A Semantic Approach to Query Rewriting for Integrated XML Data

Xia Yang[1] Mong Li Lee[1] Tok Wang Ling[1] Gillian Dobbie[2]

[1]School of Computing, National University of Singapore
{yangxia,leeml,lingtw}@comp.nus.edu.sg

[2]Department of Computer Science, The University of Auckland, New Zealand
gill@cs.auckland.ac.nz

Abstract. Query rewriting is a fundamental task in query optimization and data integration. With the advent of the web, there has been renewed interest in data integration, where data is dispersed among many sources and an integrated view over these sources is provided. Queries on the integrated view are rewritten to query the underlying source repositories. In this paper, we develop a novel algorithm for rewriting queries that considers the XML hierarchy structure and the semantic relationship between the source schemas and the integrated schema. Our approach is based on the semantically rich Object-Relationship-Attribute model for SemiStructured data (ORA-SS), and guarantees that the rewritten queries give the expected results, even where the integrated view is complex.

1 Introduction

Many query rewriting algorithms have been developed for answering queries using views in relational databases and in mediators. When answering queries using views, the objective is to find efficient methods to answer a query using a set of materialized views over the database, instead of accessing the database itself [5, 14, 16, 17].

In data integration, many systems construct a global or mediated schema from numerous heterogeneous data sources [6, 13, 18]. Users issue queries on the global schema, and the system will rewrite the query to the local sources. Each local source may not necessarily contain all the information needed to answer the query. Partial results from various local sources are combined to produce the result for the query.

When integration is carried out over XML repositories, query rewriting algorithms need to take into consideration the hierarchical structure of XML schemas. This gives rise to structural conflicts which need to be resolved during the rewriting process [22]. XML schemas such as DTD and XML Schema lack the semantic information necessary for schema integration and query rewriting. Although proposals have been put forth to augment DTD and XML Schema with information such as keys [2], and functional dependencies [10], their semantics remain limited.

In this paper, we describe a rewriting algorithm for integrated views over XML repositories. The proposed algorithm utilizes the ORA-SS model [11] which provides the necessary semantic information to produce expected answers even when the integrated view is complex. In contrast to the work in [12] which describes how relational databases can be integrated into an XML global schema, we assume that the local sources are XML repositories. XML schemas are first transformed to ORA-SS schemas with enriched semantics [3]. An ORA-SS integrated schema can be obtained using the algorithm in [21]. Compared to existing global-as-view approaches which incorporate the integrated view definition in the unfolding process, our approach uses a mapping table that is created during the integration process to rewrite queries. We also use a query allocation table to find groups of local schemas that together can answer a user query. When a query is decomposed to subqueries on the local schemas, the subqueries for each group of local schemas are composed, and answers from the composed queries are combined to give the expected results.

The rest of the paper is organized as follows. Section 2 reviews the ORA-SS model. Section 3 describes the mapping table and the query allocation table. Section 4 gives the details of the proposed query rewriting algorithm. Section 5 compares our approach with related work and we conclude in Section 6.

2 ORA-SS Model

Our rewriting algorithm employs the ORA-SS model which is a semantically rich data model designed for semistructured data [11]. This model distinguishes between objects, relationships and attributes. An object class in the ORA-SS model is similar to the concept of an entity type in an ER model. An object class may be related to other object classes through a relationship type. Attributes are properties, and may belong to an object class or a relationship type. An attribute of an object class or relationship type in an ORA-SS schema may be represented as an attribute or sub-element in XML document. The main difference between the ORA-SS model and the ER model is that the ORA-SS model has a tree-like structure, which is more suitable for XML data. The nesting of the objects is reflected directly in the ORA-SS model. Other concepts that can be modeled in ORA-SS diagrams and not in ER diagrams include the ordering of elements and attributes, and fixed and default attribute values. An algorithm to translate XML schemas to the ORA-SS Schema Diagram is given in [3]. Note that user input may be needed to identify some semantics such as attributes of relationship types.

```
<project jno=”j01”>
   <part pno=”p01”>
      <supplier sno=”s01”>
         <quantity>500</quantity>
      </supplier>
   </part>
</project>
```

Fig. 1(a). An XML document

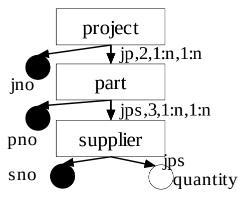

Fig. 1(b) ORA-SS schema of document in Fig. 1(a)

Fig. 1 shows an XML document and the corresponding ORA-SS schema diagram. Object classes “project” and “part” are denoted by labeled rectangles. The label “jps,3,1:n, 1:n” denotes a ternary relationship type “jps” involving object classes “project”, “part” and “supplier”, with parent cardinality 1:n and child cardinality 1:n.

That is, parts in a project may have one or more suppliers and a supplier can supply one or more parts to one or more project. Labeled circles denote attributes, and filled circles indicate identifiers. Attributes with labeled edges are relationship type attributes. For example, “jno” is an attribute of object class “project”, while “quantity” is an attribute of relationship type “jps”. Details on ORA-SS can be found in [11].

3 Mapping Table and Query Allocation Table

The proposed query rewriting algorithm utilizes two constructs: a mapping table and a query allocation table. A *mapping table* is created when an integrated schema is derived from the local schemas. This table contains the mappings from the integrated schema to the local schemas. We use the path-to-path mapping defined in [4]. The path from the root to the object class or attribute is captured in the mapping, so that one can tell the context of the object class or attribute. This will differentiate object classes and attributes with the same labels but different paths.

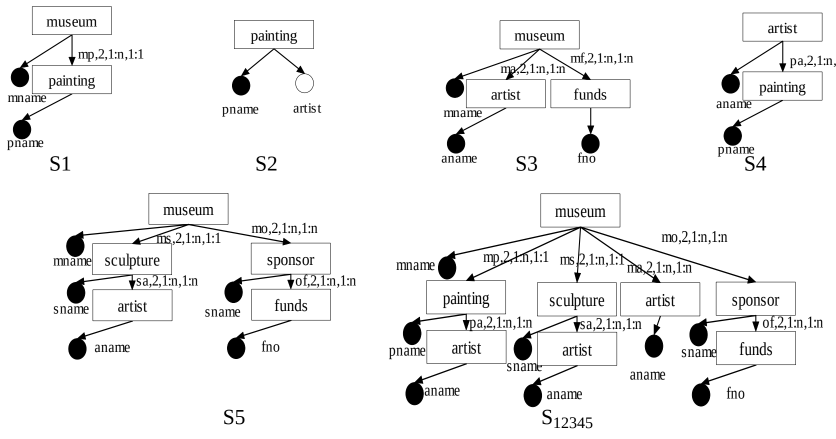

Fig. 2. S_{12345} is the integrated schema of local schemas S1, S2, S3, S4 and S5

Table 1. Mapping table for integrated schema S_{12345} in Fig. 2

Integrated schema	**Local schema**
S12345/museum	S1/museum, S3/museum, S5/museum
S12345/museum/mname	S1/museum/mname, S3/museum/mname, S5/museum/mname
S12345/museum/painting	S1/museum/painting, S2/painting, S4/artist/painting
S12345/museum/painting/pname	S1/museum/painting/pname, S2/painting/pname, S4/artist/painting/pname
S12345/museum/painting/artist	S2/painting/artist, S4/artist
S12345/museum/painting/artist/aname	S2/painting/artist, S4/artist/aname
S12345/museum/sponsor/funds	S3/museum/funds, S5/museum/sponsor/funds
…	…

Consider Fig. 2 where schema S_{12345} is an integration of the local schemas S1, S2, S3, S4, and S5. Table 1 shows a subset of the mapping table generated during the integration process. The first column of the mapping table gives the path from the root to each object class or attribute in the integrated schema; the second column shows the local schema id and the path to the equivalent object classes or attributes in the local schemas. When the mapping is not one-to-one, XQuery functions or user-defined functions are given in the second column

A query in XQuery format has two main parts: the first part contains the selection conditions, and the second part describes how the result is restructured. A *query allocation table* stores the selection condition paths and the return result paths of a query, as well as the local schemas where the data for these paths can be found. Details on the construction of the query allocation table are given in Section 4.1.

4 Query Rewriting

In this section, we will present our approach to rewrite a query on the integrated schema to query the local data sources. Partial information from various local data sources may need to be combined to produce the results of the user query. There are four steps in the proposed algorithm:

Step 1. Build the query allocation table.
Step 2. Group local schemas to form *join groups* that answer the user's query.
Step 3. Decompose user query to subqueries on the local sources.
Step 4. Compose subqueries for local schemas in a join group.

4.1 Build Query Allocation Table

A query allocation table (QAT) consists of a selection condition table and a return result table. The path of each selection condition and the return result is inserted into the selection condition table and the return result table respectively. The associated schemas identified from the mapping table are inserted into the corresponding rows. Two special cases need to be considered which can be treated as two rules.

Case 1: If a path corresponds to a branch in an ORA-SS schema with *n* (*n>1*) relationship types, it must be split into *n* subrows, one for each relationship type. Any attributes of an object class or a relationship type will appear in the row with their object class or relationship type.

Case 2: If a path contains "//" or "/*/" and does not contain any recursive relationship type, then the row that stores the original is retained and rows are created to store the expansion of each path. An expanded path that contains more than one relationship type is handled using Case 1. If "//" or "/*/" involves some recursive relationship type, then "//" or "/*/" will not need to be expanded.

Note that recursive relationship types are represented in the ORA-SS schema diagram by using reference arrow to point to some ancestor in the same path. These cases identify the relationship types involved in the query so that they can be handled properly and the results returned are expected and correct. This also highlights the advantage of using ORA-SS schema diagrams to distinguish between binary and n-ary relationship types and treat them properly in the algorithm. For example, n-ary relationship types should not be split into n-1 binary relationship types in the query allocation table.

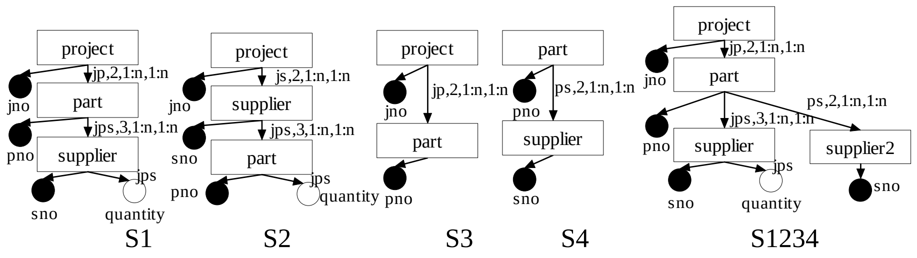

Fig. 3. S_{1234} is the integrated schema of S1, S2, S3 and S4

Example 1: Consider the schemas in Fig 3, where schema S_{1234} is an integrated schema of schemas S1, S2, S3, and S4. We issue query Q1 on the integrated schema to retrieve information about projects and their parts, and which supplier supplies the part to the project. Table 2 shows the query allocation table for query Q1. We note that the relationship type among project, part and supplier is a ternary relationship type. Hence, in the return result table, the path "/project/part/supplier" is not split into two paths. Since the local schema S4 does not model this ternary relationship type, it is not associated with this path. This prevents the retrieval of wrong results by joining the sources in S3 and S4.

```
Query Q1: for $j in /project
          return <project> {$j/jno}
                   {for $p in $j/part
                   return <part>{$p/pno}
                            {for $s in $p/supplier    return {$s}} </part>}
                   </project>
```

Table 2. Query Allocation Table for Query Q1

Selection Condition Table: Empty
Return Result Table:

/project/jno	S1, S2, S3
/project/part/pno	S1, S2, S3
/project/part/supplier	S1, S2,

Example 2: Let us now **c**onsider Fig. 2, and the query Q2 on the integrated schema S_{12345}, which retrieves the names of artists that have works in a museum with name "field". The query allocation table is shown in Table. 3. The aim of QAT is to find the join groups. Since the rewritten queries will need to refer to the user query on the integrated schema, the QAT does not need to contain the details of selection conditions such as "field" in Q2. Note /museum//aname is expanded into two XPath

expressions /museum/painting/artist/aname and /museum/sculpture/artist/aname each of which are further split into two paths because of the binary relationship types. The path "/museum//aname" is retained and rows for each expansion of this path are inserted in the QAT.

Query Q2: for $m in /museum[mname=”field”],$a in distinct-values($m//aname)
return <artist> {$a} </artist>

Table 3. Query Allocation Table for Query Q2

Selection Condition Table :

/museum/mname	S1, S3, S5

Return Result Table:

/museum//aname	S3
/museum/painting	S1
painting/artist/aname	S2, S4
/museum/sculpture	S5
sculpture/artist/aname	S5

4.2 Identify Local Sources to Answer User Query

Next, we need to determine which local schemas must be combined to get the expected results. These groups of local schemas are called *join groups*. The local schemas in each join group must contain all the paths required for the selection condition and must have at least one path for the result.

Algorithm GenerateJoinGroups scans the query allocation table (QAT) to find the join groups. Lines 1-5 create an ordering on the local schemas based on the rows in which they first occur in the QAT and store the ordered list in *lt*. A local schema is low in the ordering if it first occurs in the top row and high in the ordering if it first occurs in the bottom row of the QAT. Lines 6-31 use a stack to find the join groups. The local schemas are considered based on the ordering in the list *lt* from lowest to highest. Initially the lowest local schema is pushed onto the stack, and the next schema to be pushed onto the stack is the next lowest that occurs in a different row. When the schemas on the stack cover all the selection condition paths in the QAT, we output them as a join group. The top schema is popped off the stack, and the algorithm goes on to find the next schema which could contribute to the user query. The algorithm scans the schemas in the order of *lt*, so there is no duplication or missing join groups.

Example 3: Consider the schemas in Fig. 4. The attribute "location" in S_{12345} is a combination of the attributes "address" and "postal code" in S5. The query Q3 retrieves the year and title of the books that were written by "Tom" in the year "2000". The corresponding query allocation table is shown in Table 4.

Algorithm GenerateJoinGroups first looks at the first row "/book/author" in the Selection Condition Table, and adds S1, S2, S3 in the list *lt*. Then it checks the second row "/book/year", and adds S4 in the list *lt*. Thus, the *lt* has local schema order as S1, S2, S3, and S4. After the order is computed, S1 is first pushed on the stack, and S2 is then considered. Since it does not add any extra paths, it is not pushed on the stack. S3 is considered and because it does cover extra paths, it is pushed on the stack. Together S1 and S3 cover all the path information in the QAT, so {S1, S3} is output as a join group. S3 is then popped off the stack, S4 is considered. Together S1 and S4 cover all the path information, and {S1, S4} is output as a join group. {S2, S4} and {S3} are output after that.

Note that {S2, S3} is not a join group, because although they cover all the path information in the selection condition table of the QAT, S2 does not cover any more path information that S3 does not cover and consequently would not add new answers to the result of the query. Note that {S3} is a join group, even though {S1, S3} is also a join group. The result from the rewritten query in {S1, S3} can return the result as Q2, while {S3} can return the partial result which has missing information of the title of book.

The final result is found by taking the union of all the answers from the different join groups. Given that the relationship type information is captured in the ORA-SS model, the union can be based on the relationship type information. For each relationship type, we take the deep union [23], that is, we take the union of the objects if and only if all of their ancestors are the same.

```
Algorithm GenerateJoinGroups
  Input: Query allocation table qat;
  Output: join groups
1. create an empty list lt;
2. for i=1 to num_of_row of qat
3.     for j=1 to num_of_schema_id of row i
4.         if schemaij is present in the rowi and not in list lt
5.             add schemaij to list lt;
6. n=the number of local sources in qat;
7. create an empty stack st;
8. for i=1 to n from lt
9. {
10.    if schemai is not in the top row in qat
11.       break;
12.    push schemai on the stack st;
13.    if schemai is present in all rows of qat
14.    {
15.       Output {schemai};
16.       st=null;
17.       continue;
18.    }
19.   for j=i+1 to n if schemaj occurs in the rows, which the other schemas in st do not occur in, and schemaj does not occur in all the rows that the top element of st occurs in
20.    {
21.       push schemaj on the stack st;
22.       if (the local schemas in st has included all the path information in qat)
23.        {
24.            output all the schemas in the stack st
split by”,” in a “{}”;
25.            pop the top schema off the stack st;
26.        }
27.     }
28.     if (j= =n and st has included all the path in-
formation of the selection condition table and at least
one result in return result table)
29.        output all the schemas in the stack st split
by”,” in a “{}”;
30.     st=null;
31.}
```

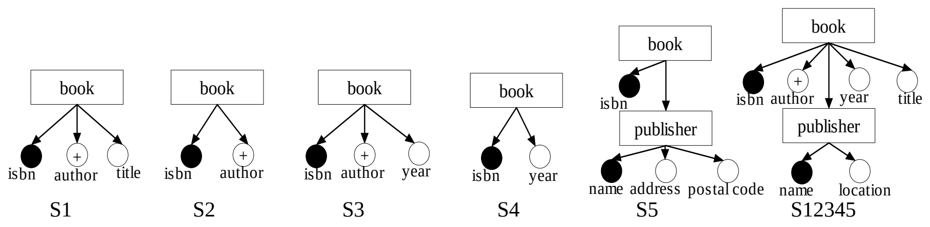

Fig. 4. S_{12345} is the integrated schema of local schemas S1, S2, S3, S4, S5

Query Q3: for $b in /book where $b/author=”Tom” and $b/year=”2000” return <result> {$b/year/text()} {$b/title/text()} </result>

Table 4. Query Allocation Table for Query Q3

Selection Condition Table:

/book/author	S1, S2, S3
/book/year	S3, S4

Return Result Table:

/book/year	S3, S4
/book/title	S1

4.3 Decompose User Query to Subqueries on Local Sources

This step decomposes the user query into queries on the local schema based on the join groups. Subqueries are composed to compute the answers in the same join group in Step 4. Hence, in addition to retrieving the data required by the user query, we also need the data necessary to join the parts of the answers from different local schemas together. We call the classes that are necessary for joining the parts of answers as *join object classes*. The key of the join object class is used for testing the equivalence when joining the subqueries.

A join object class depends on the semantics of the schema. We have 3 cases:

Case 1: For a join group, if there are *n* paths in the QAT from different local schemas with a common ancestor in the user query, then the least common ancestor in the user query is a join object class. An object class O is the least common ancestor of paths P1 and P2, if O is an object class that occurs in both P1 and P2, and O does not have any descendant object class that also occurs in P1 and P2.

Case 2: For a join group, if the paths in the QAT are from different local schemas, and there is an object class that is the end of one path and the start of the other path, then this intermediate object class is a join object class.

Case 3: For a join group, if two attributes of the same relationship type in a user query are from different local schemas, then all the object classes involved in this relationship type are join object classes.

Example 4: Recall Example 3 and the join group {S1, S3}. S1 provides "/book/title", "/book/author" and S3 provides "/book/year", "/book/author". To answer the query Q3, the subqueries from S1 and S3 need to be composed using the key of their least common ancestor i.e. the key "isbn" of the join object class "book".

We first consider the case where the local schemas are projections of the integrated schema. The rewritten query for a local schema will effectively be a projection of the user query with the join object class identifier included in the *return* part of the rewritten query. The rewritten query can be derived as follows:

1. For every path in the *let* part, *for* part, *where* part and *return* part of the user query, retain the path if it exists in the local schema.
2. Add the path to any join object class identifiers that are relevant to this local schema in the join group being considered.

When the local schemas are not projections of the integrated schema, the query will need to be rewritten based on the local schema structure. We will first describe how to rewrite a user query for a local schema where the subquery on the local schema returns only one object class or attribute. Then we discuss how to rewrite a user query for a local schema where the subquery on the local schema returns many object classes or attributes.

4.3.1 Subquery returns only one object class or attribute.

We have two cases. The first case is for queries involving one object class or attribute, while the second case is for queries involving more than one object class.

Case A1. Queries involve one object class or attribute

An object class in an integrated schema can originate from either an object class or an attribute in a local schema, or it can be derived from object classes and attributes in one local schema.

Case A1-i*. Integrated object class originates from a source object class.*

When an integrated object class is mapped to an equivalent object class from a local schema, but the path from the root to the equivalent object class is different, variable bindings in the *for* clause or *let* clause are changed according to the mapping table that specifies the path of the equivalent source object class.

Example 5: Consider the schemas in Fig. 2. Query Q4 on the integrated schema S_{12345} retrieves all the information on the object class "funds", which is in path "/museum/sponsor/funds":

```
Query Q4: for $f  in /museum/sponsor/funds
              return  <result> {$f} </result>
```

Based on the mapping table, we have S_{12345}/museum/sponsor/funds: S3/museum/funds, S5/museum/sponsor/funds. This indicates that the query can be rewritten to query local sources S3 and S5. The rewritten query on source S5 will be the same as Q4, while the query on S3 will be as follows:

```
Query Q4_S3: for $f in /museum/funds
                 return  <result> {$f} </result>
```

***Case A1-ii**. Integrated object class originates from an attribute.*

An object class can also originate from an attribute, because a concept can be expressed as an attribute in one schema, and as an object class in another schema. When rewriting such queries, variable bindings in the *for* clause or *let* clause are changed according to the mapping table that specifies the path of the equivalent attribute; the equivalent object class is created in the *return* clause with the attribute as an attribute of this object class.

Example 6: The following query is on the integrated schema S_{12345} of Fig. 2. Query Q 5 retrieves the information of artists of the painting with pname "hero".

```
Query Q5:  for $p in /museum/painting
               where $p/pname=”hero”
               return  <result> {$p/artist} </result>
```

Query Q5 will be rewritten for S2 and S4. Since schema S2 (see Fig. 2) models "artist" as an attribute of the object class "painting", Query Q5_S2 will compute the information for artist on local schema S2:

```
Query Q5_S2:  for $p in /painting
                  where $p/pname=”hero”
                  return <result> <artist> <aname> {$p/artist/text()} </aname>
                  </artist>  </result>
```

***Case A1-iii**. Integrated object class or attribute originates from a set of object classes (attributes) or vice versa.*

When one object class (attribute) in the integrated schema is the combination of many object classes (attributes) of another local schema or vice versa, XQuery or user-defined functions can be used to substitute the path in the user query.

Example 7: Consider the schemas in Fig. 4. Query Q6 retrieves the publisher location of the book with isbn "7-5053-4849-3/TP.2370" on the integrated schema S_{12345}:

```
Query Q6: for $b in /book
              where  $b/isbn=”7-5053-4849-3/TP.2370”
              return <result>{$b/publisher/location}</result>
```

Q6 will be rewritten on S5. The mapping in the mapping table shows that S12345/book/publisher/location:string-join((S5/book/publisher/address/text(), S5/book/publisher/postalcode/text()),“ ”). We assume that the attribute "location" is expressed by the address followed by a space and the postal code. The query on S5 is shown in Query Q6_S5. It combines the address and postal code by the XQuery functions from the mapping table. The rewritten query on S5 will be:

```
Query Q6_S5: for $b in /book
                  where  $b/isbn=”7-5053-4849-3/TP.2370”
                  return <result> <location> {string-join(($b/publiser/address/text(),
                        $b/publisher/postalcode/text()),” ”)}</location> </result>
```

Case A2. Query involves more than one object classes.

When the number of object classes in the query path is more than one, we need to consider the structural relationship type between the object classes. There are two cases: (1) object classes are swapped in the integrated schema, and (2) siblings in a local schema are mapped to ancestor and descendent in the integrated schema.

Case A2-i.When object classes in the integrated schema are swapped in the hierarchy compared to the local schema, the path in the subquery needs to be rewritten based on the path of the local schemas.

Example 8: The following query on the integrated schema S_{12345} in Fig. 2 retrieves all the “museum” which have the paintings by artist “David”.

```
Query Q7: for $m in /museum where $m/painting/artist/aname=”David”
              return<museum>{$m/mname/text()}</museum>
```

The join groups are {S1, S2} and {S1, S4}. In join group {S1, S4}, the join object class is painting for S4. The projection subquery on S4 is:

```
Query Q7_S4’: for $p in /painting where $p/artist/aname=”David”
                  return<painting>{$p/pname}</painting>
```

The path expression in the *where* clauses are changed to the corresponding object class (attributes) by using /../. The rewritten query on S4 is:

```
Query Q7_S4:  for $p in/artist/painting where $p/../aname=”David”
                  return <painting>{$p/pname}</painting>
```

This query needs to be joined with the subquery for S1 to get the final result.

Case A2-ii. When two object classes have an ancestor-descendant relationship type in the integrated schema, but they are siblings in the local schema, then the least common ancestor of these object classes must be used as binding variables to connect them. The related path in the where and return clause must be revised based on the structure of the local schemas.

Example 9: In Fig. 5, students work for projects, and students have their labs. The lab also has coordinators. Consider the query Q8 on the integrated schema S123, which retrieves a project lab coordinator where pno is “p01”.

```
Query Q8: for $p in /project where $p/@pno=”p01”
              return <result>{$p/student/lab/coordinator}</result>
```

The join groups are {S1, S3} and {S2, S3}. The return clause in Q8 shows that the query path is from $p to lab. In order to rewrite the query for schema S1, the algorithm looks for the nearest ancestor node that is common to both project and lab. Student is then bound to the variable in the *for* clause as follows:

Query Q8_S1: for $s in /student where $s/project/@pno="p01"
return <result>{$s/lab/@lno}</result>

This query needs to join with the subquery for S3 to get the final result.

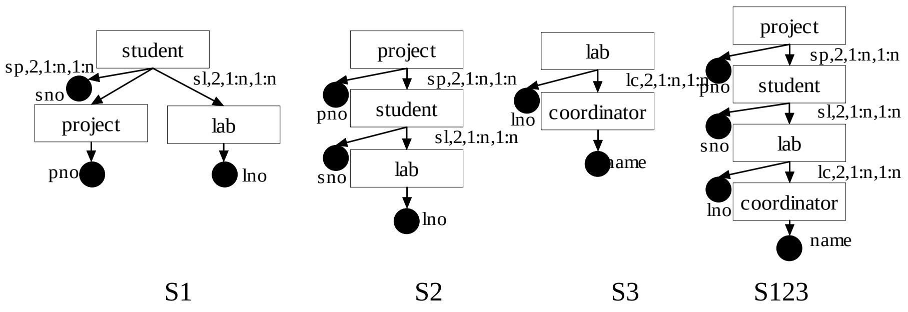

Fig. 5. S_{123} is the integrated schema of local schemas S1, S2, S3

4.3.2 Subquery returns many object classes or attributes. Chen et al. in [3] introduce an algorithm for the automatic generation of XQuery view definitions for ORA-SS views, focusing on the view definitions for hierarchical structures of XML. Due to space limitations we do not cover this case in this paper except to note that their algorithm can be used to rewrite such queries.

4.4 Compose Subqueries for Join Group

When joining subqueries on local schemas in the same join group, the identifier of the join object classes must be tested for equivalence.

We start by considering the basic case where the same object attributes are from different local schemas. To compose subqueries from these local schemas in join groups, the clauses *for*, *where*, and *return* are combined together with the join condition equivalence test inserted in the *where* clause.

We allow the return results to have missing information. The parent object will not be removed from the return result if it has a missing child. For each return object or attribute, the join equivalence condition test related to this return object or attribute is nested in the appropriate part of the query.

Example 10: Consider the schemas in Fig. 4. and query Q9 that retrieves year and title of the books that were written by "Tom" in year "2000" and retrieves the publisher name if the book's publisher location is Singapore.

```
Query Q9: for $b in /book   where $b/author="Tom" and $b/year="2000"
      return<result>{$b/year/text()} {$b/title/text()}{
                  for $p in $b/publisher
                  where contains ($b/publisher/location/text(),"Singapore")
                  return<publisher> {$b/publisher/name} </publisher> }
        </result>
```

The join groups are {S1, S3, S5}, {S1, S4, S5}, {S2, S4, S5} and {S3, S5}. We show the query example for the join group {S1, S3, S5}. The user query is decomposed into subqueries on the local schemas S1, S3, and S5. The join object class is “book” for these local schemas. The subqueries on S1, S3 and S5 are shown below:

```
Query Q9_S1:  for $b in /book
                 where $b/author=”Tom”
                return <result> {$b/isbn/text()} {$b/title/text()} </result>
Query Q9_S3: for $b in /book
                where $b/author=”Tom” and $b/year=”2000”
                return <result> {$b/isbn/text()} {$b/year/text()} </result>
Query Q9_S5: for $b in /book
               where contains ($b/publisher/address/text(),”Singapore”)
               return<result>{$b/isbn/text()}
              <publisher>{$b/publisher/name} </publisher></result>
```

The composition of the subqueries for local schemas S1, S3 and S5 are as follows:

```
for $b1 in doc(“S1.xml”)/book, $b3 in doc(“S3.xml”)/book
where $b1/author=”Tom” and $b3/author=”Tom” and $b3/year=”2000”
and $b1/isbn=$b3/isbn
return <result>{$b3/year/text()} {$b1/title/text()}
                   {for $b5 in doc(“S5.xml”)/book
                   where contains ($b5/publisher/address/text(),”Singapore”) and
                   $b5/isbn=$b1/isbn
                     return<publisher> {$b5/publisher/name}<publisher>}</result>
```

Note that although the join object class for S1, S3 and S5 is book, the equivalence tests are on separate lines in the rewritten query. This is because we allow parent information to be returned even when a child object class is missing.

5 Comparison with Related Work

Amman et al. in [1] propose a mediator architecture for querying and integrating XML data sources. Their global schema is described as an ontology, which is expressed in a light weight conceptual model. Similar to our algorithm, their method also finds join groups, where the local sources of the join groups can together compute the results for the user query. However, the limitation in [1] is that a query cannot return nested structures.

Lakshmanan and Sadri in [8] propose an infrastructure for interoperability among XML data sources. Mapping rules are created to map the items in local schemas to a common vocabulary. They also address the query processing and optimization in the system. For query processing, they differentiate between inter-source query and intra-source query, which query across local schemas and within one local schema respectively. Consistency conditions are used to optimize inter-source queries. One limitation of this work is that when results from local schemas are joined, the join variable is limited to the lowest common ancestor of nodes.

Yu and Popa in [22] introduce an algorithm for answering queries via a target schema. The algorithm uses target constraints that are used to express data merging rules. The mappings from the integrated schema and local schemas are tree to tree.

Generating such mappings is expensive, especially when the XML sources are complicated.

The models that are utilized in the works [1, 8, 22] cannot specify whether a relationship type is binary or n-ary and do not distinguish between attributes of object classes and attributes of relationship types from the local XML sources. The lack of such semantic information may lead to the retrieval of wrong results as the following example illustrates.

Example 11: Recall Example 1 where only S1 and S2 will be considered for the query Q1. Since the works in [1, 8, 22] cannot distinguish between binary or n-ary relationship types, they will join the sources from S3 and S4 to get the result, which is not correct for the user query. The example below highlights the problem for the attributes and n-ary realtionship. For simplicity, schemas S3 and S4 are omitted here. Let the data source for S1 be X1, and the data source for S2 be X2 as shown in Table 5. Table 6 shows the results for query Q1 that are retrieved by our algorithm and the methods in [1, 8, 22].

We observe that the results returned by the query rewriting method in [1, 8, 22] contain the project with jno "j01" has part "p01", which is supplied by suppliers with sno "s01" and "s02". This violates the local data sources X1 and X2, where the project with jno "j01" has part "p01" is only supplied by suppliers with sno "s01". This is because the methods in [1, 8, 22] treat the relationship type between part and supplier as a binary relationship type, instead of the intended ternary relationship type involving project, part, and supplier. They treat the quantity as the attribute of part in S2, so when they find the part with pno "p01" has quantity "100" in X1, and has quantity "200" in X2, they will combine them to make the final result. This leads to the wrong answer returned. In contrast, our algorithm takes the XML hierarchy structure into consideration and retrieves the correct answers.

To summarize, our algorithm differs from existing works in the following ways:

1. We treat binary and n-ary relationship types differently. Treating an n-ary relationship type as n-1 binary relationship types gives wrong results.
2. We treat attributes of object classes and attributes of relationship types differently in the QAT and when we compose the sub queries of the local sources.
3. Our algorithm takes the XML hierarchy structure into consideration when doing the rewriting.

Table 5. Data sources for S1 and S2 in Fig. 3

X1:	X2:
<project jno=”j01”>	<project jno=”j02”>
<part pno=”p01”>	<supplier sno=”s02”>
<supplier sno=”s01”>	<part pno=”p01”>
<quantity> 100 </quantity>	<quantity> 200 </quantity>
</supplier>	</part>
</part>	</supplier>
</project>	</project>

Table 6. Results retrieved by our algorithm and [1, 8, 22]

Results obtained by our proposed algorithm	**Result obtained by [1, 8, 22]**
<result>	<result>
<project jno=”j01”>	<project jno=”j01”>
<part pno=”p01”>	<part pno=”p01”>
<supplier sno=”s01”>	<supplier sno=”s01”>
<quantity> 100 </quantity>	<quantity> 100 </quantity>
</supplier>	</supplier>
</part>	<supplier sno=”s02”>
</project>	<quantity> 200 </quantity>
<project jno=”j02”>	</supplier>
<part pno=”p01”>	</part>
<supplier sno=”s02”>	</project>
<quantity> 200 </quantity>	<project jno=”j02”>
</supplier>	<part pno=”p01”>
</part>	<supplier sno=”s01”>
</project>	<quantity> 100 </quantity>
</result>	</supplier>
	<supplier sno=”s02”>
	<quantity> 200 </quantity>
	</supplier>
	</part>
	</project>
	</result>

6 Conclusion and Future Work

In this paper, we have introduced a semantic approach to rewrite queries for semistructured data integration. A user’s queries on the integrated schema are rewritten to query the local sources. When XML repositories are integrated there may be semantics that are not expressed explicitly, and without the necessary semantics it is possible to misinterpret the meaning of the data and combine the results from different local schemas to give unexpected results. Our algorithm uses the ORA-SS model to describe the schemas of the local data sources and the integrated schemas. This allows us to distinguish between binary and n-ary relationship types, attributes of object classes and attributes of relationship types, and in turn treat these cases differently in our rewriting algorithm.

References

1. B. Amann, C. Beeri, I. Fundulaki, M. Scholl. Querying XML sources using an Ontology-based Mediator. CoopIS, 2002.
2. P. Buneman, S. Davidson, W. Fan, C. Hara, W.C. Tan. Keys for XML. WWW Conference, 2001.
3. Y. Chen, T.W. Ling, M.L. Lee. Automatic Generation of XQuery View Definitions from ORA-SS Views. ER, 2003.
4. S. Cluet, P. Veltri, D. Vodislav. Views in a Large Scale XML Repository. VLDB, 2001.
5. O.M. Duschka, M.R. Genesereth. Answering Recursive Queries Using Views. ACM PODS, 1997.
6. L. Haas, D. Kossmann, E. Wimmers, J. Yang. Optimizing queries across diverse data sources. VLDB, 1997.
7. A. Halevy. Theory of Answering Queries Using Views. ACM SIGMOD Record 29(4), 2000.
8. L.V.S. Lakshmanan, F. Sadri. Interoperability on XML Data. ICSW, 2003.
9. M.L. Lee, T.W. Ling, W.L. Low. Designing Functional Dependencies for XML. EDBT, 2002.
10. A. Levy. Logic-Based Techniques in Data Integration. Logic based artificial intelligence, 1999.
11. T.W. Ling, M.L. Lee, G. Dobbie. Semistructured Database Design, ISBN: 0-387-23567-1, Springer, 2005.
12. I. Manolescu, D. Florescu, D. Kossman. Answering XML queries over heterogeneous data sources. VLDB, 2001.
13. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, et al. The TSIMMIS project: Integration of heterogeneous information sources. Journal of Intelligent Information Systems, 1997.
14. K. Passi, E. Chaudhry. A Global-to-Local Rewriting Querying Mechanism using Semantic Mapping for XML Schema Integration. ODBASE 2003.
15. K. Passi, L. Lane, S.Madria, Bipin C. Sakamuri, M. Mohania, S. Bhowmick. A Model for XML Schema Integration. EC-Web, 2002.
16. R. Pottinger, A. Levy. A Scalable Algorithm for Answering Queries Using Views. VLDB, 2000.
17. M. Stonebraker. Implementation of integrity constraints and views by query modification. ACM SIGMOD, 1975.
18. Xyleme. A dynamic warehouse for XML Data of the Web. IEEE Data Engineering Bulletin, 2001.
19. H.Z. Yang, P.A. Larson. Query Transformation for PSJ-queries. VLDB, 1987.
20. X. Yang. Global Schema Generation and Query Rewriting In XML Integration. MSc thesis, National University of Singapore, 2005.
21. X. Yang, M.L. Lee, T.W. Ling. Resolving Structural Conflicts in the Integration of XML Schemas: A Semantic Approach. ER 2003.
22. C. Yu, L. Popa. Constraint-Based XML Query Rewriting for Data Integration. SIGMOD 2004.
23. P. Buneman, A Deutsch, W.C. Tan. A deterministic model for semistructured data. Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, 1998.