summary in Section 6.

2 Preliminaries

In this section, we show the representations of the relational schema and the XML schema, and the corresponding queries. We also formalize the problem to solve in this paper.

2.1 Schema Representations

The relational schema is represented as a graph, where each box corresponds to a relation. The attributes associated with the relation is represented within the box, with the primary key on the top. The foreign key is represented as an arrow pointing to the corresponding primary key. A sample relational schema is illustrated in Figure 1. We can see that the attribute *partkey* of the relation *partsupp* is a foreign key corresponding to the primary key *partkey* of the relation *part*. We further classify the relations into two types. The *E-relation* refers to a relation which describes the information of an entity, *e.g.*, *supplier*, *part*, and *order*. The *R-relation* refers to a relation which describes the relationship among other entities, *e.g.*, *partsupp* and *lineitem*.

The XML schema is also represented as a graph, where elements are represented as square nodes, and the nesting relationship between elements is represented by the relationship of parent/child in the graph. The sample XML schema, which represents similar information as in Figure 1, is illustrated in Figure 2. We also define several special kinds of nodes. First, the *repeatable element* refers to an internal node which is allowed to have multiple occurrences under the same parent element, and is annotated by the symbol "star", *e.g.*, *order*. On the other hand, the *dummy element*, *e.g.*, *suppliers*, is an internal node which is usually introduced just to group elements that appear beneath it. Finally, the *leaf nodes* are associated with values. They might be elements, which are named as *value elements* and are denoted by rounded rectangles, or they might be *attributes*, which are represented using dashed lines. We also use dotted lines to connect two leaf nodes which are semantically equivalent.

To uniformly refer to the construct represented in different schemas but with the same functionality, we define the following terms. A *value construct* is the construct which directly represents data. It will be the *attribute* in relational databases, and the *value element* or the *attribute* in XML databases. The *collection construct* is a construct which represents a set (multi-set) of data with homogeneous structures. It will be the *relation* in the relational databases, or the *repeatable element* in the XML databases. The *structure construct* is used to connect two collections. In XML, it can be directly represented by the *nesting construct*. As seen in Figure 2, the *order* element is directly nested within the element *part*. In contrast, since a relational database has a "flat" structure, which has no direct structure construct, the relationship between relations is built by specifying *joining statements* in the query, particularly through primary keys and foreign keys. This will be explained further in the next subsection.

2.2 Sample Equivalent Queries

We use the standard SQL and XQuery to explain the syntactic difference of the query languages for different schemas. Suppose the user intends to identify the type of all parts with the name "dvd", and retrieves the name of its suppliers. The SQL query posed against the relational schema in Figure 1 will be as follows:

SQ1:
```
SELECT supplier.name, part.type
FROM   supplier, part, partsupp
WHERE  part.name = "dvd" AND ................................ (1)
       supplier.suppkey = partsupp.suppkey AND .... (2)
       part.partkey = partsupp.partkey ...................... (2)
```

To briefly explain, the FROM clause is used to enumerate all the relations consulted, and the SELECT clause lists the attributes for output. The conditional statements listed in the WHERE clause could be classified into two types: the one marked with (1) is called the *selection statement*, which restricts the values of certain attributes; the one marked with (2) is called the *join statement*, which constructs the relationship between two relations. Note that the two join statements construct the relationship between the two relations *supplier* and *part*.

The XQuery statement which performs the same function as *SQ1* does, but is appropriate for the XML schema in Figure 2, will be as follows:

XQ1:
```
FOR $t0 in /tpch/suppliers/supplier , $t1 in $t0/part
WHERE $t1/name = “dvd”
RETURN $t0/name, $t1/type
```

An XQuery statement uses the FOR clause to list a sequence of variable bindings. In this query, the variable $t0$ considers all suppliers, and the variable $t1$ examines all the parts supported by a supplier. The WHERE clause is used to specify the selection condition, and the RETURN clause specifies what to output. Note that XQuery uses *path expressions* such as */tpch/suppliers/supplier* to navigate the nesting structure of the XML schema.

Similarly, we define some terms which have the common functionality in different query languages. First, *value literals* refer to those statements represented in the SELECT/RETURN clause, such as *supplier.name*, or the selection statement in the WHERE clause. The *collection literals* refer to the relations or elements extracted from the FROM/FOR clause, such as */tpch/suppliers/supplier*. Finally, the structure literal will be the *join statement* in the WHERE clause, such as *“supplier.suppkey = partsupp.suppkey”*, or the *nesting statement* in the FOR clause, such as *“$t1 in $t0/part”*.

2.3 Problem Definition

In this paper, we intend to translate an input XQuery into an equivalent SQL query, or vice versa. Traditionally, two equivalent queries in the same database mean that they can retrieve the same set of data. This definition does not apply in our heterogenous environment, since the data in two databases might not be the same, due to different data sources, constraints, or formats. Therefore, we define the equivalency based on the query statements themselves, as follows:

Definition 2.1 *Given the input query q_i and the output query q_o, q_i and q_o will be strongly equivalent, denoted $q_i \equiv q_o$, if they have the same numbers of value literals, collection literals, and structure literals, and the corresponding literals are equivalent.*

Definition 2.2 *Given the input query q_i and the output query q_o, q_i and q_o will be weakly equivalent, denoted $q_i \simeq q_o$, if they have the same number of equivalent value literals, but with different numbers of equivalent collection literals. However, the collections in q_o are connected by proper structure literals.*

For the sample queries above, SQ1 is weakly equivalent to XQ1 based on the definition. It is due to multiple mappings between the two schemas, as will be explained later. Therefore, we will consider a translated query to be *correct*, if it is strongly or weakly equivalent to the input query. Now the problem to solve in this paper could be formally stated as follows: “Consider two relational or XML schemas, where there might exist multiple mappings between the value, collection, or structure constructs. Given an input query, produce the correct translated query.”

3 Representations of Schema Mappings

We discuss how to represent schema mapping between the relational schema and the XML schema in this section. The sample schemas described in Section 2 will be used as examples, and will be called rdb and xdb, respectively. Also, when refering to a construct in an XML schema, we will only specify the element name, instead of the complete path expression from the root, when there is no confusion.

3.1 Representing Value and Collection Mappings

Values between two databases might exist multiple mappings, due to redundant representations, or keys which are represented in two relations to construct joins. We define the following Value Mapping (VM) to represent the correspondence between values in two schemas:

Definition 3.1 *Given a value construct v_i from schema$_1$, VM(v_i) will return the set of tuples (v_o, type), where v_o represents the equivalent value construct represented in schema$_2$, and the value of types could be PK (standing for primary keys), FK (standing for foreign keys), or ANY.*

In the case of multiple mappings, types are used to define the priority, and PK $>$ FK $>$ ANY, since primary keys have the important identifying characteristics. For example, when mapping from xdb to rdb, VM(supplier@skey) = {(supplier.suppkey, PK), (partsupp.suppkey, FK), (lineitem.suppkey, FK)}, and we will use the attribute associated with the relation *supplier* in the translated query.

The mappings between collections are more complicated, and are discussed as follows:

1:n from xml schemas to relational schemas This refers to the situation where the value elements or attributes under a repeatable element are scattered in different relations. This is usually caused by the *normalization* process in relational databases. For example, in Figure 2, we could directly represent that a customer has many telephone numbers by associating the multi-valued element *tel* with the *customer* element. However, in the relational database which conforms to the first normal form, this information is split into another relation *custel*.

A special case concerns the R-relation, which consists of information from several E-relation. In the XML representation, such relationship could be represented by the nesting construct. For example, PARTSUPP is a relationship between the two E-relations *part* and *supplier*. In the sample XML schema, there is no such explicit element, and *part* is represented as a child element of *supplier* instead. Therefore, we will let PARTSUPP corresponds to the more specific element *part*, the same as the E-relation *part* does.

1:n from relational schemas to xml schemas This refers to the situation where the attributes in one relation are represented under several repeatable elements. It might be similarly caused by different partitions of an entity, *e.g.*, using two repeatable elements to represent normal customers and VIP customers, respectively. It is also possibly caused by the *dummy element*. For example, since the functionality of the dummy element *suppliers* is to group all the *supplier* elements, both *suppliers* and *supplier* will map to the same relation *supplier* in the sample relational schema.

Therefore, the Collection Mapping (CM) is defined as follows:

Definition 3.2 *Given a collection construct c_i from schema$_1$, CM(c_i) will return the set of tuples (c_o, type), where c_o represents the corresponding collection construct represented in schema$_2$. For an XML schema, the type could be REP (standing for repeatable elements) or DUM (standing for dummy elements). For a relational schema, the type could be E (standing for E-relations) or R (standing for R-relations).*

As to the case of multiple mappings, the priority of REP will be higher than DUM. However, we let E and R have the same priority, since which relaiton to output depends on the required attributes. For example, CM(part) = {(part, E), (partsupp, R)}, where the two relations have the same priority.

Table 1. Example of join statements in rdb

R_ID	Condition1	Condition2
RE1	SUPPLIER.SUPPKEY	PARTSUPP.SUPPKEY
RE2	SUPPLIER.SUPPKEY	LINEITEM.SUPPKEY
RE3	PART.PARTKEY	LINEITEM.PARTKEY
RR1	PARTSUPP.SUPPKEY	LINEITEM.SUPPKEY
RR2	PARTSUPP.PARTKEY	LINEITEM.PARTKEY
IJ1	PART.PARTKEY	PARTSUPP.PARTKEY
IJ2	ORDER.ODERKEY	LINEITEM.ORDERKEY
IJ3	CUSTOMER.CUSTKEY	CUSTEL.CUSTKEY

Table 2. Example of path statements in xdb

X_ID	Xpath1	Xpath2
XD1	/tpch/suppliers	part
XN1	/tpch/suppliers/supplier	part
XN2	/tpch/suppliers/supplier	part/order
XN3	/tpch/suppliers/supplier/part	order
XF1	customer@ckey	order@ckey

3.2 Representing Structure Mapping

We will define the structure mapping in this section. Recall that the structure construct might be represented as join statements or nesting statements. For easy explanation, we will represent them using tables, and give each construct an identifier to identify the type of the construct.

When describing the structure constructs in the relational schema, if the second letter of the identifier is the letter E, it will represent a join between two E-relations or one E-relation and one R-relation. If the second letter of the identifier is the letter R, it will represent a join between two R-relations. Some join statements for the sample schema rdb are represented in Table 1.

Similarly, some structure constructs for xdb are represented in Table 2, and the identifier are used to denote its type as well. If the second letter is “F”, which stands for “flat”, the two paths represented by the fields *Xpath1* and *Xpath2*, will be used to construct a joining expression. If the second letter is “N”, it will represent two *nested* elements. If a *dummy* element is involved, the second letter will be “D”.

Table 3 represents some mappings of structure constructs. Since there might exist $1 : n$ mapping between collections, there might exist $1 : n$ mapping between structure constructs as well. If multiple mappings occur in the XML schema, the priority will be XN $>$ XD, *i.e.*, the construct without involving dummy elements will have the higher priority, to be consistent with the priority level in CM. Based

Table 3. Example of structure mapping

R_ID	X_ID
RE1	XD1
RE1	XN1
RE2	XN2
RE3	XN3
RR1	XN2
RR2	XN3

on the same reason, we let RE and RR have the same priority, since which structure construct to output depends on the collection being used.

We summarize the discussion above by defining the Structure Mapping (SM) in the following:

Definition 3.3 *Given a structure construct s_i from schema$_1$, SM(s_i) will return the set of tuples (s_o, type), where s_o represents the corresponding structure construct in schema$_2$, and the type could be RE or RR in the relational schema, or XF, XN, or XD in the XML schema.*

Note that in a special case of multiple mappings between collection constructs, where several equivalent collection are selected for output at the same time, we will need to connect those collections to identify that they are mapped from the same source. For example, the two relations *PART* and *PARTSUP* both map to the repeatable element *part*, so we need to provide an *internal join* between the two relations. Several examples are listed in Table 1, whose identifiers are started with IJ.

4 The Transformation System

We discuss how to perform query transformation using the mapping information in this section, and the sample queries in Section 2 will be used to illustrate the whole translation process.

4.1 Getting Mapping Information

The transformation system is depicted in Figure 3. The input query, either in SQL or XQuery, will be first parsed into the internal representation. The value literals, collection literals, and structure literals are then extracted, and are sent to the corresponding processor for transformation.

The three processors will be invoked in sequence. First, Algorithm ColProcessor processes each collection literal, and gets all the equivalent collection constructs with the highest priority through CM. If there are several of them, each of which will be associated with a flag called UsedFlag, with the initial value FALSE. It will be set TRUE if the associated collection is used further.

Algorithm ValProcessor then identifies equivalent value literals through VM. In contrast, only one with the highest priority will be obtained, since it alone can get the most relevant information. Note that it will update the UsedFlag of the corresponding collection.

Finally, Algorithm StrProcessor will examine and identify all the equivalent structure literals with the highest priority. However, if several of them exist, we will further choose the required structure literals based on the collections being used. To perform this task, we represent all the relevant information into graphs as defined in the following:

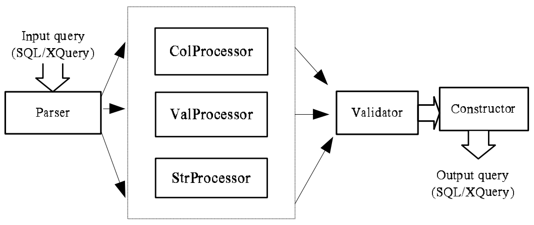


Figure 3. The transformation system

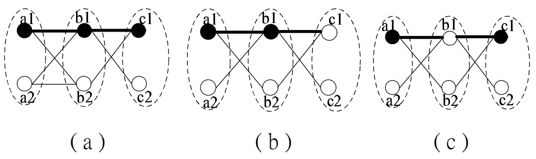


Figure 4. Example of 3-join graphs

Definition 4.1 *A 2-join graph for two input collections c_1 and c_2 is a bipartite graph, where the nodes of partition$_i$ represent the output collections corresponding to c_i. A node will be marked black if its UsedFlag has the value TRUE. The edge connecting two nodes represents the structure literal between the associate output collections, and annotated with the structure identifier.*

Definition 4.2 *A n-join graph consists of a sequence of n partitions, and i$_{th}$ and i+1$_{th}$ partitions and the edges between them form a 2-join graph, where 1 $\leq$ i $\leq$ n-1.*

Three 3-join graphs are illustrated in Figure 4(a)-(c), respectively. In each graph, there are three input collections, and each input collection has two equivalent output collections. We omit the identifier of the structure construct here. The idea is to select the structure construct if the associated collection is marked black. In all cases of this example, the two structure literals corresponding to edges (a_1, b_1) and (b_1, c_1) will be output. The complete algorithm is omitted due to space limitation.

4.2 Processing Query Syntax

After processing the individual collection, value, and structure literals, all the intermediate structures will be processed by Algorithm Validator. As discussed before, the collections which are not used will be removed, and some internal joins might be identified and added for output. Finally, Algorithm Constructor will insert the proper keywords, and produce the transformed query statement. Note that when transforming the structure literals for XQuery, the identifiers will be used to indicate whether a nesting statement in the FOR clause or a join statement in the WHERE clause should be produced.

We use the two sample queries in Section 2 to illustrate the whole process of translation. If the input query is SQ1, the two repeatable elements *supplier* and *part* will be first identified, and their UsedFlags will be further set TRUE by Algorithm ValProcessor. The first join statement, which is RE1, will map to XN1, based on Tables 1-3. XN1 will be selected since both the associated collections are marked black. Therefore, a nested path between *supplier* and *part* will be output, as seen in XQ1. Note that the second join statement is an internal join and have no corresponding output. In the reverse direction, the two repeatable elements will identify three relations, *supplier*, *part*, and *partsupp*, where only the first two relations are marked by Algorithm ValProcessor. The nested path in the FOR clause will then identify the join statement *part.partkey* = *partsupp.partkey*. Note that here Algorithm StrProcessor will mark the relation *partsupp*. Finally, Algorithm Validator identifies that relations *part* and *partsupp* are both introduced by the element *part*, and the internal join *supplier.suppkey* = *partsupp.suppkey* will be appended for output, as seen in SQ1.

5 Experiments

In this section, we evaluate the correctness and the efficiency of the proposed system. All experiments are performed on a P4-2.4GHz machine, with 512 MB of RAM.

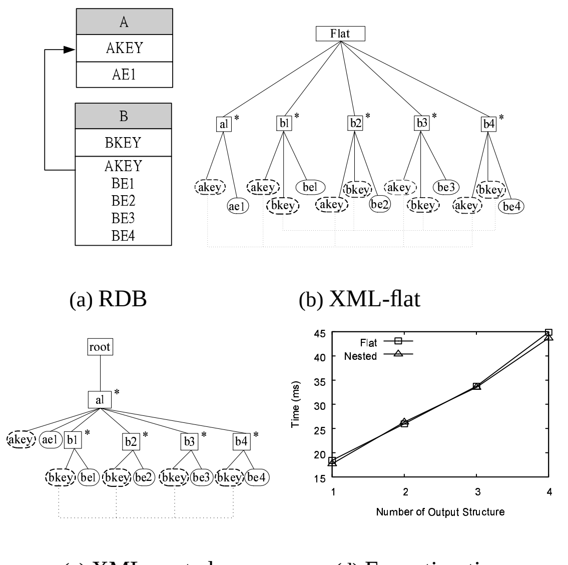
5.1 Correctness

We have applied the two sample schemas described in Section 2 to randomly generate 472 SQL queries and 445 XQuery. Among them, 207 SQL and 139 XQuery statements produce strongly equivalent output queries, and others produce weakly equivalent output queries. The latters are mainly caused by multiple mappings between the constructs, such as: (1) collections are 1:n, and the required values are scattered under different output collections (2) collections are n:1, and the required values are represented within the same output collection.

Since all the translated queries are either strongly or weakly equivalent to the input queries, we can conclude that our system could produce correct translated queries.

5.2 Efficiency

We design several experiments to examine the efficiency of our system, when there exists multiple mappings between the input and output schemas. The schemas used for the first experiment are shown in Figure 5(a)-(c). In the relational schema, the attributes BE1-BE4 are all represented within the relation B, but they are scattered under different repeatable elements in the XML schema. The difference between the two XML schemas, is that the repeatable elements in (b) are in a flat structure, but in a nested structure in (c). We perform four SQL query statements, with increasing BE attributes. Note that the structure literal will increase



(a) RDB (b) XML-flat

(c) XML-nested (d) Execution time

Figure 5. Analysis of Efficiency

along. The transformation time is shown in Figure 5(d). We can see that the number of output structure literals has linear effects on the transformation time, which is acceptable. Also note that the effect of XML schema is quite minor.

Since we can observe the similar results for other experiments, they are omitted here due to space limitation.

6 Conclusions

In this paper, we discuss how to represent the schema mapping between relational databases and XML databases, and the case of multiple mappings is particularly considered. A prototype utilizing these mappings is implemented, which could translate the core expressions between SQL and XQuery. Experimental results have shown that our system could perform transformation correctly and efficiently.

References

[1] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From xml schema to relations: A cost-based approach to xml storage. In *Proceeding 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] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database systems. In *Proceeding of the ACM SIGMOD conference*, 2002.