Query Transformation of SQL into XQuery within Federated Environments

Heiko Jahnkuhn, Ilvio Bruder, Ammar Balouch, Manja Nelius, and Andreas Heuer

Department of Computer Science, University of Rostock,
18051 Rostock, Germany,
{hj016,ilr,ab006,manel,ah}@informatik.uni-rostock.de

Abstract. Federated information systems integrate various heterogeneous autonomic databases and information systems. Queries respect to the federation have to be translated into the local query language and must be transformed with respect to the local data model. This paper deals with the problem of a global query according to an object-relational federation service. This SQL query is to be translated into an equivalent XQuery expression, so that it can be processed by the corresponding XML component database system according to the local schema.

1 Introduction

Federated database and information systems (FDBIS) represent system architectures for multidatabase systems [1]. In general, one characteristic of FDBIS is that component database systems are not restricted concerning the underlying data model or query language. As a result, the participants of the federation may be very heterogeneous. A general architecture of a federation is shown in Fig. 1.

The approach introduced in this paper deals with the heterogeneity between the object-relational and the XML-based data model, for which the query languages SQL and XQuery are applied respectively. However, this paper does not focus on transformation from XQuery into SQL, because there are several approaches and implementation techniques, for example considered in [4, 10]. It mainly focuses on the transformation of SQL into XQuery, for which two essential scenarios are possible.

In the first case, there is an XML-based federation service, which uses XQuery as query language and an object-relational component database system, which is accessed by a local application via SQL. If the local application queries the federation with respect to the local schema, this statement has to be transformed into XQuery and must correspond to the global schema. This scenario is handled e.g. in [6].

In the second case, to which the introduced algorithm refers, the federation service is based upon the object-relational data model, which is queried by SQL. These SQL queries according to the global schema have to be transformed for

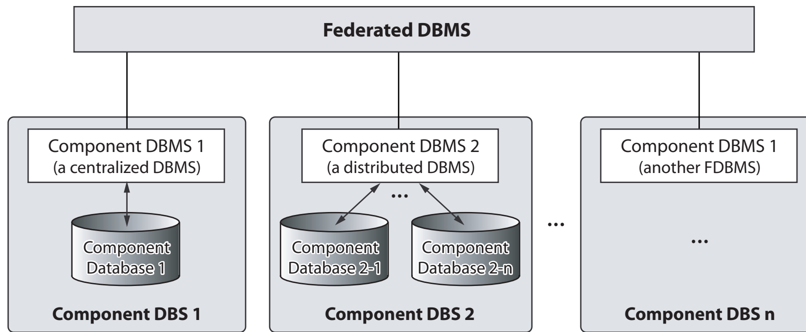

Fig. 1. General architecture of a federation (according to [3])

every participant of the federation. In case of an XML database, every query must be translated into an XQuery request respect to a single document or a collection of XML documents.

Using a rule-based system for such a transformation, as described in [5] for other languages, can cause an unreasonably high administration costs. An XML database system (XMLDBS) usually does not manage only one document collection which is valid relating to a schema, but uses a multitude of variably structured documents and diverse schemata. On the contrary, rule systems are static and cause problems within dynamic applications. However, as data, and hence the underlying schema descriptions, of XMLDBS in most cases are changed frequently, every modification would cause an adaptation of the rule system.

To counter these effects, this paper introduces an algorithm, which is able to automatically perform such a transformation relating to diverse schema descriptions. Former approaches need a valid XML and/or relational representation of the data. E.g., [6] needs a predefined schema for the transformation. This approach uses only a set of paths as mapping description of a federation system. A schema for the query transformation can be automatically and arbitrarily generated (it depends possibly on the federated system). Changing the schema causes only the adaptation of the mapping between the local and the global schema; an adaptation of the algorithm is not required. However, this paper only presents the fundamental algorithm, which is restricted to the basic SQL statements. This means, only elementary logical operators (e.g. `<`, `>`, `=`), which can be linked with `AND`, `OR`, and `NOT`, are allowed for the clauses `WHERE` and `HAVING`. Furthermore, aggregate functions are possible within the `HAVING` clause. [7] goes into details concerning the realisation of other SQL constructs (e.g., `between`, `exists`, and `contains`) and describes further phases and problems occurring at query processing in federated information systems.

The purpose of this algorithm is to generate a tuple (schema description, XQuery statement) based on the mapping of the local schema onto the global schema and of the SQL query with respect to the global schema. The XQuery

expression within the tuple represents the local equivalent of the global SQL request and is queried respect to the document collection whose documents are valid according to the given schema. Then, the federation system sends this tuple to the appropriate XML CDBS, which processes the request.

2 Query Processing

Starting point of the query processing is a SQL request in turns of the federation service's global schema. Assuming a semantics conserving mapping between the local schema description and the global federation schema, a translation table is created prior to the actual transformation. This table contains a list of all global attribute names as well as the local schema's corresponding path expressions, which are assigned to the respective global attributes. The global SQL request in conjunction with this translation table is the basis of the query processing, which can be subdivided into four separate phases described below and demonstrated with an example within the next section.

Phase 1

The first step of the algorithm analyses and classifies the query which must be transformed. For this purpose all occurrences of global attribute names within the query are determined and substituted with their corresponding path expression by applying the translation table. As a result of this substitution a list of paths is generated, which can be interpreted as a tree structure. This tree, in the following called query tree, represents a sub tree of that one that is described by the underlying federated schema, and is restricted to that information, which is relevant to the query.

The nodes of the created query tree are now tagged with three different flags, which classify the query tree. These flags are called *return elements*, *where elements* and *splitting nodes*.

The *return elements* result from those path expressions, which occur within the `SELECT`, `GROUP BY`, `HAVING`, and `ORDER BY` clauses after the substitution of all global attribute names.

The `WHERE` clause of the query has to be processed to determine the *where elements*. Because of a missing schema every element of the query tree has to be considered as a potential list of path expressions. Therefore, by using the where elements it is specified which part of the `WHERE` clause references which element of the query tree. By regarding the selection condition as a conjunction of selection conditions, every conjunction term refers to the bottom element in the query tree, which contains all occurring path expressions in the descendant-or-self axis. Therefore, the `WHERE` clause is decomposed into a list of conjunctions via the distributive law. Consequently, this list consists of either disjunctions or atomic comparisons and would fit the primal `WHERE` clause if joined with `AND`. All elements of the list include at least one path expression after the substitution of the global attributes. Then, the most common path (mcp) is determined for

every element within the list. This one is the last element of the query tree, which includes all elements of the current list element within the descendant-or-self axis. As a result every mcp now represents a where element in the query tree.

Finally, the *splitting nodes* have to be determined. Splitting nodes describe nodes, which lead to a new FLWR expression within the result query for each subtree of this node. Therefore, for each element of the query tree is checked, whether it matches one of the patterns shown in Fig. 2. In case it does, the element is marked as a splitting node. Pattern a) represents the context of the query, that is that tree element, which contains all XML elements concerning the request. Patterns b) to e) analyse, whether the current element includes where elements as well as return elements within the descendant-or-self axis.

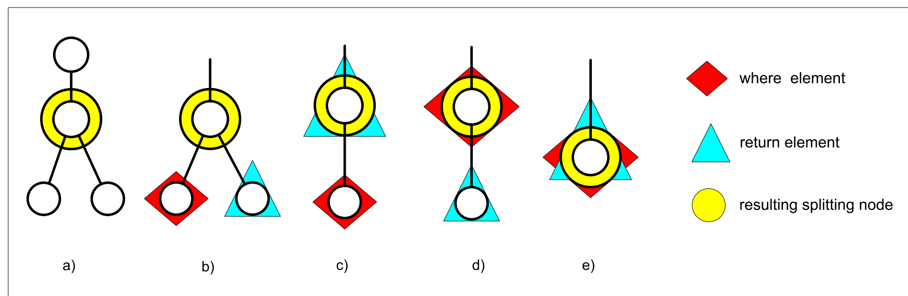

Fig. 2. Patterns of splitting nodes

Consequently, the result of the first transformation step is a tagged query tree, which is shown exemplarily in Fig. 3. This tree represents the input for the next phase of the transformation.

Phase 2

The second transformation step is a mapping algorithm which generates a For-Let-Where-Return (FLWR) expression for every splitting node within the query tree. The result is a nested XQuery statement representing the tagged query tree. The mapping uses a top-down method by determining the first splitting node beginning from the root element. This is the context node the SQL-query refers to. The outermost FLWR expression is now composed of a `for` clause, which declares the splitting node as context node, a `where` clause, which corresponds to a conjunction of the selection conditions of the where elements, and a `return` clause. For generating the `return` clause the descendant axis of each node is checked, whether it contains return elements only; in this case, the return elements are output out directly as path expressions. If the descendant axis

includes a splitting node, it will be used as context node for a new FLWR statement. Then, the mapping algorithm is repeated, whereby the context node is considered relatively to the superior splitting node, and the `where` clause represents a conjunction of all where elements within the descendant-or-self axis of the current splitting node. Fig. 3 shows an example of this mapping algorithm.

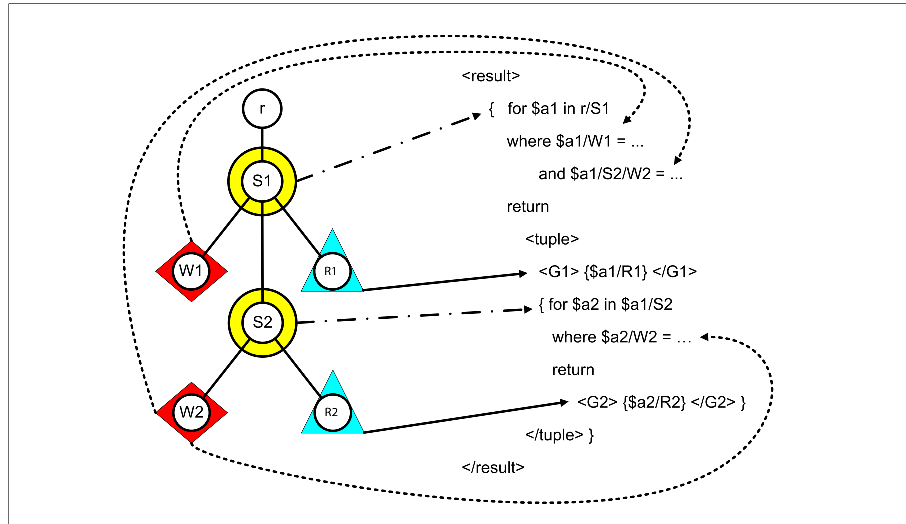

Fig. 3. Example of a mapping

The XQuery statement generated during this second transformation step already represents a relational SET-OF-TUPLE-OF structure. Consequently, the result of the XQuery expression is a set of tuple elements, which are valid with respect to the `SELECT` clause and contain the specified return elements to emulate the relational SET-OF-TUPLE-OF representation. This structure is the basis of the third transformation phase.

Phase 3

The third transformation step realises the optionally specified `GROUP BY` and `HAVING` clauses. As the second phase already generated a relational structure, the handling of these clauses is not a serious problem anymore. Thereby, this phase is subdivided into two stages, because a `HAVING` clause is optional. The first partial stage realises the grouping by generating a new FLWR statement, which uses the output of the second transformation step as input. Here, every grouping attribute is referenced within the `for` clause, so that an n-dimensional vector space is spanned. Afterwards, every tuple element is assigned to a point

inside this vector space via the `let` clause. Every non-empty point, that means a combination of values at least one tuple element is assigned to, represents a grouping. Thereupon, by the `return` clause a result set is created which consists of a set of group elements. A group element includes the grouping attributes as well as the tuple elements, which have identic values regarding the grouping attributes.

Based on this generated structure the selection concerning the `HAVING` clause can be realised. Again a FLWR expression is created, which uses the previous one as input. Thereby, every group element is passed through sequentially by using the group element inside the `for` clause as context node. The `where` clause results directly from the `HAVING` clause; only the global attribute names must be substituted with the corresponding paths and be considered relatively to the context node. The result set generated within the `return` clause matches that one of the first stage. The procedure of this third transformation step is explained graphically in Fig. 4.

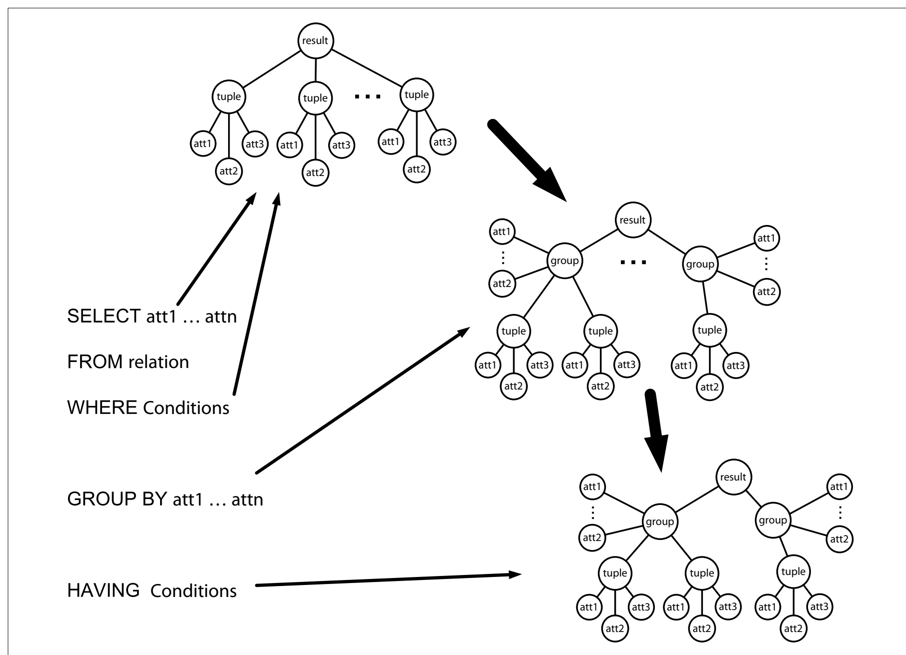

Fig. 4. Realisation of `GROUP BY` and `HAVING` clauses

Phase 4

The fourth transformation phase realises the possibly specified `ORDER BY` clause as well as the `SELECT` clause, which may include aggregate functions and renamings. For this a new FLWR expression is generated again. Three different circumstances have to be considered to determine the context node:

1. The global `SELECT` clause contains aggregate functions only and no attributes. If a grouping exists, the aggregate functions refer to the result's group element, otherwise they refer to the result element, thus the whole result set. In both cases, the aggregate functions refer to the parent node of the tuple element.
2. The global `SELECT` clause contains attributes only, no aggregate functions. If a grouping exists, the attributes have to be declared inside the `GROUP BY` clause and consequently refer to the group element. In case of no `GROUP BY` clause the attributes refer to every single tuple element. In both cases, the attributes refer to all direct children of the result element.
3. The global `SELECT` clause contains attributes as well as aggregate functions. Hence a `GROUP BY` clause was specified with respect to those attributes that occur within the `SELECT` clause as direct attributes. This also means that these attributes are identical within all tuple elements of the grouping. Consequently, this case is to be handled analogically to the second one.

The `ORDER BY` clause of the FLWR statement can be determined analogically to the `HAVING` clause, which was described in the third transformation step. Now the `return` clause contains aggregate functions, which are referenced directly via `Agg(/tuple/attribute)`, and attributes, which can be output immediately via `/attribute`. Thereby, every output is enclosed in tags which are named either after the attribute name, after the specified alias, or after the concatenation of the aggregate function name and the attribute name. The so defined output is enclosed in `<tuple>` tags again. In this way, a SET-OF-TUPLE-OF representation of the requested output is created.

The output generated during this phase represents the final result of the query transformation. Now, the tuple (schema name, XQuery statement) can be transmitted to the respective XML CDBS for evaluation. It is also possible to optimise or minimise the XQuery expression first. There are several proposals, [9] describes one of them.

3 Transformation Example

The described algorithm is demonstrated by an example in the following section. Therefore, the following query example is used:

```
SELECT   Name, Address->City AS City
FROM     Hotels
```

```
WHERE    Name LIKE "Beach%" AND
         (Address->ZIP < 20000 OR
          Address->City = "Freiburg")
GROUP BY Name, Address->City
```

During the first step of the transformation a list of all occurring path expressions is determined. Based on the underlying schema description the list may look like this:

```
hotels/hotel/name
hotels/hotel/address
hotels/hotel/address/zip
hotels/hotel/address/city
```

With this list of path expressions a sub tree of the underlying schema is described, which is marked with `where` elements, `return` elements and splitting nodes. In the example the `return` elements only result from the substituted path expressions within the `SELECT` and `GROUP BY` clause. On the one hand, the list of `where` elements contains the atomic comparison

```
W1 = Name LIKE "Beach%",
```

on the other hand, it includes the disjunction

```
W2 = Address->ZIP < 20000 or
     Address->City = "Freiburg".
```

After substituting all global attributes the most common paths of the where elements are:

```
mcp(W1) = hotels/hotel/name,
mcp(W2) = hotels/hotel/address.
```

By applying the patterns in Fig. 2 three splitting nodes can be derived from this result. This transformation step as well as the marked query tree and the resulting splitting nodes are shown in Fig. 5.
The second step of the transformation uses the marked query tree in Fig. 5 as input for the mapping algorithm. Starting from the root element the first splitting node (`hotels/hotel`) is determined as context node, whereupon the first FLWR expression is generated accordingly. As both child elements also contain splitting nodes, two FLWR expressions are generated for them. These new expressions are positioned relatively to the first splitting node. Within these splitting nodes, the return elements can be directly referenced as a result. This transformation phase is illustrated in Fig. 6
As a result of the generated XQuery expression a structure is created, which is valid respect to the following generic DTD:

```
<!ELEMENT result (tuple*)>
<!ELEMENT tuple  (Name, City)>
<!ELEMENT Name   ANY>
<!ELEMENT City   ANY>
```

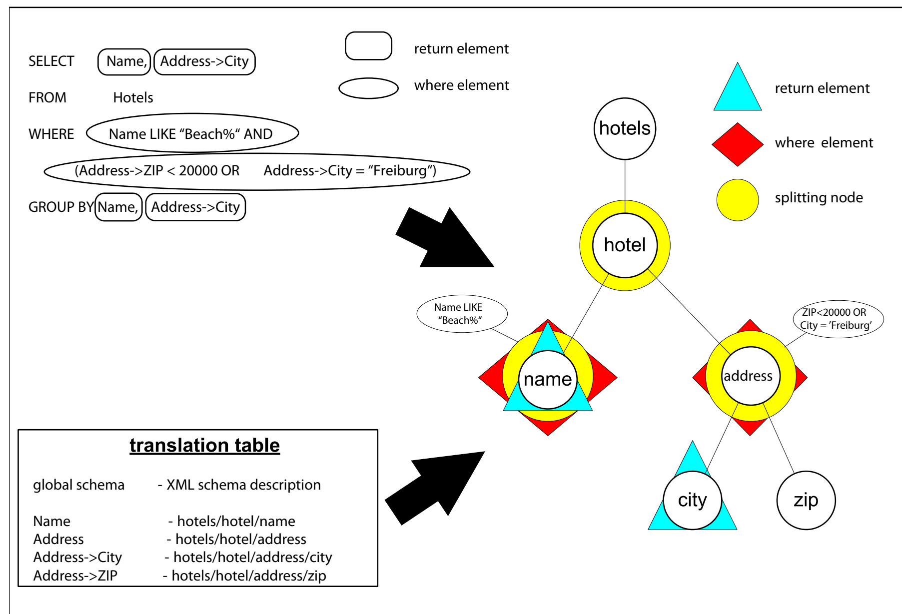

Fig. 5. Process of the first transformation step

From this DTD it is recognisable, that the step already created a relational SET-OF-TUPLE-OF representation of the primary hierarchic structure. This structure is used again as an input for the third transformation step, which implements the grouping. Therefore, a new FLWR expression, consisting of a `let` and `return` clause, is generated. Within the `let` clause a variable is bound to the output of the second transformation step, which is rearranged within the `return` clause. This is done by generating a new FLWR statement, which references the grouping attributes inside the `for` clause as well as `name` and `city`. As a result, every possible combination of both attributes is considered. After that, those tuple elements, which contain the same combination of values, are assigned to every combination inside the `let` clause. Finally, within the `return` clause is ensured that no empty combinations are accepted for the result and a group element is created. The latter consists of the combination of all values as well as of those tuple elements which contain the respective combination. This grouped structure is used as an input for the fourth step of the transformation.

Based on the case differentiation concerning the `SELECT` clause, the second condition is fulfilled (the `SELECT` clause contains only attributes, but no aggregate functions). Hence, the selection is applied to the child elements of the result element, which is the tuple element in the example. The just generated FLWR expression references the tuple element as context node, and the selection attributes can be output inside the `return` clause directly. The XQuery state-

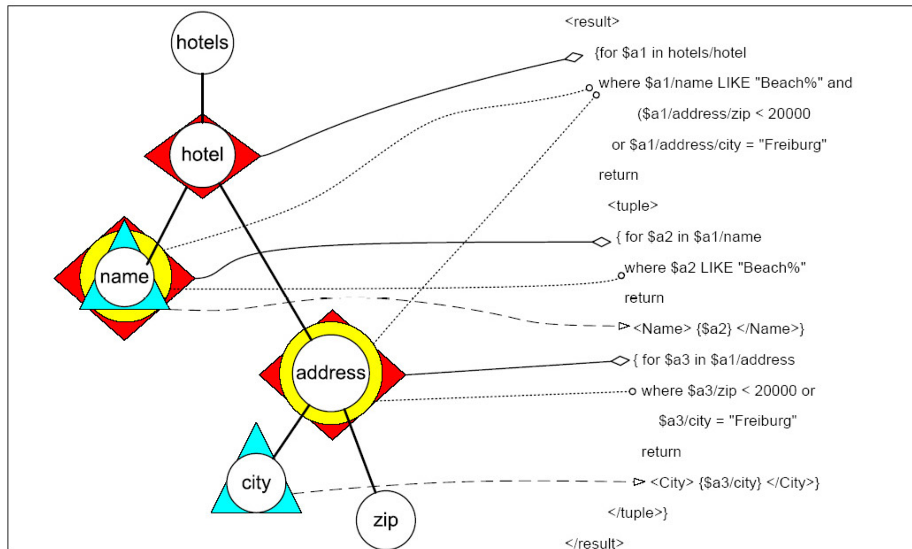

Fig. 6. Process of the second transformation step

ment created during this transformation step represents the overall result of the transformation algorithm[1]:

```
for $final in (
  let $result :=(
    <result>
      { for $a1 in hotels/hotel
        where $a1/name LIKE "Beach%" and
              ($a1/address/zip < 20000
               or $a1/address/city = "Freiburg")
        return
          <tuple>
            { for $a2 in $a1/name
              where $a2 LIKE "Beach%"
              return
                <Name> {$a2 } </Name>,
                for $a3 in $a1/address
                  where $a3/zip < 20000 or
                        $a3/city = "Freiburg"
                  return
                    <City> {$a3/city } </City>}
          </tuple>}
```

[1] The `LIKE` operator acts only as an interim solution and has to be transformed in accordance with the rules described in [7]. The corresponding XQuery expression depends on the function `fn:substring()`.

```
    </result>)
  return
   <result>
     { for $groupBy1 in distinct-values($result/tuple/Name ),
           $groupBy2 in distinct-values($result/tuple/City )
       let $new := $result/tuple[Name = $groupBy1 and
                                  City = $groupBy2]
       return
         if ($new != " ") then (
           <group>
             <Name> {$groupBy1 } </Name>
             <City> {$groupBy2 } </City>
             {$new }
           </group> ) }
  </result> )/child::node()
return
  <tuple>
    <Name> {$final/Name } </Name>
    <City> {$final/City } </City>
  </tuple>
```

This XQuery statement along with the name of the corresponding schema can be forwarded to the XML CDBS. Based on these information, the latter is able to determine a document collection whose documents are valid respect to the given schema and can apply the generated query to this collection.

4 Related Work

With the exception of the approach of Escobar et al. mentioned in Sect. 1, we are not aware of other architectures which realise an algorithm for transforming SQL queries into XQuery statements. Though, there are many commercial systems which integrate disparate data sources (in most cases (object-)relational and XML-based data) into a global schema, these systems either allow only one query language according to the global data model or they forward the XQuery and SQL statements only to the corresponding data sources.

The commercial BizQuery Suite by ATS [2], for instance, is a software system for virtual integration of disparate data, which are provided by a unified XML-based view. In this way, XQuery requests across multiple data sources are possible, but no transformation of SQL into XQuery statements is performed. The same approach is realised by the first type 4 JDBC driver for XML files by Sunopsis [8]. The driver loads (upon connection or user request) the XML, EDI or IDoc structure and data into a relational schema, using an XML to SQL mapping. After that, the user works on the relational schema, manipulating data through regular SQL statements or specific driver commands. Upon disconnection or user request, the XML driver is able to synchronise the data and the structure stored in the schema back to the XML file. Similar to the BizQuery system the JDBC driver implements the querying according to heterogeneous

data sources by mapping the data into a global schema, which is either object-relational or XML-based, but not by translating the requests.

5 Summary & Future Prospects

The aim of the introduced algorithm is the fully automatic transformation of a global SQL request into an equivalent XQuery expression which can be delivered to an XML component database system with relating to an XML document collection. The only precondition for that is a semantics conserving mapping of the local schema description onto the federation service's global schema. Based on this mapping and the current SQL request, the corresponding XQuery statement is generated. The presented example of such a transformation demonstrates the application of the algorithm.

However, the algorithm as introduced here is restricted to simple structured queries and was implemented prototypically to demonstrate the feasibility in principle. The realisation of further constructs, as for instance the `LIKE` operator or nested queries, is discussed in [7]. The application to SQL extensions is also imaginable. An interesting SQL extension in this context is MM/Full-Text. By the provided information retrieval techniques, which allow comparisons with stemming or distance based queries, the full-text supplement of XQuery could also be realised by this algorithm.

References

1. Bell, D., Grimson, J.: Distributed Database Systems. Addison-Wesley (1992)
2. BizQuery: A Software System for Virtual Integration of Disparate Data. ATS. `http://www.atssoft.com/products/bizquery.htm` (22 August 2005)
3. Sheth, A. P., Larson, J. A.: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. ACM Computing Surveys, Vol. 22, No. 3 (September 1990)
4. DeHaan, D., Toman, D., Consens, M.P., Özsu, M.T.: A comprehensive XQuery to SQL translation using dynamic interval encoding. In Proceedings of the 2003 ACM SIGMOD Int. Conf. San Diego (2003)
5. Elmagarmid, A., Rusinkiewicz, M., Sheth, A.: Management of Heterogeneous and Autonomous Database Systems. Morgen Kaufmann Publishers Inc. (1999)
6. Escobar, F., Espinosa, E., Lozano, R.: XML Information Retrieval Using SQL2Xquery. Tecnolgico de Monterrey-Campus cd. De México. Departmento do Computacin (2002)
7. Jahnkuhn, H.: Transformation von SQL- in XQuery-Anfragen innerhalb föderierter Informationssysteme. University of Rostock. Department of Computer Science. Student research project (2005)
8. JDBCforXML: Sunopsis XML Driver (JDBC Edition) `http://www.sunopsis.com/corporate/us/products/jdbcforxml/` (22 August 2005)
9. Michiels, P.: XQuery Optimization. In Proceedings of VLDB 2003 PhD-Workshop. Berlin (2003)
10. Rost, G.: Implementierung von XQuery auf objektrelationalen Datenbanken. University of Rostock. Department of Computer Science. Diploma thesis (2002)