Towards an Exhaustive Set of Rewriting Rules for XQuery Optimization: BizQuery Experience

Maxim Grinev[1] and Sergey Kuznetsov[2]

[1] Moscow State University, Vorob'evy Gory, Moscow 119992, Russia
grinev@acm.org,
WWW home page: http://www.ispras.ru/~grinev
[2] Institute for System Programming of Russian Academy of Sciences,
B. Kommunisticheskaya, 25, Moscow 109004, Russia,
kuzloc@ispras.ru

Abstract. Today it is wildly recognized that optimization based on rewriting leads to faster query execution. The role of a query rewriting grows significantly when a query defined in terms of some view is processed. Using views is a good idea for building flexible virtual data integration systems with declarative query support. At present time such systems tend to be based on the XML data model and use XML as the internal data representation for processing query over heterogeneous data. Hence an elaborated algorithm of query rewriting is of great importance for efficient processing of XML declarative queries.
This paper describes the query rewriting techniques for the XQuery language that is implemented as part of the BizQuery virtual data integration system. The goals of XQuery rewriting are stated. An algebra for rewriting is proposed. Besides query rewriting rules for FLWR expressions the rules for XQuery functions and recursive XQuery functions are presented. Also the role of the XML schema in query rewriting is discussed.

1 Introduction

It is widely accepted doctrine that query languages should be declarative. As a consequence of this there are frequently several alternative ways to formulate a query. It is noticed that different formulations of a query can provide widely varying performance often differing by orders of magnitude. Relaying on the reasoning, sophisticated techniques for query transformations for traditional query languages such as SQL was worked up [16–20]. Using the techniques allows rewriting a query into equivalent one that can be executed faster. The general characteristics of query rewriting can be summarized as follows:

– The phase of rewriting optimization follows query parsing and precedes query plan generation and optimization.
– Rewriting optimization transforms a query into equivalent one and certainly doesn't make queries worse in respect to their execution time.

– Query rewriting is usually carried out on the basis of information obtained from the query itself, views to which the query is addressed, integrity constraints and the schema of data queried. The important note is that data and even statistic about data are not involved in query rewriting. It is used at the phase of cost based plan optimization.

The emergence of XQuery [3] as pretending to be the standard declarative language for querying XML data [1] calls for rewriting techniques that meet the same challenges as those for traditional query languages but developed in new XQuery terms. This paper is devoted to a comprehensive discussion of XQuery rewriting in the presence of views and/or data schema.

Moving towards an exhaustive set of rewriting rules for XQuery, we have identified the optimization tasks that can be naturally accomplished at the phase of rewriting. Some of those tasks seem to be solved by us in the BizQuery virtual integration system [21], and the rules and algorithms on which the implementation is based are described in this paper. For the rest tasks, preliminary ideas are considered.

We have realized the need for advanced XQuery rewriting facilities during the work on the BizQuery virtual integration system. This work helps us not only to understand the significant importance of XQuery rewriting but also reveal goals of rewriting optimization in respect to XQuery. Therefore, in order to make our ideas more clear we would like to say a few words about the BizQuery project.

1.1 BizQuery Integration System and Needs for Query Rewriting

BizQuery is a system for querying data across multiple heterogeneous data sources in a uniform way. In essence, a global schema is created to represent a particular application domain and data sources are mapped as views onto the global schema. Global schema is described in a XML schema definition language (BizQuery supports Relax NG [4]), source data schemas are described in Relax NG, and views are specified in XQuery. BizQuery employs virtual approach to data integration and supports XQuery as external interface. The user asks an XQuery query over the global schema and the data integration system merges it with the views to which the query is addressed. Then the merged query is optimized and splitted up into subqueries over the data sources. There subqueries are translated into query languages supported by data source management systems and are sent to the sources. In case of a cross-source query (i.e. a query addressing two or more sources), the subqueries are executed on the side of data sources to be integrated, and part of the query is executed on the side of BizQuery (in general, this part of query is very important and needs to be carefully optimized). At that BizQuery tries to select the biggest subqueries to send to data sources. More on BizQuery can be found in [21].

Our aims were not to restrict facilities for global schema definition by the structural organization of data sources to allow presenting integrated data in any desired form. This is accomplished by specifying views over the source data structures. It required using the XQuery transformational facilities to full extent.

Notice that because the current version of XQuery [3] provides the only way to perform transformation in XQuery: to build new XML structures from the existing ones using the XML element and attribute constructors that will form the desired XML data. Analyzing the obtained views, it becomes obvious that queries with predicates issued to transformational views can be rewritten to equivalent queries changing the order of the operations and applying predicate as soon as possible. This well known practice is called “predicate push-down”. It allows the system not to materialize the whole view before the predicates are applied. Instead of this the system may transform smaller amount of data after their selection with the pushed predicates. Besides it does not often require the information from schema to perform rewriting because the necessary structural information is captured in the view definitions in the form of XML element and attribute constructors. “Predicate push-down” technique is not restricted by dealing with transformational queries but also may be useful to optimize queries containing “joins”.

Also implementing XQuery for data integration purposes we encountered the following difficulty. A subset of XQuery operations relies on the notion of unique identity as defined in [2]. For instance, the union operator, that takes two sequences as operands and returns the sequence containing all the items that occur in either of the operands, eliminates duplicates from their result sequences basing on the identity comparison. Unique id is an internal thing of data source. Thus such operations must be passed to the data source for processing but it can be impossible in case of cross-source queries. Applying rewriting techniques might make it possible to rewrite the query into equivalent one that doesn’t contain such operations.

1.2 Goals of XQuery rewriting

Analyzing BizQuery experience and works on rewriting optimization for traditional query languages [16–20], we have tried to state the goal of XQuery rewriting.

The goal of XQuery rewriting is fivefold:

– *Perform natural heuristics.* Certain heuristics can be used in XQuery rewriting and are generally accepted in the literature as being valuable. Examples of those are “predicate push-down”, in which predicates are applied as early as possible in the query (i.e. they are “pushed” from their original positions into sub-queries, views, etc) and “eliminate redundant computation” (i.e. reduce expressions that can be computed without data access).
– *Perform natural heuristics in the presence of calls to the user defined functions.* User defined functions (formulated in XQuery) are very important because some queries, such as queries to recursive XML structures, cannot be expressed without using such functions. That is why query rewrite engine should be capable of performing natural heuristics described above for queries with user defined function calls.

– *Make queries as declarative as possible.* In declarative languages such as XQuery, several alternative formulations of a query are often possible. These expressions can enforce plan optimizer into choosing query execution plans that are varying in performance by order of magnitude. Some of such query formulations might be more "procedural" than others enforcing a way of query execution. A major goal is the transformation of such "procedural" queries into equivalent but more declarative queries for which more query execution plans can be generated.
– *Transform a query into "well-aimed" one on the basis of schema information.* XQuery allows formulating queries when the user have vague notion about schema. Execution of such queries can lead to superfluous data scanning. Sometimes it can be avoided by means of rewriting the query into one returning the same result but scanning less data.
– *Eliminate operations based on identity.* Such operations can be unfeasible in distributed environment but they might be transformed into expressions without such operations as it was argued in 1.1.

Notice that only the last goal is specific for integration system while others are useful for XML DBMS with local data storage.

1.3 Related work

Research on XQuery optimization is now at the early stage. There is only a few works on that. As regards XQuery rewriting, a suggestive rather than complete set of rules is given in [9]. In [10] a set of equivalent transformation rules are defined that bring a query to a form which can be directly translated to SQL, if possible. Although these rules are designed to facilitate XQuery to SQL translation, they are also useful for general-purpose optimization such as predicate push down and prior normalization rules that prepare the query to be processed by rewrite engine.

We were inspired by works on rewriting optimization for SQL [16–20]. These are a thorough research on the matter and much of this can be adapted to rewriting XML query languages.

Also due to strong similarity between XML and semi-structured data [11–13], works on optimization for semi-structured query languages [14, 15] can be of much help.

1.4 Structure of the paper

Section 2 presents the abstract representation of queries used by the BizQuery rewriter. The rewriting rules for predicate push down are presented are presented in Section 3. An extension of these rules for rewriting queries with calls to user-defined functions is described in Section 4. Future work and conclusion appear in Section 5.

2 Algebra XML Schema Definition Language for XQuery Rewriting

Fig. 1. defines the grammar in BNF for the algebra (i.e. the convenient syntax in which rewriting rules will be specified and that can be directly mapped onto in-memory structures for internal representation).

```
<name>::= /*element or attribute name*/
<fun-name>::= /*function name*/
<var>::=  /*variable name*/
<const>::= /*constant value of Boolean, Integer or String type*/
<test>::=
/*it is used in children and descendant to filter nodes*/
node() /*returns nodes of all kinds (element, attribute,
text, etc.)*/
/*returns elements with the given name*/
| element-test(<name>)
/*returns all elements*/
| element-test(*)
/*returns attributes with the given name*/
| attribute-test(<name>)
/*returns all attributes*/
| attribute-test(*)

<function definition> ::= fun <fun-name>(<var>, ...,<var>) = <exp>

<exp> ::= <const>
| <var>
| for <var> in <exp> do <exp> /*iteration*/
| if <exp> then <exp> else <exp> /*conditional*/
/*element and attribute constructors*/
| element(<tag-name>, <exp>) | attribute(<tag-name>, <exp>)
/*computed element and attribute constructors*/
| element(<exp>, <exp>) | attribute(<exp>, <exp>)
/*XML node tree traversal*/
| children(<exp>,<test>) | descendant(<exp>,<test>)
| parent(<exp>)
/*set-theoretic operations*/
| union(<exp>, <exp>) | intersect(<exp>, <exp>)
| except(<exp>, <exp>)
| sequence(<exp>, <exp>) /*sequence concatenation*/
| () /*empty list*/
| <exp> * <exp> | <exp> > <exp> | <exp> = <exp> | <exp> and <exp>
| <fun-name>(<exp>, ...,<exp>) /*function call*/
| dereference(<exp>)
/*existential and universal quantifiers*/
```

```
| some <var> in <exp> do <exp> | every <var> in <exp> do <exp>
| empty(<exp>) /*is the sequence empty?*/
| name(<exp>) /*name of element or attribute*/
| node-kind(<exp>) /*returns "element", "attribute", "text"*/
```

Fig.1. Algebra Grammar

Our algebra is not complete but captures the essence of XQuery. It is close to that used in [9]. Some of the above expressions are in fact shorthands and can be expressed in others (e.g. some expression can be rewritten using for, if, empty). They are left in their original form because, if rewritten, they become less declarative and enforce some concrete implementation.

Almost all XQuery expressions are translated directly into the algebra, except let and where clauses. Basic transformation principles are:

– A sequence of for-clauses is translated into nested for-iterators.
– Let-clause is treated as syntactic sugar and all occurrences of the variable defined in a let-clause are replaced with the bound expression.
– Where-clause in for v in exp where exp1 return exp2 is treated as syntactic sugar for if-expression in the algebra and is translated as follows: `for v in exp do if expr1 then exp2 else ()`.

Consider an example of typical XQuery query and its translation into the algebra.

```
FOR $e IN document("employees.xml")/data/employee,
    $d IN document("department.xml")/data/department
LET $dep_name:=$d/name/node()
WHERE $e/dep-no/node()=$d/no/node() and $e/age/node() > 25
RETURN
  <person>
    <name>{$e/name/node()}</name>
    <department_name> {$dep_name} </department_name>
  </person>
```

The query demonstrates a FLWR expression (i.e. a series of FOR, LET, WHERE and RETURN clauses) that is the main pattern for XQuery query formulation. XQuery query is usually composed of FLWRs that can be nested with full generality. In this query `$e` iterates over employees stored in employee.xml document. For each value bound with `$e`, `$d` iterates over departments stored in department.xml document. For each pair of bindings `($e, $d)`, the value of `$dep_name` in the LET-clause is computed. Then each triple of the form `($e, $d, $dep_name)` is subject to further filtering by the WHERE-clause. RETURN-clause containing constructors of XML elements is executed once for each triple that is generated by FOR and LET-clauses and satisfies the condition of the WHERE-clause. This query is translated into the following expression in the algebra:

```
for $e in children(children(document("employees.xml"),
                                  element-test(data)),
                   element-test(employee))
do
  for $d in children(children(document("department.xml"),
                                element-test(data)),
                       element-test(department))
  do
    if  children(children($e, element-test(dep-no)), node()) =
        children(children($d, element-test(no)), node())
        and
        children(children($e, element-test(age)), node()) > 25
    then
     element(person,
             sequence(
               element(name,
                       children(children($e, element-test(name)),
                       node())),
               element(department_name,
                       children(children($d, element-test(name)),
                       node()))))
    else ()
```

In the introduction we discussed that the schema of XML document queried can be used in XQuery rewriting, so availability of the schema is assumed when XQuery rewriting implementation in BizQuery is concerned. For the time being, several languages for schema definition exist (e.g. DTD [1], XML Schema [5, 6], Relax NG [4]). BizQuery supports Relax NG, but the language used to describe the schema is not so important with respect to query rewriting. It is because schema information is used mainly through type inference. By "type inference" we mean here that a schema of a given query result is constructed on the basis of the query and the schema of the addressed XML document. Thus, choosing any specific schema definition language mainly determines type inference techniques used but not rewriting ones. Examples of this paper use DTD because of its compact syntax.

3 Predicate push down rewriting rules

Query simplification by rewriting can often reduce the size of the intermediate results computed by a query executor. It can be achieved by changing the order of operations to apply predicate as soon as possible. Let's consider an example. Suppose we have a query formulated as follows (algebra defined in the previous section is used to present the query):

```
for s in
  (for b in children(children(document("catalog"),
                                element-test(catalog)),
                       element-test(book))
```

```
   do
     element(book,
       sequence(
         element(title,children(children(b,
                                         element-test(title)),
                                node())),
         element(price,children(children(b,
                                         element-test(price)),
                                node())*2))))
do
  if children(children(s, element-test(title)), node()) =
     "Seven years in Tibet"
  then s
  else ()
```

This query might have been obtained as the result of merging a transformation view (that returns all books from catalog with the title and price doubled) and a query with predicate (that selects all books named `"Seven years in Tibet"`). If the query could be given to the execution engine that processes the operations in the order specified in the query, it would lead to constructing the new book element containing title and doubled price for each book in the database and then comparing the title of the book with the string `"Seven years in Tibet"`. Thus, transformation is performed for all books while a few of them have the title specified in the query and will be returned as the query result.

In order to avoid undesired overheads as in this example, we propose a set of query rewriting rules (see Fig. 2) mainly aimed at pushing predicate down.

```
(1) for v2 in (for v1 in e1 do e2) do e3 =
    for v1 in e1 do e3{v2:=e2}
(2) for v in element(e1, e2) do e3 = e3{v:= element(e1, e2)}
(3) for v in (if e1 then e2 else e3) do e4 =
    if e1 then (for v in e2 do e4) else (for v in e3 do e4)
(4) for v in e1 do (if e2 then e3 else e4) =
    if e2 then (for v in e1 do e3) else (for v in e1 do e4)
    /*if there is no occurrence of v in e2
    (e2 is independent of v)*/

/* ? is child or descendant */
(5) for v in e1 do ?(v, node-test) = ?(e1, node-test)

/* ? is union or intersect or except*/
(6) for v in sequence(e1, e2) do e3 =
    sequence(for v in e1 do e3, for v in e2 do e3)

/* ? is some, every */
(7) for v in (? v1 in e1 satisfy e2) do e3 =
    e3{v:= (? v1 in e1 satisfy e2)}
```

```
/* ? is child or descendant */
(8) ?(for v in e1 do e2, node-test) =
    for v in e1 do ?(e2, node-test)
(9) ?(if e1 then e2 else e3, node-test) =
    if e1 then ?(e2, node-test) else ?(e3, node-test)
(10) ?(element(e1,e2), node-test) =
     for v in e2 do (if c then v else ())
    /*c is an expression constructed by node-test. For example,
    if node-test is element-test(name) then will be
    node-kind(v)="element" and name(v)="name"*/

/* ?? is union or intersect or except or sequence */
(11) ?(??(e1,e2), node-test) =
     ??(?(e1, node-test), ?(e2, node-test))

/* ? is union or intersect or except or sequence */
(12) e1 ? (if e2 then e3 else e4) =
     if e2 then (e1 ? e3) else (e1 ? e4)
(13) e1 union element(e2,e3) = sequence(e1, element(e2,e3))
     /*because element returns a new element with a new id*/
(14) e1 intersect element(e1,e2) = () /*for the same reason*/
(15) e1 except element(e1, e2) = e1 /*for the same reason*/

/* ? is some or every */
(16) ? v in (for v1 in e1 do e2) satisfy e3 =
     ? v in e1 satisfy e3{v1:=e2}
(17) ? v in (if e1 then e2 else e3) satisfy e4 =
     if e1 then (? v in e2 satisfy e4) else (? v in e3 satisfy e4)
(18) ? v in e1 satisfy (if e2 then e3 else e4) =
     if e2 then (? v in e1 satisfy e3) else (? v in e1 satisfy e4)
     /*if e2 is independent of v*/
```

Fig. 2. Rewriting rules for predicate push down (vi - variable name; ei - any expression; e1v:=e2 - replace all occurrences of v in e1 with e2)

Rules presented in Fig. 2 are got by examining pairwise combinations of the algebra operations. There are also rules that enable rewriting expressions where some arguments of operations are constants that unambiguously determine the values of the expressions (e.g. `for v in () do e = ();` `name(element(book,e)) = "book";` etc.). These rules are straightforward and not presented here. For a query addressed to a virtual document (i.e. the content of a document defined by view) there is a very useful rewriting rule for a modified version of dereference. In the algebra dereference is defined as `dereference(e)` where `e` is an expression that should return a sequence of string values, and the result is a sequence of all XML elements in the queried document that have an attribute of type ID and its value belongs to the sequence returned by e. This definition conforms to the XQuery specification. However the problem is that which attribute has type ID is known only when the schema is available. Even if the schema is available, we might want to treat some attributes obtained in the result of a query (e.g. attributes in a virtual document) as having type ID but this treatment cannot be specified in XQuery query and cannot be inferred. Some implementations (e.g. [22]) support a modified dereference. The modified version of dereference is `dereference(e, element-name, attribute-name)` where e is as in the standard one and the result is all XML elements that have the given element name and attribute name, and the value of the attribute belongs to the sequence returned by e. For this modified version, the following rule eliminates dereference allowing going on rewriting:

```
dereference(e, element-name, attribute-name) =
 for v in descendant(document("foo.xml"),
                     element-test(element-name))
 do
   if some v1 in e
        satisfy v1 = children(v, attribute-test(attribute-name))
   then v
   else ()
```

Applying rules described in this section to the above example, we can get more optimal query (rules applied are 1, 10, 6, 2, 10):

```
for b in children(children(document("catalog.xml"),
                           element-test(catalog)),
                  element-test(book))
do
  if children(children(b,element-test(title)), node()) =
     "Seven years in Tibet"
  then
    element(book,
      sequence(
        element(title,children(children(b,
                                        element-test(title)),
                               node())),
        element(price,children(children(b,
                                        element-test(price)),
                               node())*2))
  else ()
```

4 Rewriting rules for queries with calls to user defined function

XQuery supports the notion of function defined in XQuery itself that can be treated as a parameterized query called setting the values of parameters. Function definition can be recursive. Rules described in the previous section can be used to rewrite queries that don't contain function calls though using functions in query definitions is common practice and a rewriting technique that is not able to handle such queries cannot be considered full value. In this section an approach to rewriting queries with function calls is proposed. Before getting down to considering the approach itself, let's say a few words about why functions are so important in XQuery and techniques for efficient processing function calls is of so great demand. The point is that functions in Query serve not only as a mean to improve query modularity and expression reuse but there are also important tasks that cannot be accomplished without using functions. Thus, the basic capability of XQuery is implemented in functions. A very illustrative example of that is provided by transformational queries like the following: To leave the whole XML document as it is except doubling the contents of all elements named "price". If the XML document does not contain recursive element definitions with unlimited level of nesting, in the presence of a schema of this document we can express the query using subqueries nested in XML element constructors. This query will rebuild the document doubling only the content of price elements and reproducing all the rest without modification. But this query can be formulated in a shorter form using recursive function traversing the document (that is presented as a tree in the XML model). Moreover, it can be done without employing information from the document schema that significantly simplifies writing the query. Further to that, in case of recursive XML element definitions, the only way to express the query is to use a recursive function!

Keeping all considered above in mind, let's proceed to discussion of the rewriting optimization for queries with function calls. We believe that a more effective query can be obtained replacing the function call with the function body with proper actual parameters substitution during query rewriting. Though such a strategy of rewriting does not allow using the advantages of subquery reuse, often allows reducing the query considerably that leads to more efficient query processing in the end.

Such kind of rewriting allows us to use all the rules specified in the previous section without any modification. But the problem that immediately arises is how to avoid an infinite loop of function call replacements when recursive functions are concerned. This problem follows from the fact that the condition of recursive function termination depends on data in general and thus cannot be checked in many cases during rewriting because the rewriter doesn't have access to the data. But having the schema of the XML document queried and involving type inference some conditions can be evaluated without access to the data. An example of such condition is `name(\$k)="Book"`. The condition compares the name of an XML element bound to `$k` with `"Book"`. The result of the condition can be evaluated inferring the type of the expression bound to `$k`.

In the current version of the rewriting engine we implement an algorithm that is not general but is suitable for many queries used in practice according to our experience. This algorithm can identify recursive function that may be rewritten with the function call replacement technique without leading to the infinite replacement loop. The algorithm takes the function definition as input and returns whether calls to the function can be replaced with the function body for subsequent rewriting or not. If yes, the replacements take place, if no, calls to the function are remained unprocessed by the rewriter engine. The following is a brief description of the algorithm.

The algorithm is based on the assumption that expressions of two types are used within functions in queries to terminate recursive calls (processing of other types of expressions can be added in future when identified):

– If-expression
  (i.e. `if <condition> then <expression1> else <expression2>`) when only one expression (i.e. `expression1` or `exression2`) contains recursive function call(s).
– For-expression
  (i.e. `for <variable name> in <expression1> do <expression2>`) when recursive function calls are in `expression2` and the termination condition is `empty(expression1)` because in this case `expression2` is not evaluated.

The algorithm is two-step.

At the first step, for given function definition, a set of all termination conditions found in the function body is constructed by gathering all termination conditions of the form described above.

At the second step, termination conditions in the set constructed during the first step are analyzed with the object to find out whether the condition can be evaluated without accessing data (i.e. using type inference on the basis of information obtained from the schema of a document to which the query is addressed).

This analysis can be described in the form of two-state machine. The states are normal and anxious. For each state all query operations are divided into groups. Roughly speaking, the machine traverses the termination condition in wide-first order and depending on what group the current operation belongs to, its arguments are analyzed by the machine in different ways: the machine changes its state or goes on analyzing in the same state or stops identifying that condition can not be evaluated. Group membership is verified according to the current state of the machine. The division into groups is needed because, for instance, the child operation inside argument expression of the name operation can be evaluated over the schema while it cannot be done if child is inside an argument of the comparison operation (e.g. `>`). If machine manages traversing the condition to the end without identifying that condition cannot be evaluated then this condition can be evaluated over the schema.

More precisely, the machine starts in the normal state and traverses the analyzed condition taking the current operation and acting depending on the current state:

– If the machine is in the normal state then

    - if the current operation belongs to the group of suspicions operations: `=`, `>`, `*`
    - then the machine moves into the anxious state because evaluation of these operations requires involving data and analyzes each argument of the current operation.
    - else the current operation does not belong to the suspicions group (it can be `name`, `empty`, `node-kind`, `for`, `or`) and the machine goes on analyzing argument expressions in the normal state because the current operation can be evaluated over the schema.
  - If the machine is in the anxious state then
    - If the current operation belongs to the group of fatal operations: `child`, `descendant`, `parent`
    - then the condition is identified as one that can not be evaluated without involving data.
    - else
      - If the current operation is for
      - then the machine stays in the anxious state and goes on analyzing the arguments in the do-clause because for is just an iterator and the result of it depends on the expression in the do-clause statement
      - else the current operation is not fatal and not for (it can be `name`, `empty`, `node-kind`, `and`, `or`) and the machine move into the normal state and goes on analyzing the arguments.

Let's consider an example of rewriting a query containing a call to a recursive function to show how principles and algorithm described above works in practice. Suppose the schema of a document queried is (presented in DTD):

```
<!ELEMENT catalog (book | magazine)*>
<!ELEMENT book (title, price)>
<!ELEMENT magazine (title, price)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

The definition of a recursive function that traverses the document and doubles price of all books remaining all the rest unchanged:

```
fun transform($s) =
  for $k in children($s, node())
  do
    if name($k)="price" and name(parent($k))="book"
    then element(price, children($k, node())*2)
    else
      if node-kind($k)="text"
      then $k
      else element(name($k), transform($k))
```

This function doesn't depend on a schema of a document to be processed.

A query with call to this function is *"select books with price greater then 20"* after applying function "transform" to the document:

```
for $c in
  children(transform(element("input",
                           children(document("catalog.xml"),
                                    element-test(catalog)))),
  element-test(book))
do
  if children(children($c, element-test(price)), node())>20
  then $c
  else ()
```

Rewriting the query, rewriter should check whether calls to transform could be replaced with the function body without resulting in infinite loop or not. The algorithm works as follows:

At the first step, termination conditions in transform is identified. It is a set of expressions: `empty(children($s, node()))`, `name($k)="price"` and `name(parent($k))="book"`, `node-kind($k)="text"`.
At the second step, rewriter analyzes whether these expressions can be evaluated on the schema (i.e. the state machine works). Because all of them can be evaluated by means of type inference, rewriting with function body substitution can be performed. After rewriting, we get another expression of the original query with predicate pushed down (this transformed query depends on the schema of "catalog.xml"):

```
for $k in children(children(document("catalog.xml"),
                            element-test(catalog)),
                   node())
do
  if name($k)="book"
     and
     children(children($k, element-test(price)), node())*2 >20
  then
    element(book,
      sequence(
        element(title, children($k, node()) ),
        element(price, children($k, node())*2)))
  else ()
```

5 Conclusion and Future work

The results described in this paper have been implemented and are used successfully within the BizQuery prototype system. The techniques of XQuery query rewrining have given us a possibility to decompose an original query with substituted views and exposed (if possible) function calls into a set of "maximal" subqueries to be executed by local DBMs. This techniques also provide us with a possibility to generate a rich set of final part of query execution on the side of BizQuery engine. The main conclusions of this work are as follows:

– Query rewriting is extremely important for XML-based virtual data integration systems.
– The major part of rewriting techniques that have been developed within this work may be useful for XQuery optimization in native XML-based systems too.
– It's very reasonable to continue the work to get a really exhaustive set of rewriting rules for XQuery optimization to archive all goals identified in the Introduction to the present paper.

In introduction to this paper the goals of XQuery rewriting were identified. For the first two the techniques were proposed in this paper. The last three needs to be archived: make queries as declarative as possible, transform a query into "well-aimed" one on the basis of schema information, and eliminate operations based on identity. Here some preliminary ideas are discussed. Making queries more declarative is well elaborated for relational query languages such as SQL [16–20]. The major strategy used is to rewrite subqueries into joins because there are more options in generating execution plans for them that increases possibility to find the most optimal one. It is not quite right when XQuery is concerned. XQuery is not so declarative and there is less freedom in generating execution plans for the XQuery "join". It follows from the fact that XML items are ordered as defined in [2]. Join in XQuery is expressed as nested for iterator and the outermost for expression determines the order of the result. It means that XQuery join doesn't commute as relational join does that doesn't allow evaluating join in any order. But XQuery also supports for unordered sequences, which enables commutable joins. In this case the relational techniques mentioned above seems to be adaptable for the purpose of XQuery optimization.

The execution of some XQuery queries may lead to data scanning that is a priori unnecessary. But support for such queries is very useful because it allows the user to formulate queries not keeping the schema of the addressed data in mind. This unnecessary data scanning can be avoided if the query is rewritten in more precise one by analyzing the schema. For example, suppose we have a DTD:

```
<!ELEMENT catalog (book | CD )*>
<!ELEMENT book (title, ISBN, price)>
<!ELEMENT CD (performer, title, price)>
```

and a query is `/catalog/*[ISBN/text()="foo"]`. To process this query the system should scan book and CD elements. This query can be rewritten in `/catalog/book[ISBN/text()="foo"]` because it is known from the DTD that only book elements contain ISBN subelement. This new query returns the same result as the original query but scanning CD elements is not performed. We are planning to identify cases when a data schema can help and support them in future versions of BizQuery.

We argued in the introduction of this paper that execution of operations based on the identity is problematical in distributed systems. We have carried out some preliminary analysis of queries with such operations. It turns out that many reasonable queries with parent and union operations, for example, can be rewritten in equivalent queries that don't contain such operations, and complexity of new queries is comparable to that of original ones. Unfortunately, for some queries the rewritten form can be much more complex but we find these queries unreasonable and infrequent. It gives us a hope that such rewriting can be useful in practice. It is still not clear whether all identity-based operations can be rewritten. But it can be easily shown that ids in the XML data model are computable: id of XML item can be mapped onto the path from the document root to that item because XML item tree is ordered (it can even be rewritten using the XPointer abbreviated syntax [7], for example, `/1/3/4/1/3`). The fact that ids are computable might help to prove that all identity-based operations can be rewritten into others.

6 Acknowledgments

Thanks a lot to Kirill Lisovsky, Leonid Novak, and Andrei Fomichev for many discussions during the work on the XQuery rewriter and this paper.

References

1. "Extensible Markup Language (XML) 1.0", W3C Recommendation, 10 February 1998, http://www.w3.org/XML/
2. "XQuery 1.0 and XPath 2.0 Data Model", W3C Working Draft, 20 December 2001, http://www.w3.org/TR/query-datamodel/
3. "XQuery 1.0: An XML Query Language", W3C Working Draft, 20 December 2001, http://www.w3.org/TR/xquery/
4. "RELAX NG Specification", OASIS Committee Specification, 3 December 2001, http://www.oasis-open.org/committees/relax-ng/spec-20011203.html
5. "XML Schema Part 1: Structures", W3C Recommendation, 2 May 2001.
6. "XML Schema Part 2: Datatypes", W3C Recommendation, 2 May 2001.
7. "XML Pointer Language (XPointer) 1.0", W3C Candidate Recommendation, 7 June 2000.
8. Klemens Bohm, Kathrin Gayer, Karl Aberer, M. Tamer Ozsu, "Query Optimization for Structured Documents Based on Knowledge on the Document Type Definition", Advances in Digital Libraries Conference, April 1998
9. Mary Fernandez, Jerome Simeon, Philip Wadler. "A semistructured monad for semistructured data", ICDT, January 2001, http://www.research.avayalabs.com/user/wadler/topics/xml.html
10. Ioana Manolescu, Daniela Florescu, Donald Kossmann, "Answering XML Queries on Heterogeneous Data Sources", VLDB Conference, 2001, http://www.vldb.org/dblp/db/conf/vldb/vldb2001.html
11. S. Abiteboul. "Querying Semi-Structured Data", available at http://pubdb.stanford.edu/publist.html
12. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom. "Lore: A Database Management System for Semistructured Data". Available at www-db.stanford.edu/lore

13. S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Weiner. “The Lorel query language for semistructured data”. International Journal on Digital Libraries, 1(1):68-88, April 1997.
14. P. Buneman, S. Davidson, G. Hillebrand, D.Suciu. “A query language and optimization techniques for unstructured data”. In Proc. of ACM SIGMOD Conference On Management of Data, pages 505-516, Montreal, Canada, 1996.
15. Mary Fernandez, Dan Suciu. “Optimizing Regular Path Expressions Using Graph Schemas”, available at http://citeseer.nj.nec.com/fernandez98optimizing.html
16. Hamid Pirahesh, Joseph M. Hellerstern, Waqar Hasan. “Extensible/Rule based Query Rewrite Optimization in Starburst”, SIGMOD International Conference on Management of Data, 1992.
17. W. Kim. “On Optimizing an SQL-like Nested Query”, ACM Transactions on Database Systems, 7(3), September 1982.
18. Richard A. Gansky and Harry K. T. Wong. Optimization of Nested SQL Queries Revisited. In Proc. ACM-SIGMOD International Conference on Management of Data, pages 23-33, 1987.
19. Umeshwar Dayal. “Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates, and Quantifiers”, VLDB Conference, 1987.
20. Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, Raghu Ramakrishnan. “Magic is Relevant”, SIGMOD International Conference on Management of Data, 1990.
21. Maxim Grinev, Sergei Kuznetsov. “An Integrated Approach to Semantic-Based Searching by Metadata over the Internet/Intranet”, 5th East-European Conference on Advances in Databases and Information Systems (ADBIS), Professional Communications and Reports, Vol. 2, 2001
22. Kweelt project, http://sourceforge.net/projects/kweelt/