JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 33, 251-270 (2017)
DOI: 10.6688/JISE.2017.33.1.15

A Tree-Based Approach to Support Query Translation for Schema Mappings with Weights

YA-HUI CHANG, CHIA-ZHEN LEE AND SI-YEN ZHUANG
Department of Computer Science and Engineering
National Taiwan Ocean University
Keelung City, 202 Taiwan
E-mail: {yahui; jiazhen; 10257031}@ntou.edu.tw

Query translation has been a central task for data integration or information sharing. In this paper, we assume that the source and target databases may be XML or relational, with the query languages XQuery and SQL, respectively. Besides, there may exist many possible mappings with different weights between two schemas, which are usually produced by automatic schema matching tools. We intend to output the most preferable query based on weights among all equivalent ones, so we need to properly represent the weighted schema mappings and input queries. Our approach is to first classify the schema representation into two basic units, *i.e.*, *collections* and *values*, and apply mapping functions to represent the correspondence for each basic unit with weight information. We then propose a set of tree structures, collectively called *CanForest*, which show the structural constraints and semantics of the input query and assist in producing the output queries. We have constructed the complete translation system and shown its effectiveness. Experimental results also demonstrate that the system is very efficient.

Keywords: XQuery, SQL, query translation, schema mapping, query tree

1. INTRODUCTION

Query translation has become an important research issue currently since many types of data are connected through the World-Wide-Web. For example, XML, with XQuery as the query language, is the standard for data exchange in the Web. On the other hand, relational databases with mature techniques are still widely used in enterprises to support critical business operations. An SQL statement encoded in existing applications might need to be transformed to an XQuery for retrieving relevant data from the XML data source, so that the information presented respectively as XML data or in the relational database could be shared.

Performing query translation needs to utilize the correspondence of the *target* schema, *i.e.*, where the query is issued, and the *source* schema, *i.e.*, where the data are stored. Previous studies basically considered simple schema mappings, but some researchers [5] started to investigate the problem of *uncertainty* in schema matching. That is, a unit represented in the target schema may correspond to several units represented in the source database, with different *similarity scores*, or called *weights*. Such situation frequently occurs when automatic schema matching tools are used to identify the correspondence of two schemas, and we call it the case of *multiple mappings* between schema units, or *schema mappings with weights*. One way to handle such case is to represent all possible mappings between two schemas in a compact way and produce all correspond-

Received August 22, 2014; revised December 11, 2015; accepted June 4, 2016.
Communicated by Wen-Chih Peng.

ing transformed queries [5], but there are cases that users are only interested in the most possible one, and require only one translated query. Therefore, in this paper, we will output the most convincible query based on the weights of schema mappings.

To achieve this goal, the major challenges are twofold. The first one is how to properly represent the correspondence between schemas with weight information. Schema mappings are usually represented by mapping rules or mapping languages [9, 17], but it is not easy to represent the weights of mappings by using such approach. Therefore, we propose to classify the schema representation into several basic units, which are *collections* and *values*, such as the *relations* and *attributes* in the relational database. We then utilize *mapping functions* to represent the correspondence of the basic unit, and denote the *weight* of each correspondence. Such design can be used to easily identify the most appropriate corresponding unit, *i.e.*, which has the highest weight among all candidates.

The second challenge lies in how to represent the complex structure of a query to facilitate query translation. In this paper, we focus on the core expressions of SQL and XQuery, which consist of Selection-Projection-Join (SPJ) expressions with nested subqueries. To reflect the structural constraints and semantics of the input query, we adopt the tree structure to represent a query as a set of *collection trees* and a set of *value trees*, which are collectively called the *Canonical Forest*, or *CanForest* in short. Take XQuery as an example. Each collection tree is used to illustrate the structural constraint imposed by a binding variable, by representing all the *path expressions* associated with the particular variable specified in the FOR, LET, and WHERE clauses. On the other hand, the *path expressions* specified in the RETURN clause will be used to construct the value trees. The purpose of CanForest is to divide a query into different components, and we can then invoke the corresponding mapping function to identify the appropriate matching units, and directly represent such information within the node of CanForest for producing output query fragments. CanForest is also constructed level by level to reflect the possible nested structure of the original query. Such level information together with the structural constraints represented by the edges of CanForest, can guide us to compose all transformed query fragments into the final correct translated query.

To summarize, the contributions of this paper are as follows:

- We propose to classify schema representations into units of collections and values. Such concepts are used to represent the mappings between two schemas as a set of mapping functions. The weight information associated with each correspondence can assist us in handling multiple mappings between schema units.
- We define the CanForest structure. It represents the structural constraints and returned values specified in the original query clearly to help producing the transformed query.
- We have implemented a system which can perform translation between XQuery and SQL in either direction. Experimental results show that the transformation process can be performed efficiently.

The rest of this paper is organized as follows. Related works are first discussed in Section 2, followed by the description of our approach. In Section 3, we present the schemas and queries to illustrate the problem to tackle in this paper. In Section 4, we illustrate the proposed architecture and discuss the required mapping functions and in-

formation. The *CanForest* structure is explained in Section 5, and the procedure of producing the complete output query is presented in Section 6. Finally, experimental results are shown in Section 7; summaries and future research directions are given in Section 8.

2. RELATED WORK

The needs for translating queries exist in many situations. It might happen when an XML view is on top of a relational engine or XML data are shredded into the relational database, so an XQuery submitted by users needs to be translated to an SQL for executing in the underlying relational databases. The research issues related to this problem are basically about how to execute a more flexible representation of XQuery by SQL expressions. Particularly, an efficient way to handle wild cards in the path expression [6], the technique of performing path expressions on recursive XML schemas [8], and how to produce optimal queries [10], have all being investigated. However, these translation algorithms usually consider fixed and simple schema mappings. In contrast, we focus on handling multiple mappings in our translation system.

The other type of query translation, or called *query rewriting* more precisely, relates to transforming an input XQuery based on views. The variables in the original XQuery can be used to find possible views to rewrite the query [14], and the concept of *maximal contained rewriting* is proposed to use views as much as possible [11]. A *tree pattern language* is designed to describe both the queries and views, and help combine the views into the same structure as the input query [4]. The issue caused by multiple views has also being discussed [13]. These researches are similar to ours in the way that *trees* are used to describe the complex structure of XQuery, but differ in that our proposed tree structure is especially designed to assist in performing query translation in a heterogeneous environment.

Another direction of related researches is about the presentation of complex mappings between schemas. Some representation is proposed to handle "composite" mappings, such as the object *address* consisting of many components like *street*, *city* and *state* [7]. The *tree patterns* are used to represent source-to-target dependence, when both schemas are represented in DTD [3]. A series of researches utilize *languages* to represent mappings between target and source schemas. For example, the source information is represented in the *foreach* clause, and the target information is represented in the *exist* clause [17]. This language is further extended to include the concept of *dynamic element*, to support the need of exchanging between data and meta-data [9]. Although the above representations are useful in certain domains, they are not designed to handle weighted schema mappings. The basic idea of our approach is to divide a schema into several basic units, and utilize *mapping functions* to represent the correspondence of each unit with weights. Such design is flexible and extendable, and can convey more information if required.

Finally, how to automatically determine the correspondence between schemas is a large field and out of the scope of our research. We only list some of the research results here and refer the interested readers to the original papers. An early survey can be found in literature [16], while the emergence of XML data incurs new problems. Some researchers determine the similarity between two DTDs based on the tree structure [12].

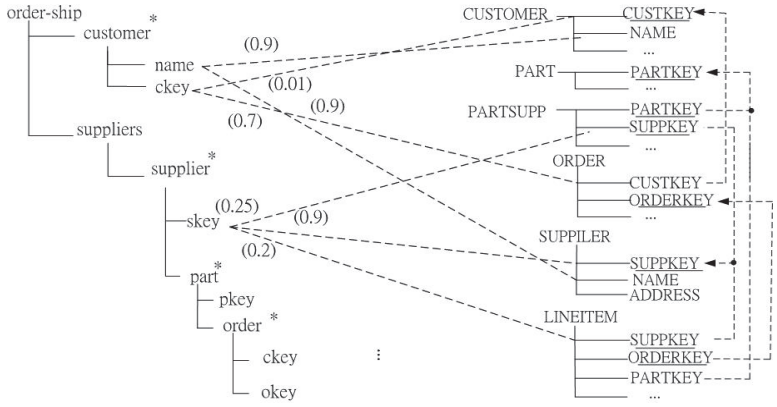
Other researchers propose the *conceptual model* to represent the semantics of the source and target schemas, and then transform the model as graphs, so that the graph algorithms can be used to find mappings [2]. Since many mapping tools exist, a benchmark system called STMark is constructed to evaluate the interfaces of such tools [1]. A self-configuring matching system is also proposed [15].

3. PRELIMINARIES

In this section, we describe the schema representations and query languages of the XML and relational models covered in this research. The sample schemas, which represent the information about *orders*, *customers*, *parts*, and *suppliers*, are adopted from the TPCH benchmark and are slightly modified for being used as the running example.

3.1 Schema Representations and Mappings

The XML schema proposed by W3C consists of many constructs. In this paper, we focus on its nested nature, which is the most important characteristic of an XML schema. For clear illustration, an XML schema definition is depicted as a schema graph. In the sample graph shown in Fig. 1 (a), the *root node* represents the root element *order-ship*, which defines two nested sub-elements *customer* and *suppliers*, and the *customer* element in turn defines two *leaf nodes*, which are *name* and *ckey*. Leaf nodes directly represent values, which correspond to elements with the type #PCDATA or attributes in the DTD definition.



(a) Target (b) Source

Fig. 1. Sample schemas and schema mapping.

The relational schema considered in this paper follows the traditional definition and satisfies the First Normal Form. It is also represented by the graph structure, as depicted in Fig. 1 (b). Since nested relations are not allowed as an attribute domain, the graph only consists of two levels, where a *relation* is represented by a root node, and the component *attributes* are represented by leaf nodes. In addition, the primary key is denoted

by an underline, and the foreign key is depicted as a dotted arrow pointing to the corresponding primary key.

An obvious difference between the two data models is that an XML schema explicitly shows the nested relationship between elements, while a relational schema tends to be flat. To facilitate later discussion, we additionally define the following terms. A *repeatable element* in an XML schema refers to a node which is allowed to have multiple occurrences under the same parent element, and is annotated by the symbol "*" in the schema graph, *e.g.*, the *customer* element in Fig. 1 (a). Relations in a relational schema are classified into two types. The *E-relation* corresponds to a relation which describes the information of an entity, *e.g.*, the *CUSTOMER* relation. The *R-relation* corresponds to a relation which describes the relationship among other entities and has a composite primary key, *e.g.*, the *LINEITEM* relation.

To uniformly refer to the units represented in different models or schemas but with the same functionality, we define the following two terms:

Definition 1: A *value* is a unit which directly represents data, which corresponds to a *leaf node* in the schema graph. A *collection* is a unit which represents a set (multi-set) of data with homogeneous structures. It will be a *relation* in the relational databases, or a *repeatable element* in the XML databases.

Between the two sample schema graphs in Fig. 1, we use the dashed line to connect two matching values. We consider the case of *multiple mappings*, and each dashed line is annotated with a value to represent its *weight* as defined in the following:

Definition 2: Given a value v_i from the target schema and a corresponding value v_o in the source schema, the weight of v_o denotes its similarity degree to v_i, which is obtained by a certain similarity function and represents the priority of choosing v_o.

Assume that the XML schema is the target schema and the relational schema is the source schema. The *name* element of the *customer* element in the target schema corresponds to two *NAME* attributes in the source schema, with the weights 0.9 and 0.01 respectively. Since larger weights represent higher degrees of similarity, we will prefer the attribute defined within the CUSTOMER relation in this case.

XQuery ::= ForClause LetClause? WhereClause? RetClause	W_i ::= P_i CompOP Value | P_i CompOP P_i
ForClause ::= FOR $Var 'in' P_i{, $Var 'in' P_i}	P_i::= PathExpr | $Var Axis PathExpr
LetClause ::= LET {$Var = P_i | XQuery}	PathExpr ::= NameTest | NameTest Axis PathExpr
WhereClause ::= WHERE W_i{'and' W_i}?	NameTest ::= Element | @Attr
RetClause ::= RETURN P_i, P_i	Axis ::= / | //
	CompOP ::= > = <

Fig. 2. The scope of the XQuery query language.

3.2 Query Statements and Problem Definition

The queries considered in this paper consist of the Selection-Projection-Join (SPJ) expressions with possibly nested sub-queries. Note that the *join* expression may be a *valued* join as commonly seen in SQL, or a structural join specified by the path expression of

XQuery. The XQuery syntax is formally specified in Fig. 2, while the SQL syntax is similar and is omitted. In the following, we provide two sample queries based on SQL and XQuery respectively to illustrate their differences. These queries output the identifier of each supplier, and those customers who have posed orders on this company. The SQL query *SQ*1 appropriate for the sample relational schema in Fig. 1 (b) is as follows:

```
L1: SELECT S.SUPPKEY, A.Name
L2: FROM SUPPLIER as S, LINEITEM as L,
L3:         (SELECT C.Name, O.ORDERKEY
            FROM ORDER as O, CUSTOMER as C
            WHERE O.CUSTKEY=C.CUSTKEY) as A
L4: WHERE S.SUPPKEY=L.SUPPKEY and L.ORDERKEY=A.ORDERKEY
```

In an SQL query, the SELECT clause lists the attributes to output. The FROM clause enumerates the consulted relations, within which a nested sub-query, or called a derived relation, can be also specified. In *SQ*1, the statements in L3 are used to derive the correspondence between names of customers and identifers of orders. The WHERE clause specifies the constraints to be satisfied, and the statements in L4 consist of two valued join expressions, which force the equality on two leaf nodes, respectively. Note that the consulted relations *SUPPLIER* and *ORDER* cannot be directly joined, so the R-relation LINEITEM is introduced in L2 and used in L4.

For comparison, the XQuery statement *XQ*1 appropriate for the sample schema in Fig. 1 (a) is specified as follows, where we assume that *ckey* and *skey* are attributes:

```
L1: FOR $s in //supplier
L2: LET $a := FOR $o in $s//order, $c in //customer
L3:           WHERE $c/@ckey = $o/@ckey
L4:           RETURN $c/name
L5: RETURN $s/@skey, $a
```

An XQuery statement basically uses *path expressions* to navigate an XML document, which might consist of the *descendant steps* "*//*" or the *child step* "/'. For example, the expression "*//supplier//order*" retrieves the *order* elements which are descendants of the *supplier* elements. Such expression is a form of *structural joins*. An XQuery also consists of several clauses. Particularly, variable bindings are specified in the FOR clause; the LET clause, which constructs a collection of XML values to be evaluated later, assigns the result of a nested sub-query to a variable (L2-L4 of *XQ*1); the WHERE clause is used to specify conditions on the variables; the RETURN clause specifies what to output. Comparing *SQ*1 and *XQ*1, we can see the direct correspondence between the SELECT and the RETURN clauses, the FROM and the FOR clauses, and the two WHERE clauses, respectively. However, they differ in forming join expressions and nested subqueries.

Observe that *SQ*1 and *XQ*1 show two *equivalent* queries appropriate for databases with different schemas. In this paper, we wish to transform a query like *SQ*1 to a query like *XQ*1, and vice versa. To handle the case of multiple mappings between schema constructs, we define the *weight of a query* as follows:

Definition 3: An expression within a query which outputs or operates on a value, excluding the one being used to perform valued joins, is called a *value literal*. The *weight of a query* is the sum of all the weights of its component value literals, which are in turn determined by the weights of the values operated by these literals.

A query with higher weights means higher credibility, and we assume the one with the highest weight to be *the most preferable query* for the users. Therefore, the *problem definition* of this paper can be formally stated as follows: "Consider a target schema *t* and a source schema *s*, where *t* and *s* are either XML or relational. Given the weighted schema mapping *m* between *t* and *s* and a query *q* posed against *t*, produce the query with the highest weight among all equivalent ones appropriate for *s*."

4. KNOWLEDGE REPRESENTATIONS

In this section, we first give an overview of our approach by introducing the translator architecture, and then discuss the information needed for query translation.

4.1 The Architecture

Fig. 3 shows the proposed architecture for performing query translation. Before this system starts to function, we need to first off-line provide the definitions and structural information of each local schema, and construct the mappings between the target and source schema units. The task of online query translation is then supported by the two component modules. The functionality of the first module will be explained in Section 5, followed by the second module in Section 6. Here we provide a brief description.

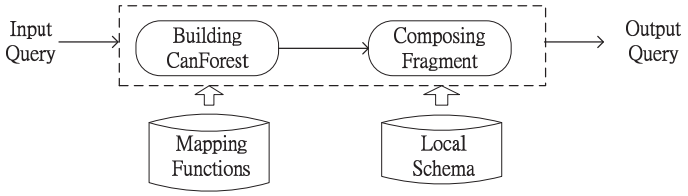


Fig. 3. The architecture for query translation.

First, the module of **Building CanForest** parses a given input query and represents it as a CanForest, which is designed to distinguish the collection component and value component of the input query. For each value component within a query, we then identify the equivalent value with the highest weight through the mapping functions, which are also used to obtain those collections defining the identified value. These equivalent units are recorded within the corresponding nodes in CanForest, and are used to produce primitive query fragments.

Second, the Module of **Composing Fragment** utilizes the structural constraint and level information conveyed by CanForest to retrieve proper structural expressions for connecting the identified collections. All the produced query fragments and appropriate structural expressions will be combined together based on the source local schema to formulate the complete output query.

4.2 Representing Value and Collection Mappings

We propose two mapping functions to represent respectively the correspondence of the two basic types of schema units. The functions *Value Mapping* (*VM*) and *Collection Mapping* (*CM*) are defined as follows:

Definition 4: Given a value v_i from the target schema, $VM(v_i)$ will return the set of tuples (v_o, *weight*), where v_o represents a corresponding value represented in the source schema, and *weight* denotes its similarity degree to v_i, as explained in Definition 2.

Definition 5: Given a collection c_i from the target schema, $CM(c_i)$ will return the set of tuples (c_o, *condition*), where c_o represents the corresponding collection represented in the source schema, and *condition* represents the additional conditional statement which is required to make the two collections equivalent.

Example 1: As shown in Fig. 1, *VM*(*/order-ship/suppliers/supplier/@skey*) = {(*SUPPLIER.SUPKEY*, 0.9), (*PARTSUPP.SUPPKEY*, 0.25), (*LINEITEM.SUPPKEY*, 0.2)}, and *CM(/order-ship/suppliers/supplier*) = {(*SUPPLIER*, *null*), (*PARTSUPP*, *null*), (*LINEITEM*, *null)*}. Note that the three tuples in *CM* have the value *null* for the condition information. However, sometimes we might need it to make two collections equivalent. For example, if there exist two relations *VIP-CUSTOMER* and *OTHER-CUSTOMER*, which are vertical partitions of the repeatable element /order-ship/customer based on its attribute type. We will require a conditional statement such as type = "*VIP*" to exactly identify the corresponding relation *VIP-CUSTOMER*.

4.3 Representing Structural Expressions of Local Schemas

As shown by sample queries *SQ*1 and *XQ*1, structural expressions are necessary components of a valid query, which are basically *valued join* statements or *structural join* statements specified by path expressions of XQuery. We list some join expressions for the sample relational schema in Table 1 as an example.

In Table 1, we give each expression an identifier, which shows how this join is formed. If the identifier is denoted by the letter *R*, it represents a join between one E-relation and one R-relation. For example, *R*1 in Table 1 represents a join between an E-relation PART and an Rrelation LINEITEM. If the identifier is denoted by the string

Table 1. Some join expressions in the sample relational schema.

ID	Table1	Table2	Join Expression	Cost
R1	PART	LINEITEM	PART.PARTKEY=LINEITEM.PARTKEY	1
R2	SUPPLIER	LINEITEM	SUPPLIER.SUPPKEY=LINEITEM.SUPPKEY	1
EE3	SUPPLIER	PART	SUPPLIER.SUPPKEY=PARTSUPP.SUPPKEY.SUPPKEY; PARTSUPP.PARTKEY=PART.PARTKEY	2
R4	ORDER	CUSTOMER	ORDER.CUSTKEY=CUSTOMER.CUSTKEY	1
R5	ORDER	LINEITEM	ORDER.ORDERKEY=LINEITEM.ORDERKEY	1
EE6	ORDER	PART	PART.PARTKEY=LINEITEM.PARTKEY; ORDER.ORDERKEY=LINEITEM.ORDERKEY	2
RR7	PARTSUPP	LINEITEM	PARTSUPP.SUPPKEY=LINEITEM.SUPPKEY; PARTSUPP.PARTKEY=LINEITEM.PARTKEY	2

EE, it will represent a join between two E-relations. In such case, it usually needs other R-relations to construct the relationship, and therefore requires more than one join statements. For example, as specified by *EE*6, the two E-relations ORDER and PART cannot be directly joined, so the R-relation LINEITEM is required. Finally, the identifier *RR* represents a join between two R-relations. Since an R-relation usually has a composite primary key, we need to let the join statements involve all component keys to make the query uniquely identify the correct tuples.

In the last column of this table, we denote the *cost* of each structural expression. It is calculated based on the join distance, *i.e.*, the number of statements required to properly join the two tables. For example, as shown in the first row, we can directly join tables PART and LINEITEM through the join statement "PART.PARTKEY = LINEITEM. PARTKEY", so the cost is "1". In contrast, since tables SUPPLIER and PART cannot be directly joined and another table PARTSUPP is required to form the two join statements properly, the cost is "2". We will show how to use this table later in Section 6.

5. CONSTRUCTING QUERY FORESTS

In this section, we discuss how the query forest, *i.e.*, CanForest, represents the semantics and structural constraints of the input query as a canonical form. We will also present the construction algorithm.

5.1 The CanForest Structure for an Un-nested Query

CanForest consists of a set of *collection trees*, denoted as *ColForest*, and a set of *value trees*, denoted as *ValForest*. Basically, a collection tree represents how a variable imposes the *selection condition* and *structural constraint* in the query. It is rooted by a binding variable specified in the FOR clause or the LET clause. For this particular variable, all the related path expressions specified in the FOR clause, the LET clause or the WHERE clause will be used to construct the remaining nodes of the tree. In contrast, the *Value tree* represents how a variable is going to project data. It is constructed based on the path expressions specified in the RETURN clause in a similar manner. The CanForest structure corresponding to the following query is shown in Fig. 4:

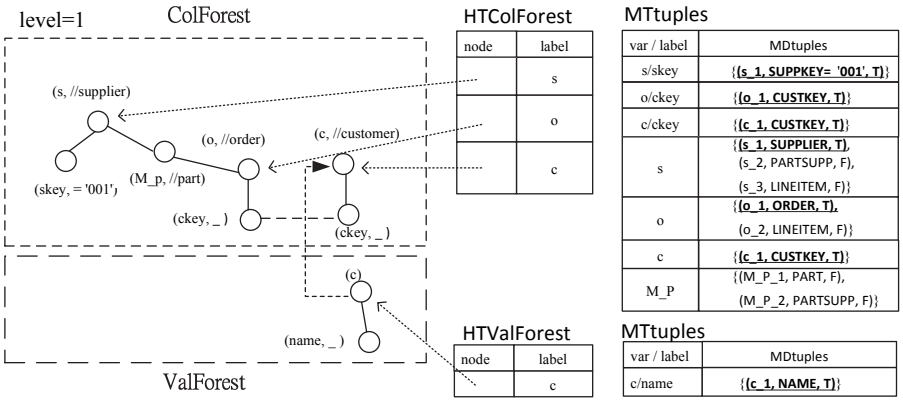


Fig. 4. The sample one-level CanForest for query *XQ*2.

*XQ*2:
FOR $s in //supplier, $o in $s//order, $c in //customer
WHERE $s/@skey = '001' and $c/@ckey = $o@ckey RETURN $c/name

Before formally defining CanForest, we first define *MDtuple* as follows:

Definition 6: Each *MDtuple* associated with a node N is a 3-tuple (*var*, *equi-unit*, *used-flag*). *equi-unit* is the unit in the source schema equivalent to the target unit represented by the node N. *var* is the variable assigned to the collection defining the equi-unit. *used-flag* will have the value "T" if this unit is chosen, and "F" otherwise.

MDtuples represent the mapping information associated with each node. For easy representation, they are listed in separate tables in Fig. 4. In the upper table, the first three rows correspond to the three leaf nodes in ColForest, while the remaining four rows correspond to the four non-leaf nodes in ColForest. We then define each node in CanForest as follows:

Definition 7: A CVnode *n* is defined as follows: (1) If *n* is a non-leaf node, it will be associated with a triple information: (*label*, *path*, *MDset*), where *label* corresponds to a binding variable in the input query, *path* denotes the location of the associated collection in the target schema, and *MDset* consists of a set of corresponding MDtuples. (2) If *n* is a leaf node, it will be associated with a quadruple information: (*label*, *path*, *condition*, *MDset*), where *label* is an element or attribute name, and *condition* represents a selection predicate, which might be *null* and denoted by " ".

The *path* information in a CVnode is used as an input to the mapping functions to get the mapping information. In Fig. 4, it is omitted for leaf nodes in ColForest and all nodes in ValForest. For non-leaf nodes in ColForest, it is only denoted by descendant steps for simplification. Besides, since the leaf node in ValForest is used only for output, its *condition* will be null.

The tree structure and the forest structure can be defined as follows:

Definition 8: A *collection tree* or a *value tree* is a pair of (*N*, *E*), where each node in *N* is a CVnode, and each edge in *E*, called an *Axis edge*, will connect two nodes if their correspondences in the original schema graph have the parent/child relationship.

Definition 9: A *CanForest* for an un-nested query is a set of collection trees and value trees, which might be connected by the following two types of edges: (1) *FlatJoin*: representing a join statement between two leaf nodes of two collections trees; (2) *BoundJoin*: pointing from the root node of a value tree to the root node of the corresponding binding collection tree, so that the root node of the bound value tree can share the same MDset with the root node of the binding collection tree.

Note that there are totally three types of edges in CanForest. In Fig. 4, an *Axis* edge is represented by a solid line, a *FlatJoin* edge is represented by a dashed line, and a *BoundJoin* edge is represented by a dotted arrow. For example, a FlatJoin edge exists between the two leaf nodes with the label *ckey*. A BoundJoin edge pointing from the root node "*c*" in ValForest to the root node "*c*" in ColForest.

5.2 Construction Algorithm

We explain how to construct CanForest in this subsection. Note that there are two hash tables in Fig. 4, which are HTColForest and HTValForest. Their input is a label and the output is a pointer to the corresponding node in CanForest. This design is to let a non-root node efficiently identify the tree which it belongs to.

In our system, individual algorithms are designed to process the different syntaxes of XQuery and SQL. Instead of giving the complete detailed algorithm, we discuss the major steps of processing XQuery as an example. As listed in Fig. 2, this paper considers four clauses of XQuery. In the following, we discuss how to process each individual clause to build the corresponding structure of CanForest, and please note that the descendant steps in each path expression will be expanded by consulting the local schema and additional nodes will be created if necessary:

- the FOR clause:
  Extract the defined binding variable and create a corresponding internal node in the ColForest. Record an entry in HTColForest to represent the correspondence of this variable and its location in ColForest.
- the LET clause:
  For an un-nested query, the expression associated with the variable will be a path expression. Process it in the way similar to the variable binding in the FOR clause.
- the WHERE clause:
  Use HTColForest to identify the collection tree under which the leaf node corresponding to the component path expression should be created. Create a FlatJoin edge if a valued join expression is detected.
- the RETURN clause:
  Process it as processing the FOR clause, except that a value tree is created, the HTValForest structure is used, and a BoundJoin edge is created to point to its binding counterpart.

Example 2: Consider *XQ*2. When processing the *FOR* clause, the three internal nodes labeled "*s*", "*o*", and "*c*" are created for the three binding variables, and three corresponding entries are inserted into *HTColForest*. An additional node with the label "*M_p*" is created when expanding the descendant step, which corresponds to an intermediate element named "*part*" in Fig. 1 (a). Then, for the *WHERE* clause, a leaf node is created and labeled with "*ckey*" according to the path expression "*$c/@ckey*". Besides, two leaf nodes both denoted as "*ckey*" will be created under the internal nodes "*o*" and "*c*", for the join statement. A *FlatJoin* edge will be also created to link these two nodes. Finally, based on the *RETURN* clause, the root node of a value tree along with a *BoundJoin* edge will be first created, and followed by the leaf node "*name*".

During the construction process, our algorithm also needs to represent the mapping information in CanForest. The process differs for different types of nodes. For internal nodes, we get all the matching collections from the mapping function *CM* using the *path* information associated with this node, and represent them as the MDset structure. If it consists of several MDtuples, each of which will be associated with the *usedflag* with the initial value "F". For leaf nodes, we identify the equivalent value from the mapping

function *VM*, similarly based on the *path* information associated with this node. In contrast, only the one with the highest weight will be obtained, since it represents the most relevant information. Such information will be also used to update the *usedflag* of the MDtuple which represents the collection defining the chosen value, and to obtain the variable name assigned to that collection.

Example 3: Consider the leftmost leaf node in Fig. 4, which is labeled by "*skey*". As discussed in Example 1, it has three equivalent values defined by the mapping function *VM*. We choose the one with the highest weight, which is *SUPPLIER.SUPPKEY*, and represent it in the *MDtuple*, as shown in the first row of the upper *MTtuples* table. In contrast, the leftmost root node, which is denoted by "*s*", has three equivalent collections in *CM*. We will create three *MDtuples* for each of them, but only mark the *usedflag* of the first one as "*T*", to show that this one defines the chosen value.

5.3 The CanForest for a General Query

We now discuss the CanForest for a general query. The sample CanForest corresponding to *XQ*1 in Section 3 is illustrated in Fig. 5. A major difference between this example with the previous one is that the CanForest is divided into several level blocks, which is denoted by the darker dashed line. In this example, there are two level blocks, and each level block will produce a corresponding sub-query.

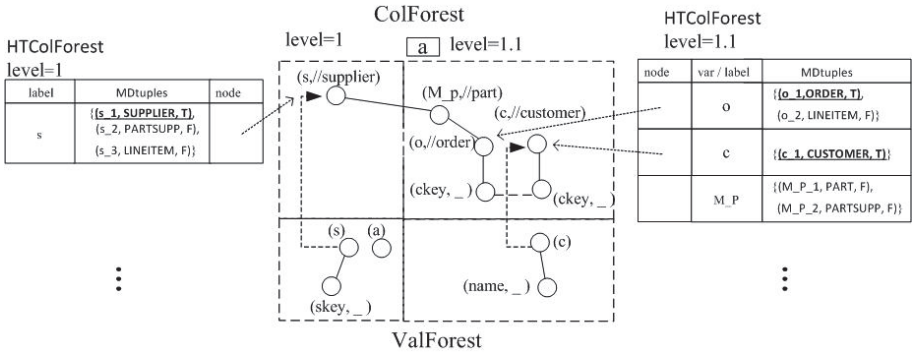


Fig. 5. The sample CanForest for query *XQ*1.

Note that each level block is denoted by a *level number*, which is a sequence of positive integers separated by the period. It is used to reflect the nested structure represented in the original query. We briefly describe the procedure of assigning level numbers here. The outermost level has the initial number 1. Whenever a nested structure is encountered in a LET clause, the period will be added to the current level number to show the nested structure. For each level block, we may also denote its *nested name*, which is used as the alias of the output derived relation if applicable. We now provide the formal definition of CanForest for a general query as follows:

Definition 10: A *CanForest* for a general query is a set of collection trees and value trees, which can be connected by *FlatJoin* and *BoundJoin* as before. In addition, all

CVnodes corresponding to the same sub-query are represented in the same level block. Each level block is assigned a *level number* and possibly with a *nested name*.

Note that each level block has its own HTColForest and HTValForest. We only represent part of its contents in Fig. 5, and also let the MDtuples be represented in the hash tables to save space. The algorithm to process a general query will be the one discussed in Section 5.2 but extending the step of processing the LET clause as follows:

If the expression associated with the variable is a nested *XQuery* expression, encode the level number properly and assign the nested name as discussed above. Then, invoke the construction algorithm recursively to process the nested sub-query.

6. FORMULATING THE OUTPUT QUERY

Based on CanForest, we can identify the proper values and those collections which define them in the source schema. In this section, we discuss how to choose the proper structural expressions to connect those collections, and how to formulate the final output query. Query *XQ*1 and its corresponding CanForest structure in Fig. 5 will be used as the examples for explanation.

6.1 Identifying Structural Expressions

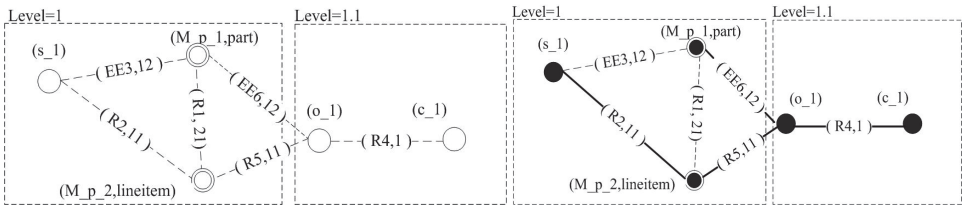
The Axis edges and FlatJoin edges in ColForest show the structural constraint imposed by the input query. The main issue is how to identify the proper corresponding structural expressions in the source schema. To achieve this task, we first create a *join graph* based on the internal nodes in ColForest, as defined in the following:

Definition 11: A *join graph* is a pair of (*N*, *E*). Each node in *N* corresponds to an MDtuple with the form (*var*, *equi-unit*, *usedflag*), and is identified by *var*. An edge in *E* will be created between two nodes, if there is a proper structural expression between the two associated equi-units. Each edge will be annotated by the pair (*ID*, *cost*), where the *ID* and *cost* information correspond to the first and the last columns of Table 1.

To construct the join graph, we examine each internal node of ColForest as follows. If a certain MDtuple associated with that node has the *usedflag* with the value "T", we will only use that MDtuple to create a corresponding node in the join graph. Otherwise, all the associated MDtuples will be used to create nodes. Fig. 6 (a) shows the initial join graph built based on Fig. 5. Note that the two MDtuples associated with node *M_p* both have the value "F" for *usedflag*, so we create two corresponding nodes in the join graph, and especially denote them by double circles. Besides, recall that the original *cost* is based on the number of the join expressions required to construct the structural relationship. In the join graph, if the associated node has the false value for the flag *usedflag*, we further increase the cost by 10 to represent that this edge is less preferred. As depicted in Fig. 6 (a), the edge with the ID "*EE*6" has the cost 12, which is calculated by adding 10 to 2.

Note that the join graph is also classified into several level blocks the same as in CanForest. Besides, the sub-query in each level needs to be joined properly, and the outer and the nested levels also need correlated join expressions. Therefore, we will identify

the join expressions level by level from the inner to the outer. Besides, we apply the algorithm of finding the *minimum spanning tree* to reduce the number of outputted join expressions, since the cost denoted with each edge corresponds to the number of join expressions. For Fig. 6 (a), we first process the join graph with the level number "1.1". It is trivial since there is only one edge. We then process the join graph "1", and the resultant minimum spanning tree is shown by the solid edge in Fig. 6 (b). However, we consider a node to be *necessary*, if it either consists of an attribute to output, or it is required to construct the structural relationship between two other collections. Therefore, if a node has the false value for *usedflag*, and only has one associated edge, we will delete it along with the associated edge. To conclude, the final output for this example will be edge *R*4 for level 1.1, edge *R*2 for level 1, and edge *R*5 for correlating the two levels.



(a) The initial state. (b) The minimum spanning tree.

Fig. 6. Examples of the join graphs.

6.2 Formulating the Query

At this stage, all the required mapping data are identified. We will insert the proper keywords, and combine those intermediate query fragments as a syntactically correct statement with the proper nesting level. It is basically straightforward and we will only use examples to illustrate the major idea.

Table 2. The process of transforming *XQ*1.

Sub-query	Step 1	Step 2
Outer	SELECT s1.SUPPKEY, a FROM SUPPLIER as s1	SELECT s1.SUPPKEY, a FROM SUPPLIER as s1, **LINEITEM as Mp2** WHERE s1.SUPPKEY =Mp2.SUPPKEY (R2) and **ORDER.ORDERKEY** = Mp2.ORDERKEY (R5)
Nested	SELECT c1.NAME FROM ORDER as o1, CUSTOMER as c1	SELECT c1.NAME FROM ORDER as o1, CUSTOMER as c1 WHERE o1.CUSTKEY = c1.CUSTKEY (R4)

Consider the running example. Table 2 summarizes the process of forming and composing query fragments for *XQ*1. In Step 1, we produce the query fragment based on CanForest, where the nodes in ColForest will be used to construct the FOR clause and the WHERE clause, and the nodes in ValForest are used for the SELECT clause. The query fragments formulated respectively for the outer level and the nested level are listed in the second column of Table 2. Note that at this stage, the node *M_p* has no corresponding output statements since both of its associated MDtuples have the *usedflag* as "*F*". In Step 2, we utilize those structural expressions outputted by the minimum span-

ning tree of the join graph, and insert them into the previous query fragments. The result is shown in the third column of Table 2. Note that *R*2 introduces a new relation LINEITEM, which is added into the FROM clause. Besides, *R*5 shows that the correlated value is ORDER.ORDERKEY. Recall that the sub-query statement for the nested level is aliased as "*a*", as shown in Fig. 5. By properly using the information for correlation, we get the final SQL statement as *SQ*1 different only in variable naming.

7. EVALUATION

In this section, we will first show the effectiveness of our system by proving that it can produce the required query. We then evaluate the efficiency of our translator system. All experiments are performed on a P4-2.4GHz machine, with 2 GB of DDR2-RAM.

7.1 Effectiveness

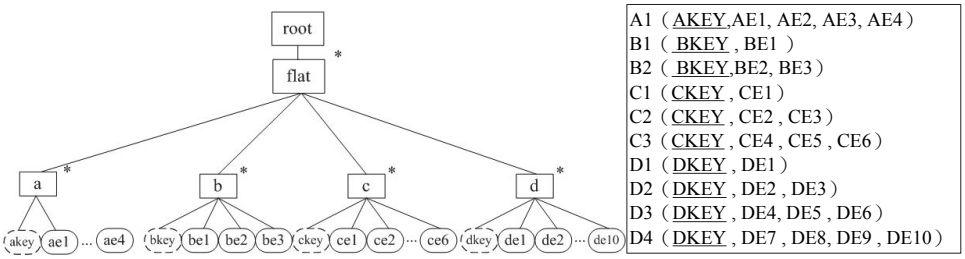
We prove that our system can produce the required query based on the *problem definition* specified in Section 3. Specifically, the following two things have to be satisfied: (1) the output query should be appropriate for the source schema and equivalent to the input query; (2) the weight of the output query should be the highest among all candidates.

For the first requirement, recall that when constructing CanForest, we identify a single equivalent counterpart for each value construct, and then identify those equivalent collection constructs which define the chosen values. Then, in the composition stage, we choose proper structural expressions for joining the collections in the same level of sub-queries, and also those for correlating the outer and nested sub-queries. Therefore, our transformed query is equivalent to the input query and appropriate for the source schema.

For the second requirement, recall that the weight of a query is determined by its operated value constructs. As discussed in Section 5, when determining the equivalent counterpart for each value construct, we choose the one with the highest weight. Therefore, it is guaranteed that the sum of all weights will be the highest among all equivalent transformed queries.

7.2 The Effect of Multiple Mappings on Efficiencies

We study the impact of multiple mappings on the transformation time in this set of experiments. We will measure the total execution time and the time required by the two major component modules of our system as shown in Fig. 3. Since we cannot find a set of equivalent schemas in the real world which can meet our needs, we design the schemas by ourselves. The target XML schema is shown in Fig. 7 (a), where square nodes represent internal nodes, rounded square nodes represent value elements, and dashed circles represent attributes. The source relational schema is shown in Fig. 7 (b), where primary keys are denoted by the underline. In this pair of schemas, the value with the same name is equivalent. Therefore, the repeatable element "*/root/flat/a*" corresponds to relation *A*1; "*/root/flat/b*" corresponds to relations *B*1 and *B*2; "*/root/flat/c"* corresponds to relations *C*1, *C*2, and *C*3; "*/root/flat/d*" corresponds to relations *D*1, *D*2, *D*3, and *D*4.



(a) Target schema (b) Source schema

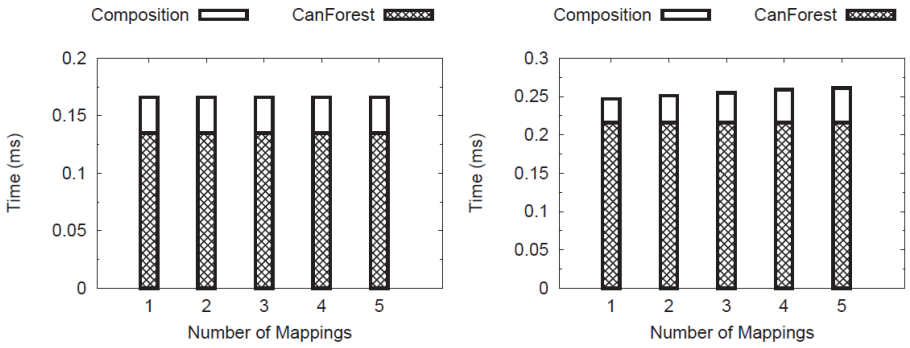
Fig. 7. Test schemas.

Table 3. Sample Input and Output Queries for Schemas I.

Type	Input Query	Output Query
value 1-1	FOR $a in /flat/a RETURN $a/@akey	SELECT a1.AKEY FROM A1 as a1
value 1-2	FOR $b in /flat/b RETURN $b/@bkey	SELECT b2.BKEY FROM B2 as b2
collection 1-1	FOR $a in /flat/a RETURN $a/@akey,$a/ae1,$a/ae2, $a/ae3	SELECT a1.AKEY, a1.AE1, a1.AE2, a1.AE3 FROM A1 as a1
collection 1-2	FOR $b in /flat/b RETURN $b/@bkey, $b/be1, $b/be2, $b/be3	SELECT b1.BKEY, b1.BE1, b2.BE2, b2.BE3 FROM B1 as b1, B2 as b2 WHERE b1.BKEY = b2.BKEY

We consider the following two cases: multiple mapping between *values* and multiple mapping between *collections*. Table 3 lists the input and output queries for some of the cases. Compare the two input queries specified in the first two rows of the table, which only return one value from one collection. Observe the value *bkey* used in the second row. It is represented by one unit in the target schema, but two units in the source schema, so this constitutes a 1-2 value mapping. In contrast, compare the two input queries represented in the last two rows of the table. They both return four values from the same element. However, since the equivalent value counterparts used in the last query are represented respectively in two relations *B*1 and *B*2, the output queries will need to include both of them and connect them by a join statement. Therefore, we name it a 1-2 collection mapping.

We design five queries with increasing numbers of mappings between values. The experimental result is shown in Fig. 8 (a). We can see that the transformation time remains almost constant, since only one value is outputted, and we use the hash function to directly identify the value with the highest weight. We then design five queries with increasing numbers of mappings between collections. From Fig. 8 (b), we can see that the time for constructing CanForest is almost the same for all queries, since they are basically in the same format. In contrast, the time for composition increases linearly slightly, since the number of output collection and structure expressions will increase along. Specifically, if a collection in a target schema maps to n collections in the source schema, the output query will consist of n collections in the FROM clause and $n-1$ join statements in the WHERE clause. However, our system is efficient enough so that the increased time is pretty minor.

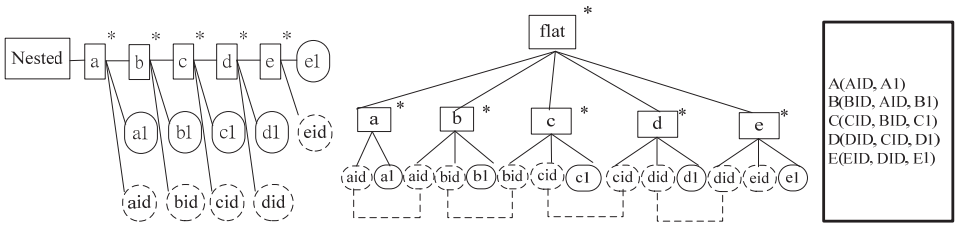


(a) Multiple values (b) Multiple collections

Fig. 8. Analysis of the effect caused by the number of mapping units.

7.3 The Effect of Nested Depths on Efficiencies

In this set of experiments, we study the impact of the nesting depths of sub-queries. The test schemas consist of one relational schema and two XML schemas, which are called "Nested" and "Flat", respectively. The two XML schemas are depicted in Figs. 9 (a)-(b), and differ in that the repeatable elements, denoted by the asterisk, are in a nested structure in Fig. 9 (a), but in a flat structure in Fig. 9 (b). The relational schema is listed in Fig. 9 (c), where the adjacent tables have common attributes for performing joins.



(a) Nested XML schema (b) Flat XML schema (c) RDB

Fig. 9. Test schemas II.

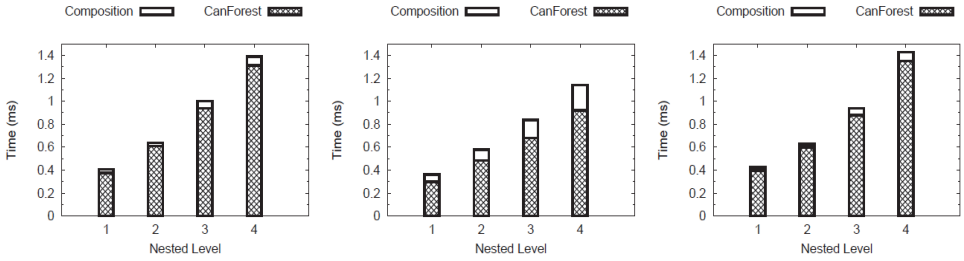
Table 4. Sample input queries for schemas II.

XQuery (for Nested DTD)	XQuery (for Flat DTD)	SQL
for $a in /nested/a let $leta ::= for $b in $a/b let $letb ::= for $c in $b/c return $c/c1, $c/@cid return $b/b1, $letb/c1 return $a/a1, $leta/b1	for $a in /flat/a let $leta ::= for $b in /flat/b let $letb ::= for $c in /flat/c where $b/@bid = $c/@aid return $c/c1, $c/@cid where $a/@aid = $b/@aid return $b/b1, $letb/c1 return $a/a1, $leta/b1	SELECT A.A1, SUB.CID FROM A, (SELECT SUBC.cid, B.aid FROM B, (SELECT C.C1, C.BID FROM C) as SUBC WHERE B.bid=SUBC.bid) as SUBB WHERE A.aid=SUBB.aid

We create four queries with increasing nested depths of sub-queries for each schema. We only list the query with the nested level "two" in Table 4 to illustrate the idea. For comparison, we give the number of nodes in CanForest for each query here. For the nested XML schema, the number of nodes are 10, 15, 20, and 25, respectively; for the

flat schema, the number of nodes are 13, 20, 27, and 32, respectively; for the relational schema, the number of nodes are 11, 18, 25, and 30, respectively. We then design the following three scenarios: (1) the target schema is relational and the source schema is the nested XML schema; (2) the target schema is the nested XML schema and the source schema is relational; (3) the target schema is the nested XML schema and the source schema is relational.

From the experimental results shown in Fig. 10, we can see that the transformation time mainly increases linearly along with the depth of the sub-query, no matter which is the input schema. It is reasonable since the nodes in CanForest increase along with the nested depths of sub-queries in each case, as stated previously. In other words, the format of nested sub-queries does not incur extra overhead for computation. The main reason is that our system processes nested queries or un-nested queries in a uniform way, so it can achieve the same performance when processing both types of queries. Another thing to note is that the major portion of execution time is on building the CVForest, and the flat and the relational schema require more time, since the corresponding queries tend to use valued join expressions, and have more nodes and edges in CanForest.



(a) Relational-nested (b) Nested-relational (c) Flat-relational

Fig. 10. Analysis of the effect caused by the nesting depths of subqueries.

To conclude, the performance of our translator system is satisfiable since the transformation process is very efficient. Moreover, the cases of multiple mappings and nested depths of queries only affect the performance to a limited extent.

8. CONCLUSIONS

In this paper, we discuss the issue of translating an input query against the target schema to an output query appropriate for the source schema. The queries considered are XQuery or SQL with SPJ expressions and nested sub-queries, and there might exist multiple possible mappings between the values of two schemas denoted with weights. We propose to use mapping functions to represent the weighted correspondence between schema units, and design the tree-based structure, *i.e.*, CanForest, to clearly represent the semantics and nesting levels of an input query. Accordingly, a translation system is constructed to produce the most preferable query under such environment. The experimental results show its efficiency. In the future, we plan to extend our system to efficiently produce Top-K preferable queries. We will also investigate the techniques of processing more complex queries and schema expressions.

REFERENCES

1. B. Alexe, W. C. Tan, and Y. Velegrakis, "Stbenchmark: Towards a benchmark for mapping systems," in *Proceedings of International Conference on Very Large Databases*, 2008, pp. 230-244.
2. Y. An, A. Borgida, R. J. Miller, and J. Mylopoulos, "A semantic approach to discovering schema mapping expressions," in *Proceedings of International Conference on Data Engineering*, 2007, pp. 206-215.
3. M. Arenas and L. Libkin, "Xml data exchange: Consistency and query answering," in *Proceedings of International Conference on Principles of Database System*, 2005, pp. 13-24.
4. A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou, "Structured materialized views for xml queries," in *Proceedings of International Conference on Very Large Databases*, 2007, pp. 87-98.
5. R. Cheng, J. Gong, and D. W. Cheung, "Managing uncertainty of xml schema matching," in *Proceedings of International Conference on Data Engineering*, 2010, pp. 297-308.
6. D. DeHaan, D. Toman, M. P. Consens, and M. T. Ozsu, "A comprehensive xquery to sql translation using dynamic interval encoding," in *Proceedings of International Conference on Management of Data*, 2003, pp. 623-634.
7. D. W. Embley, L. Xu, and Y. Ding, "Automatic direct and indirect schema mapping: experiences and lessons learned," *SIGMOD Record*, Vol. 33, 2004, pp. 14-19.
8. W. Fan, J. X. Yu, J. Li, B. Ding, and L. Qin, "Query translation from XPath to SQL in the presence of recursive DTDs," *The VLDB Journal*, Vol. 18, 2009. pp. 857-883.
9. M. A. Hernandez, P. Papotti, and W.-C. Tan, "Data exchange with data-metadata translations," in *Proceedings of International Conference on Very Large Databases*, 2008, pp. 260-273.
10. R. Krishnamurthy, R. Kaushik, and J. F. Naughton, "Efficient XML-to-SQL query translation: Where to add the intelligence?" in *Proceedings of International Conference on Very Large Databases*, 2004, pp. 144-155.
11. L. V. S. Lakshmanan, H. Wang, and Z. J. Zhao, "Answering tree pattern queries using views," in *Proceedings of International Conference on Very Large Databases*, 2006, pp. 571-582.
12. M. L. Lee, L. H. Yang, W. Hsu, and X. Yang, "Xclust: Clustering xml schemas for effective integration," in *Proceedings of International Conference on Information and Knowledge Management*, 2002, pp. 292-299.
13. I. Manolescu, K. Karanasos, V. Vassalos, and S. Zoupanos, "Efficient xquery rewriting using multiple views," in *Proceedings of International Conference on Data Engineering*, 2011, pp. 972-983.
14. N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola, "Rewriting nested xml queries using nested views," in *Proceedings of International Conference on Management of Data*, 2006, pp. 443-454.
15. E. Peukert, J. Eberius, and E. Rahm, "A self-configuring schema matching system," in *Proceedings of International Conference on Data Engineering*, 2012, pp. 306-317.
16. E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *The VLDB Journal*, Vol. 10, 2001, pp. 334-350.

17. Y. Velegrakis, R. J. Miller, and J. Mylopoulos, "Representing and querying data transformations," in *Proceedings of International Conference on Data Engineering*, 2005, pp. 81-92.

Ya-Hui Chang (張雅惠) is a Professor of Computer Science and Engineering Department at National Taiwan Ocean University. She received the B.S. degree in Computer Science and Information Engineering from National Taiwan University, and the M.S. and Ph.D. degree in Computer Science from University of Maryland at College Park, U.S.A. Her research interests include database query processing, XML techniques, graph databases and spatial databases.

Jia-Zhen Li (李佳臻) received the B.S. and M.S. degrees in Computer Science Department from National Taiwan Ocean University, Taiwan, in 2006 and 2009, respectively. Her research interests include XML query optimization, XML query processing and database management.

Si-Yen Zhuang (莊思彥) is a graduate student of Computer Science and Engineering Department at National Taiwan Ocean University. He received the BS degree in Department of Computer Science and Information Engineering from Chinese Culture University, Taiwan, in 2013. His research interests include database management, SQL query processing and Web applications.