A Virtual XML Database Engine for Relational Databases

Chengfei Liu[1], Millist W. Vincent[1], Jixue Liu[1], Minyi Guo[2]

[1] University of South Australia, Adelaide, SA 5095, Australia
[2] The University of Aizu, Aizu-Wakamatsu City, ukushima, 965-8580, Japan

Abstract. While XML is emerging as the universal format for publishing and exchanging data on the Web, most business data is still stored and maintained in relational DBMSs. To enable eBusiness database applications, Web access to the legacy data managed by DBMSs needs to be provided. In this paper, we introduce a virtual XML database engine VXE-R which allows users query a relational database via XML as if they were accessing XML documents. Algorithms for schema transformation and query translation in VXE-R are presented.

1 Introduction

While XML [1,4] is emerging as the universal format for publishing and exchanging data on the Web, most business data is still stored and maintained in relational DBMSs. In fact, relational DBMSs will remain dominant in managing business data in foreseeable future because of their powerful data management services. However, relational databases are proprietary and only accessible within an enterprise. To enable eBusiness database applications, it is important for enterprises to publish their relational databases as XML documents given that XML documents are universally accessible.

A general approach to publish relational data is to create XML views of the underlying relational data. Once XML views are created over a relational database, there are two ways to use these views. A simple way is to materialize the XML views by physically creating the result XML documents specified by the views. Obviously, this may not applicable to a large view; otherwise tremendous amount of spaces may be used. Maintenance of the materialized views may also need extra computation. A better way is to support queries over XML views. SilkRoute [7] is one of the systems taking this approach. In SilkRoute, XML views of a relational database are defined using a relational to XML transformation language called RXL, and then XML-QL queries are issued against views. The queries and views are combined together by a query composer and the combined RXL queries are then translated into corresponding SQL queries. XPERANTO [5,10,11] takes a similar approach but uses XQuery [3] for user queries.

We take a different approach. Instead of defining views based on relational databases, we translate the underlying relational schema into equivalent XML schema. Then XML queries are issued directly against the XML schema. Schema mapping rules are designed to generate a normalized XML schema which bring no data redundancy from the underlying relational schema. The translated XML schema also preserves integrity constraints defined in a relational database schema. It is important for users to be aware of the constraints in the XML schema against which they are going to issue queries. In the SilkRoute and XPERANTO approaches, users cannot see the integrity constraints buried in the relational schema from the XML views defined. Another benifit of our proposed approach is that the query translation process gets simplified.

In this paper, we introduce a virtual XML database engine VXE-R which allows users query a relational database via XML as if they were accessing XML documents. VXE-R is composed of three components. A schema translator which translates the underlying relational schema into equivalent XML schema, a query translator which translates the XQuery queries against XML schema into the corresponding SQL queries against the underlying relational schema, and an XML document generator which converts SQL result tables into XML documents.

The rest of the paper is organized as follows. After the architecture of VXE-R is presented in Section 2, we discuss the translation from relational schema to XML schema in Section 3. The translation of XQuery queries to corresponding SQL queries is described in Section 4. The XML document generator is introduced in Section 5. Section 6 concludes the paper.

2 The Architecture

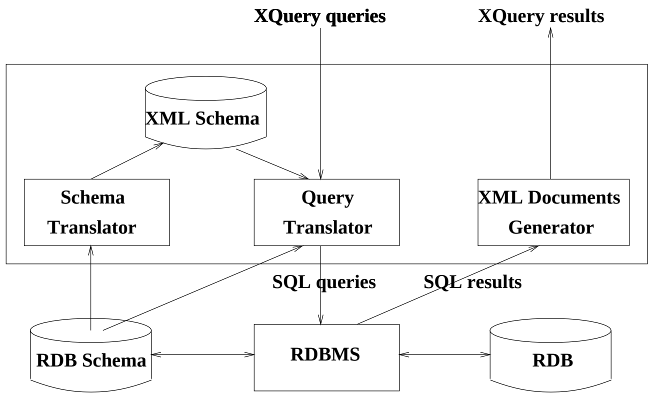

Fig. 1. Architecture of VXE-RF

The architecture of the virtual XML database engine VXE-R is shown in Figure 1. There are three components:

- A schema translator
- A query translator
- An XML document generator

The schema translator is responsible to translate a relational database schema into the corresponding schema in XML Schema. We choose XML Schema [6] because Data Type Definition (DTD) has a number of limitations, e.g., it is written in a non-XML syntax; it has no support of namespaces; it only offers extremely limited data typing. XML Schema is a more comprehensive and rigorous method for defining content model of an XML document. The schema itself is an XML document, and so can be processed by the same tools that read the XML documents it describes. XML Schema supports rich built-in types and allows complex types built based on built-in types. It also supports key and unique constraints which are important to map relational databases to XML documents.

Once an XML schema is created, user queries in XQuery can be formulated against it. As the real data is stored in relational databases, it is the responsibility of the query translator to translate the XQuery queries into the corresponding SQL queries against the underlying relational schema. The translated SQL queries are passed to a relational DBMS for execution. XQuery [3] is chosen as the XML query language since it is currently being standardized by the W3C.

After the execution of the translated SQL queries, the result relations are passed to the XML document generator which generates the result XML documents for users after possible re-structuring according to the requirements specified in the XQuery queries.

In the following sections, we describe these three components in detail.

3 Schema Translation

In a relational database schema, different types of integrity constraints may be defined, e.g., primary keys (PKs), foreign keys (FKs), null/not-null, unique, etc. It is important to map all these constraints to the target XML schema. Also we aim to achieve high level of nesting and to avoid introducing redundancy in the target schema.

Basically, the null/not-null constraint can be easily represented by properly setting *minOccurs* of the transformed XML element for the relation attribute. The unique constraint can also be represented by the unique mechanism in XML Schema straightforwardly. In the following, we first focus on the mapping of PK/FK constraints, then we consider further on the null/not-null and unique constraints.

XML Schema supports two mechanisms to represent identity and reference: one is ID/IDREF while the other is KEY/KEYREF. There are differences in using these two mechanisms. The former supports the dereference function in path expressions in most XML query languages including XQuery, however, it only applies to a single element/attributes. The latter may apply to multiple elements/attributes but cannot support the dereference function. For schema translation, we use ID/IDREF where possible because of the dereference function support. For this purpose, we will differentiate the single attribute primary/foreign keys from multi-attribute primary/foreign keys while transforming the relational database schema to XML schema. We also classify a relation into the following four categories based on different types of primary keys:

regular: the primary key of a regular relation contains no foreign keys.
component: the primary key of a component relation contains one foreign key which references its parent relation. The other part of the primary key serves as a local identifier under the parent relation. A component relation is used to represent a component or a multi-valued attribute of its parent relation.
supplementary: the primary key of a supplementary relation as a whole is also a foreign key which references another relation. This relation is used to supplement another relation or to represent a subclass for translating a generalization hierarchy from a conceptual schema.
association: the primary key of an association relation contains more than one foreign keys, each of which references a participant relation.

Based on above discussion, we give the set of mapping rules.

3.1 Basic Mapping Rules

Given a relational database schema *Sch* with primary/foreign key definitions, we may use the following basic mapping rules to convert *Sch* into a corresponding XML schema *Sch_XML*.

Rule 1 *For a relational database schema* Sch, *a root element named* Sch_XML *is created in the corresponding XML schema as follows.*

```
<xs: element name = "Sch_XML">_
  <xs: complexType>_
    <xs: sequence>_
      <!-- translated relation schema of Sch -->_
    </xs: sequence>_
  </xs: complexType>_
</xs: element>_
```

Rule 2 *For each regular or association relation R, the following element with the same name as the relation schema is created and then put under the root element.*

```
<xs: element name = "R" minOccurs = "0" maxOccurs = "unbounded">_
  <xs: complexType>_
    <xs: sequence>_
      <!-- the attributes of R -->_
    </xs: sequence>_
  </xs: complexType>_
</xs: element>_
```

Rule 3 *For each component relation R_1, let its parent relation be R_2, then an element with the same name as the component relation is created and then placed as a child element of R_2. The created element has the same structure as the element created in Rule 2.*

Rule 4 *For each supplementary relation R_1, let the relation which R_1 references be R_2, then the following element with the same name as the supplementary relation schema is created and then placed as a child element of R_2. The created element has the same structure as the element created in Rule 2 except that the* maxOccurs *is 1.*

Rule 5 *For each single attribute primary key with the name A of regular relation R, an attribute of the element for R is created with ID data type as follows.*

```
  <xs: attribute name = "PKA" type = "xs:ID"/>_
```

Rule 6 *For each multiple attribute primary key of a regular, a component or an association relation R, suppose the key attributes are A_1, , KA_n, an attribute of the element for R is created for each $A_i (1 \leq i \leq n)$ with the corresponding data type. If R is a component relation and A_i is a single attribute foreign key contained in the primary key, then the data type of the created attribute is* IDREF. *After that a* key *element is defined with a* selector *to select the element for R and several* fields *to identify A_1, , KA_n. The key element can be defined inside or outside the element for R. The name of the element should be unique within the namespace.*

```
<xs: element name = "R" minOccurs = "0" maxOccurs = "unbounded">_
  <xs: complexType>_
    <xs: attribute name = "PKA1" type = "xs:PKA1_type"/>
    ... ...
    <xs: attribute name = "PKAn" type = "xs:PKAn_type"/>_
  </xs: complexType>_
  <xs: key name = "PK"/>_
    <xs: selector xpath = "R/"/>_
    <xs: field xpath = "@PKA1"/>
    ... ...
    <xs: field xpath = "@PKAn"/>_
  </xs: key>_
</xs: element>_
```

Rule 7 *Ignore the mapping for primary key of each supplementary relation.*

Rule 8 *For each single attribute foreign key A of a relation R except one which is contained in the primary key of a component or supplementary relation, an attribute of the element for R is created with IDREF data type.*

```
  <xs: attribute name = "FKA" type = "xs:IDREF"/>_
```

Rule 9 *For each multiple attribute foreign key of a relation R except one which is contained in the primary key of a component or supplementary relation, suppose references of the referenced relation, and the foreign key attributes are* $_1$, , $_n$*, an attribute of the element for R is created for each* $_i(1 \leq i \leq n)$ *with corresponding data type. Then a* keyref *element is defined with a* selector *to select the element for R and several* fields *to identify* $_1$, , $_n$. *The keyref element can be defined either inside or outside the element. The name of the element should be unique within the namespace and* refer *of the element is the name of the key element of the primary key which it references.*

```
<xs: element name = "R" minOccurs = "0" maxOccurs = "unbounded">_
  <xs: complexType>_
    <xs: attribute name = "FKA1" type = "xs:FKA1_type"/>_

    <xs: attribute name = "FKAn" type = "xs:FKAn_type"/>_
  </xs: complexType>_
  <xs: keyref name = "FK" refer = "PK"/>_
    <xs: selector xpath = "R/"/>_
    <xs: field xpath = "@FKA1"/>_

    <xs: field xpath = "@FKAn"/>_
  </xs: keyref>_
</xs: element>_
```

Rule 10 *For each non-key attribute of a relation R, an element is created as a child element of R. The name of the element is the same as the attribute name.*

Rule 1 to Rule 10 are relatively straitforward for mapping a relational database schema to a corresponding XML schema. One property of these rules is redundancy free preservation, i.e., Rule 1 to Rule 10 do not introduce any data redundancy provided the relational schema is redundancy free.

Theorem 3.1 *If the relational database schema* Sch *is redundancy free, the XML schema* Sch_XML *generated by applying Rule 1 to Rule 10 is also redundancy free.*

This theorem is easy to prove. For a regular or an association relation R, an element with the same name R is created under the root element, so the relation R in *Sch* is isomorphically transformed to an element in *Sch_XML*. For a component relation R, a sub-element with the same name R is created under its parent R_p. Because of the foreign key constraint, we have the functional dependency $_R \rightarrow \; _{R_p}$, i.e., there is a many to one relationship from R to R_p, therefore it is impossible that a tuple of R is placed more than one time under different element of R_p. Similar to a component relation, there is no redundancy introduced for a supplementary relation.

3.2 An Example

Let us have a look of a relational database schema *Company* for a company. Primary keys are underlined while foreign keys are in *italic* font.

Employee(eno, name, city, salary, *dno*)
Dept(dno, dname, *mgrEno*)
DeptLoc(*dno*, city)
Project(pno, pname, city, *dno*)
WorksOn(*eno, pno*, hours)

Given this schema as an input, the following XML schema will be generated:

```
<xs:element name="Company_XML">_
 <xs:complexType>_
  <xs:sequence>_
   <xs:element name="Employee" minOccurs="0" maxOccurs="unbounded">_
    <xs:complexType>_
     <xs:sequence>_
      <xs:element name="name" type="xs:string"/>_
      <xs:element name="city" type="xs:string"/>_
      <xs:element name="salary" type="xs:int"/>_
     </xs:sequence>_
     <xs:attribute name="eno" type="xs:ID"/>_
     <xs:attribute name="dno" type="xs:IDREF"/>_
    </xs:complexType>_
   </xs:element>_
   <xs:element name="Dept" minOccurs="0" maxOccurs="unbounded">_
    <xs:complexType>_
     <xs:sequence>_
      <xs:element name="dname" type="xs:string"/>_
      <xs:element name="city" type="xs:string"/>_
      <xs:element name="DeptLoc" minOccurs="0" maxOccurs="unbounded">_
       <xs:complexType>_
        <xs:attribute name="dno" type="xs:IDREF"/>_
        <xs:attribute name="city" type="xs:string"/>_
       </xs:complexType>_
       <xs:key name="PK_DeptLoc"/>_
        <xs:selector xpath="Dept/DeptLoc/"/>_
        <xs:field xpath="@dno"/>_
        <xs:field xpath="@city"/>_
       </xs: key>_
      </xs:element>_
     </xs:sequence>_
     <xs:attribute name="dno" type="xs:ID"/>_
     <xs:attribute name="mgrEno" type="xs:IDREF"/>_
    </xs:complexType>_
   </xs:element>_
   <xs:element name="Project" minOccurs="0" maxOccurs="unbounded">_
    <xs:complexType>_
     <xs:sequence>_
      <xs:element name="pname" type="xs:string"/>_
      <xs:element name="city" type="xs:string"/>
     </xs:sequence>
     <xs:attribute name="pno" type="xs:ID"/>
     <xs:attribute name="dno" type="xs:IDREF"/>
    </xs:complexType>
   </xs:element>
   <xs:element name="WorksOn" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
     <xs:element name="hours" type="xs:int"/>
     <xs:attribute name="eno" type="xs:IDREF"/>
     <xs:attribute name="pno" type="xs:IDREF"/>
     <xs:key name="PK_WorksOn"/>
      <xs:selector xpath="WorksOn/"/>
      <xs:field xpath="@eno"/>
      <xs:field xpath="@pno"/>
     </xs: key>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>_
```

The root element *Company_XML* is created for the relational database schema *Company*. Under the root element, four set elements *Employee, Dept, Project* and *WorksOn* are created for relation schema *Employee, Dept, Project* and *WorksOn*, respectively. For component relation schema *DeptLoc*, element *DeptLoc* is created under element *Dept* for its parent relation. PK/FK constraints in the relational database schema *Company* have been mapped to the XML schema *Company_XML* by using ID/IDREF and KEY/FEYREF.

3.3 Exploring Nested Structures

As we can see, the basic mapping rules fail to explore all possible nested structures. For example, the *Project* element can be moved to under the *Dept* element if every project belongs to a department. Nesting is important in XML schema because it allows navigation of path expressions to be processed efficiently. If we use IDREF instead, we may use system supported dereference function to get the referenced elements. In XML, the dereference function is expensive because ID and IDREF types are value based. If we use KEYREF, we have to put an explicit *join* condition in an XML query to get the referenced elements. Therefore, we need to explore all possible nested structure by investigating the referential integrity constraints in the relational schema. For this purpose, we introduce a reference graph as follows:

Definition 3.1 *: Given a relational database schema* $ch = \{R_1, \quad , R_n\}$*, a* reference graph *of the schema* ch *is defined as a labeled directed graph* $RG = (V, \quad , L)$ *where* V *is a finite set of vertices representing relation schema* $R_1, \quad , R_n$

in h; is a finite set of arcs, if there is a foreignkey defined in R_i which references R_j, an arc $e = R_i, R_j > \in$; is a set of labels for edges by applying a labeling function from to the set of attribute names for foreign keys.

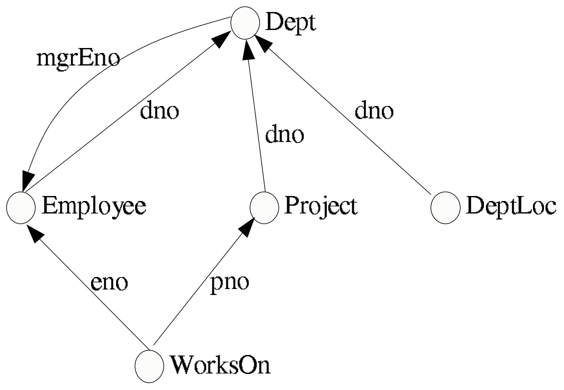

Fig. 2. A Reference GraphF

The reference graph of the relational schema *Company* is shown as in Figure 2. In the graph, the element of node *DeptLoc* has been put under the element of node *Dept* by Rule 3. From the graph, we may have the following improvement if certain conditions are satisfied.
(1) The element of node *Project* could be put under the element of node *Dept* if the foreign key *dno* is defined as NOT-NULL. This is because that node *Project* only references node *Dept* and a many to one relationship from *Project* to *Dept* can be derived from the foreign key constraint. In addition, the NOT-NULL foreign key means every project has to belong one department. As a result, one project can be put under one department and cannot be put twice under different departments in the XML document.
(2) A loop exists between *Employee* and *Dept.* What we can get from this is a many to many relationship between *Employee* and *Dept.* In fact, the foreign key *mgrEno* of Dept reflects a one to one relationship from *Dept* to *Employee.* Fortunately, this semantics can be captured by checking the *unique* constraint defined for the foreign key *mgrno.* If there is such a unique constraint defined, the foreign key *mgrEno* of Dept really suggests a one to one relationship from *Dept* to *Employee.* For the purpose of nesting, we delete the arc from Dept to Employee labelled *mgrno* from the reference graph. The real relationship from *Employee* to *Dept* is many to one. As such, the element of the node *Employee* can also be put under the element of the node *Dept* if the foreign key *dno* is defined to NOT-NULL.
(3) The node *WorksOn* references two nodes *Employee* and *Project.* The element of *WorksOn* can be put under either *Employee* and *Project* if the corresponding foreign key is NOT-NULL. However, which node to choose to put under all depends on which path will be used often in queries. We may leave this decision to be chosen by a designer.

Based on the above discussion, we can improve the basic mapping rules by the following theorems.

Theorem 3.2 *In a reference graph* RG, *if a node n_1 for relation R_1 has only one outcoming arc to another node n_2 for relation R_2 and foreign key denoted by the label of the arc is defined as NOT-NULL and there is no loop between n_1 and n_2, then we can move the element for R_1 to under the element for R_2 without introducing data redundancy.*

The proof of this theorem has already explained by the relationships between *Project* and *Dept*, and between *Dept* and *Employee* in Figure 2. The only arc from n_1 to n_2 and there is no loop between the two nodes represents a many to one relationship from R_1 to R_2, while the NOT-NULL foreign key gives a many to exact one relationship from R_1 to R_2. Therefore, for each instance of R_1, it is put only once under exactly one instance of R_2, no redundancy will be introduced.

Similarly, we can have the following.

Theorem 3.3 *In a reference graph* RG, *if a node n_0 for relation R_0 has outcoming arcs to other nodes $n_1, \quad , n_k$ for relations $R_1, \quad , R_k$, respectively, and the foreign key denoted by the label of at least one such outcoming arcs is defined as NOT-NULL and there is no loop between n_0 and any of $n_1, \quad , n_k$, then we can move the element for R_0 to under the element for R_i $(1 \leq i \leq k)$ without introducing data redundancy provided the foreign key defined on the label of the arc from n_0 to n_i is NOT-NULL.*

Rule 11 *If there is only one many to one relationship from relation R_1 to another relation R_2 and the foreign key of R_1 to R_2 is defined as NOT-NULL, then we can move the element for R_1 to under the element for R_2 as a child element.*

Rule 12 *If there are more than one many to one relationship from relation R_0 to other relations $R_1, \quad , R_k$, then we can move the element for R_0 to under the element for R_i $(1 \leq i \leq k)$ as a child element provided the foreign key of R_0 to R_k is defined as NOT-NULL.*

By many to one relationship from relation R_1 to R_2, we mean that there is one arc which cannot be deleted from node n_1 for R_1 to node n_2 for R_2, and there is no loop between n_1 and n_2 in the reference graph. If we apply Rule 11 to the transformed XML schema *Company_XML*, the elements for *Project* and *Employee* will be moved to under *Dept* as follows, the attribute *dno* with IDREF type can be removed from both *Project* and *Employee* elements.

```
<xs:element name="Dept" minOccurs="0" maxOccurs="unbounded">_
 <xs:complexType>_
  <xs:sequence>_
   <xs:element name="dname" type="xs:string"/>_
   <xs:element name="city" type="xs:string"/>_
   <xs:element name="DeptLoc" minOccurs="0" maxOccurs="unbounded">_
```

```
     <xs:complexType>
      <xs:attribute name="dno" type="xs:IDREF"/>
      <xs:attribute name="city" type="xs:string"/>
     </xs:complexType>_
     <xs:key name="PK_DeptLoc"/>
      <xs:selector xpath="Dept/DeptLoc/"/>
      <xs:field xpath="@dno"/>
      <xs:field xpath="@city"/>
     </xs: key>
    </xs:element>
    <xs:element name="Project" minOccurs="0" maxOccurs="unbounded">
     <xs:complexType>_
      <xs:sequence>
       <xs:element name="pname" type="xs:string"/>
       <xs:element name="city" type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="pno" type="xs:ID"/>
     </xs:complexType>
    </xs:element>
    <xs:element name="Employee" minOccurs="0" maxOccurs="unbounded">
     <xs:complexType>_
      <xs:sequence>
       <xs:element name="name" type="xs:string"/>
       <xs:element name="city" type="xs:string"/>
       <xs:element name="salary" type="xs:int"/>
      </xs:sequence>
      <xs:attribute name="eno" type="xs:ID"/>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
   <xs:attribute name="dno" type="xs:ID"/>
   <xs:attribute name="mgrEno" type="xs:IDREF"/>
  </xs:complexType>_
</xs:element>_
```

XML Schema offers great flexibility in modeling documents. Therefore, there exist many ways to map a relational database schema into a schema in XML Schema. For examples, XViews [2] constructs graph based on PK/FK relationship and generate candidate views by choosing node with either maximum in-degree or zero in-degree as root element. The candidate XML views generated achieve high level of nesting but suffer considerable level of data redundancy. NeT [8] derives nested structures from flat relations by repeatedly applying *nest* operator on tuples of each relation. The resulting nested structures may be useless because the derivation is not at the type level. Compared with XViews and NeT, our mapping rules can achieve high level of nesting for the translated XML schema while introducing no data redundancy provided the underlying relational schema is redundancy free.

4 Query Translation

In this section, we discuss how XQuery queries are translated to corresponding SQL queries. SQL is used to express queries on flat relations, where a join operation may be used frequently to join relations together; while XQuery is used to express queries on elements which could be highly nested by sub-elements or linked by IDREF, where navigation via path expression is the main means to link elements of a document together. As XQuery is more powerful and flexible than SQL, it is hard to translate an arbitrary XQuery query to corresponding SQL query. Fortunately, in VXE-R, the XML schema is generated from the underlying relational database schema, therefore, the structure of the mapped XML elements is *normalized*. Given the mapping rules introduced in Section 3, we know the reverse mapping which is crucial for translating queries in XQuery to the corresponding queries in SQL.

As XQuery is still in its draft version, in this paper, we only consider the translation of basic XQuery queries which do not include aggregate functions. The main structure of an XQuery query can be formulated by an FLWOR expression with the help of XPath expressions. An FLWOR expression is constructed from FOR, LET, WHERE, ORDER BY, and RETURN clauses. FOR and LET clauses serve to bind values to one or more variables using (path) expressions. The FOR clause is used for iteration, with each variable in FOR iterates over the nodes returned by its respective expression; while the optional LET clause binds a variable to an expression without iteration, resulting in a single binding for each variable. As the LET clause is usually used to process grouping and aggregate functions, the processing of the LET clause is not discussed here. The optional WHERE clause specifies one or more conditions to restrict the binding-tuples generated by FOR and LET clauses. The RETURN clause is used to specify an element structure and to construct the result elements in the specified structure. The optional ORDER BY clause determines the order of the result elements.

A basic XQuery query can be formulated with a simplified FLWOR expression:

```
FOR x1 in p1,       xn in pn_                   ,
WHERE c_
RETURN s_
```

In the FOR clause, iteration variables $x_1, \quad , x_n$ are defined over the path expressions $p_1, \quad , p_n$. In the WHERE clause, the expression c specifies conditions for qualified binding-tuples generated by the iteration variables. Some conditions may be included in p_i to select tuples iterated by the variable x_i. In the RETURN clause, the return structure is specified by the expression s. A nested FLWOR expression can be included in s to specify a *subquery* over sub-elements.

4.1 The Algorithm

Input A basic XQuery query Q_{xquery} against an XML schema *Sch_XML* which is generated from the underlying relational schema *Sch*.

Output A corresponding SQL query Q_{sql} against the relational schema *Sch*.
Step 1: *make Q_{xquery} canonical* - Let p_i defined in the FOR clause be the form of $/step_{i1}/ \quad /step_{ik}$. We check whether there is a test condition, say c_{ij} in $step_{ij}$ of p_i from left to right. If there is such a step, let $step_{ij}$ be the form of $l_{ij}[c_{ij}]$, then we add an extra iteration variable y_{ij} in the FOR clause which is defined over the path expression $/l_{i1}/ \quad /l_{ij}$, and move the condition c_{ij} to the WHERE clause, each element or attribute in c_{ij} is prefixed with $\$y_{ij}/$.
Step 2: *identify all relations* - After Step 1, each p_i in the FOR clause is now in the form of $/l_{i1}/ \quad /l_{ik}$, where $l_{ij}(1 \leq j \leq k)$ is an element in *Sch_XML*. Usually p_i corresponds to a relation in *Sch* (l_{ik} matches the name of a relation schema in *Sch*). The matched relation name l_{ik} is put in the FROM clause of Q_{sql} followed by the iteration variable x_i served as a tuple variable for relation l_{ik}. If there is an iteration variable, say x_j, appears in p_i, replace the occurrence of x_j with p_j. Once both relations, say R_i and R_j, represented by p_i and p_j respectively are identified, a link from R_i to R_j is added in a temporary list *LINK*. If there are nested FLWOR expressions defined in RETURN clause, the relation identification process is applied recursively to the FOR clause of the nested FLWOR expressions.
Step 3: *identify all target attributes for each identified relation* - All target attributes of Q_{sql} appear in the RETURN clause. For each leaf element (in the form of $\$x_i/t$) or attribute (in the form of $\$x_i/@t$) defined in s of the RETURN clause, replace it with a relation attribute in the form of $x_i.t$. Each identified target attribute is put in the SELECT clause of Q_{sql}. If there are nested FLWOR expressions defined in RETURN clause, the target attribute identification process is applied recursively to the RETURN clause of the nested FLWOR expressions.
Step 4: *identify conditions* - Replace each element (in the form of $\$x_i/t$) or attribute (in the form of $\$x_i/@t$) in the WHERE clause of Q_{xquery}, then move all conditions to the WHERE clause of Q_{sql} with a relation attribute in the form of $x_i.t$. If there are nested FLWOR expressions defined in RETURN clause, the condition identification process is applied recursively to the WHERE clause of the nested FLWOR expressions.
Step 5: *set the links between iteration variables* - If there is any link put in the temporary list LINK, then for each link from R_i to R_j, create a join condition between the foreign key attributes of R_i and the corresponding primary key attributes of R_j and ANDed to the other conditions of the WHERE clause of Q_{sql}.

4.2 An Example

Suppose we want to find all departments which have office in Adelaide and we want to list the name of those departments as well as the name and salary of all employees who live in Adelaide and work in those departments. The XQuery query for this request can be formulated as follows:

```
FOR $d in /Dept, $e in $d/Employee, $l in $d/DeptLoc_
```

```
WHERE $l/city = "Adelaide" and_
      $e/city = "Adelaide" and_
      $e/@dno = $d/@dno_
RETURN_
      <Dept>_
        <dname> $d/dname </dname>_
        <employees>_
           <name> $e/name </name>_
           <salary> $e/salary </salary>_
        </employees>_
      </Dept>_
```

Given this query as an input, the following SQL query will be generated:

```
SELECT d.dname, e.name, e.salary_
FROM Dept d, Employee e, DeptLoc l_
WHERE l.city = "Adelaide" and_
      e.city = "Adelaide" and_
      e.dno = d.dno and_
      l.dno = d.dno_
```

5 XML Documents Generation

As seen from the query translation algorithm and example introduced in the previous section, the translated SQL query takes all leaf elements or attributes defined in an XQuery query RETURN clause and output them in a flat relation. However, users may require a nested result structure such as the RETURN structure defined in the example XQuery query. Therefore, when we generate the XML result documents from the translated SQL query result relations, we need to restructure the flat result relation by a *grouping* operator [9] or a *nest* operator for N^2 relations, then convert it into XML documents.

Similar to SQL *GROUP BY* clause, the grouping operator divides a set or list of tuples into groups according to key attributes. For instance, suppose the translated SQL query generated from the example XQuery query returns the following result relation as shown in Table 1. After we apply grouping on the relation using dname as the key, we have the nested relation as shown in Table 2 which can be easily converted to the result XML document as specified in the example XQuery query.

dname	**name**	**salary**
development	Smith, John	70,000F
marketing	Mason, Lisa	60,000F
development	Leung, Mary	50,000F
marketing	Lee, Robert	80,000F
development	Chen, Helen	70,000F

Table 1. Relation Example

dname	**name**	**salary**
development	Smith, JohnF	70,000F
	Leung, MaryF	50,000F
	Chen, HelenF	70,000F
marketing	Mason, LisaF	60,000F
	Lee, RobertF	80,000F

Table 2. Nested Relation ExampleF

6 Conclusion and Future Work

This paper introduced the architecture and components of a virtual XML database engine VXE-R. VXE-R presents a normalized XML schema which preserves integrity constraints defined in the underlying relational database schema to users for queries. Schema mapping rules from relational to XML Schema were discussed. The Query translation algorithm for translating basic XQuery queries to corresponding SQL queries was presented. The main idea of XML document generation from the SQL query results was also discussed.

We believe that VXE-R is effective and practical for accessing relational databases via XML. In the future, we will build a prototype for VXE-R. We will also examine the mapping rules using our formal study of the mapping from relational database schema to XML schema in terms of functional dependencies and multi-valued dependencies [12, 13], and investigate the query translation of complex XQuery queries and complex result XML document generation.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers, 2000.F
2. C. Baru. Xviews: Xml views of relational schemas. In *Proceedings of DEXA Workshop*, pages 700–705, 1999.F
3. S. Boag, D. Chamberlin amd M. ernandez, D. J. Robie, J. Simeon, andF lorescu, M. Stefanescu. *XQuery 1.0: An XML Query Language*, April 2002. W3C WorkingF Draft, http://www.w3.org/TR/2002/WD-xquery-20020430/.F
4. T. Bray, J. Paoli, C. Sperberg-McQueen, and E. Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*, October 2000. W3C Recommendation,F http://www.w3.org/TR/REC-xml.F
5. M. Carey, J. Kiernan, J. Shanmugasundaram, E. Shekita, and S. Subramanian.F Xperanto: Middleware for publishing object-relational data as xml documents. InF *Proceedings of VLDB*, pages 646–648, 2000.F
6. D. allside. *XML Schema Part 0: Primer*, May 2001. W3C Recommendation,F http://www.w3.org/TR/xmlschema-0/.F
7. M. ernandez, W. Tan, and D. Suciu. Silkroute: Trading between relations andF xml. In *Proceedings of WWW*, pages 723–725, 2000.F
8. D. Lee, M. Mani, Chiu, and W. Chu. Nesting-based relational-to-xml schemaF translation. In *Proceedings of the WebDB*, pages 61–66, 2001.F
9. J. Liu and C. Liu. A declarative way of extracting xml data in xsl. In *Proceedings of ADBIS*, pages 374–387, September 2002.F
10. J. Shanmugasundaram, J. Kiernan, E. Shekita, C. an, and J. underburk. Querying xml views of relational data. In *Proceedings of VLDB*, pages 261–270, 2001.F
11. J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh,F and B. Reinwald. Efficiently publishing relational data as xml documents. InF *Proceedings of VLDB*, pages 65–76, 2000.F
12. M. Vincent, J. Liu, and C. Liu. A redundancy free 4nf for xml. In *Proceedings of XSYM*, September 2003.F
13. M. Vincent, J. Liu, and C. Liu. Redundancy free mapping from relations to xml.F In *Proceedings of WAIM*, August 2003.F