Rewriting-based Optimization for XQuery Transformational Queries

Maxim Grinev, Peter Pleshachkov
Institute for System Programming RAS, Moscow, Russia
maxim@grinev.net, peter@ispras.ru

Abstract

The modern XML query language called XQuery includes advanced facilities both to query and to transform XML data. An XQuery query optimizer should be able to optimize any query. For “querying” queries almost all techniques inherited from SQL-oriented DBMS may be applied. The XQuery transformation facilities are XML-specific and have no counterparts in other query languages. That is why XQuery transformational queries need to be optimized with novel techniques. In this paper two kinds of such techniques (namely push predicates down XML element constructors and projection of transformation) are considered. A subset of XQuery for which these techniques can be fully implemented is identified. This subset seems to be the most interesting from the practical viewpoint. Rewriting rules for this subset are proposed and the correctness of these rules is formally justified. For the rest of the language we propose solutions that work for the most of common cases or consider the problems we have encountered.

1 Introduction

It is a widely accepted doctrine that query languages should be declarative. As a consequence there may be 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. Consequently, sophisticated query optimization techniques based on rewriting the formulation of a query for relational query languages such as SQL were worked up [7, 16, 14, 9]. The techniques allow rewriting a query into an equivalent one that can be executed faster. General characteristics of query rewriting techniques can be summarized as follows:

- A query is rewritten into an equivalent one.
- The selection of kinds of rewriting to be applied is carried out heuristically. Query rewriting should be ameliorative for the majority of queries.
- Query rewriting is usually carried out on the basis of information obtained from the query itself, the views to which the query is addressed, the integrity constraints and the schema of the queried data. The important note is that data and even statistics about data are not involved in query rewriting.

The emergence of XQuery [2] as a standard declarative language for querying XML data [5] calls for rewriting techniques that meet the same challenges as those for traditional query languages but that are developed in the XQuery terms. In our previous papers [11, 13] we considered the kinds of query rewriting that are effective for XQuery optimization and proposed preliminary solutions for each kind. This paper provides complete solutions for the following two of these kinds that concern XML transformational queries with XML element constructors.

1. To push predicates down XML element constructors is to change the order of operations in an XQuery query to apply predicates before XML element constructors.

2. To perform projection of transformation is to compute in static (by means of query transformation) path operations that are applied to the result of XML element constructors (i.e. to new XML elements constructed during query evaluation).

From a theoretical point of view, these two kinds of query rewriting are interesting because they are the most XQuery-specific ones and require new techniques that have no counterparts among the techniques for traditional query languages. From a practical point of view, these kinds of query rewriting are of much benefit when queries that transform XML data are optimized. The only way to express data transformation in XQuery is to use XML element constructors.
XML element constructor is one of the most expensive XQuery operations because its evaluation requires deep copy of the XML tree that presents the content of the constructed XML element. The first kind of query rewriting helps to reduce the size of intermediate results to which XML element constructors are applied. The second one helps to remove redundant XML element constructors from the optimized query. We will call these two kinds of query rewriting the required kinds of query rewriting.

The rest of the paper is organized into nine sections. Section 3 is devoted to related work. In Section 4 we present a conceptual framework for the XQuery optimization techniques described in the next sections. Section 2 gives a motivating example. In Section 5 a query logical representation used for optimization is described. In Sections 6, 7 and 8 we consider rewriting-based optimization for different subsets of the XQuery language. In Section 9 we make up a conclusion and consider future work.

2 Motivating Example

Let us consider by an example how the required kinds of query rewriting contribute to query optimization. Suppose the user query an XML document that is named "db.xml" and satisfies to the following schema. The schema is expressed in the DTD language defined in [5].

```xml
<!ELEMENT stores (bookstore*)>
<!ELEMENT bookstore (name, book*)>
<!ELEMENT book (title, pub-year, authors, price)>
<!ELEMENT title (PCDATA)>
<!ELEMENT pub-year (PCDATA)>
<!ELEMENT authors (PCDATA)>
<!ELEMENT price (PCDATA)>
```

According to the schema, the document consists of a set of the `bookstore` elements, each of which contains information about books in the `book` elements, and each book is described by a title, a publication year, a list of authors, and price in the elements `title`, `pub-year`, and `authors` respectively. Suppose the user's query is as follows.

```xml
for $book in doc("db.xml")/stores/bookstore
where $book/title="Seven Years in Tibet"
return

  $bs/name,
  for $b in $bs/book
  for $book in doc("db.xml")/stores/bookstore
  where $book/title="Seven Years in Tibet"
  return $book

In this query there is a subquery in which the variable `$bs` iterates over all bookstores, and for each binding a new `bookstore` element is constructed. The content of each constructed `bookstore` element includes the name of the bookstore and newly constructed book elements. The content of each constructed `book` element is reproduced but with the price multiplied by the discount rate that is equal to 0.5. This subquery is an example of data transformation expressed in element constructors. The result of this subquery is projected by applying the path operation `.../book` that returns child subelements named `book`. The result of the projection is filtered out to select books satisfying the predicate posing the restriction on the book's title as equal to "Seven Years in Tibet". The latter is an example of querying with a predicate.

Such a query is very interesting from the practical point of view because it may ensue, for example, as the result of the substitution of a view definition into a query addressed the view. Such substitution technique is used in many implementations of databases systems for evaluating a query addressing some view. In our example, the inner transformational subquery could be a view definition and the outer filtering subquery is a query addressing the view. Such a query may also be generated by some sort of visual query constructor tool that allows the user to defined a query step by step. In our example, the inner transformational subquery presents the first step and the outer filtering subquery presents the second one.

The example query can be rewritten in the following equivalent form. The detailed step by step explanation for this example is given in Section 6.

```xml
for $book in doc("db.xml")/stores/bookstore
where $book/title="Seven Years in Tibet"
return

  <book>
    {$b/title,$b/pub-year,$b/authors,
    <price>{$b/price * 0,5}</price>}
  </book>
```

In the rewritten query the predicate is pushed down element constructors. This means that the `book` element is constructed only for the books that are tiled as "Seven Years in Tibet". Thus expensive element constructors are executed less number of times because there are likely few books with such title. Also the transformation is projected. This means that the `bookstore` elements are no longer constructed in the rewritten query. It allows avoiding deep copy of the bookstores' contents that include of all books stored in them!

3 Related Work

Research on the XQuery optimization is now at the initial stage. As regards XQuery query rewriting, a "suggestive rather than complete set of rules" is given in [10]. [8] presents a number of interesting ideas on query rewriting optimization for XQuery. In [15] a set of rewriting rules is defined that brings a query to a form which can be directly translated to SQL,
Conceptual Framework for XQuery Rewriting Optimization

Query rewriting optimization is traditionally implemented as a set of rewriting rules defined in terms of a logical representation. Logical representation is a set of algebraic operations in which any query can be expressed and which allows us to abstract from syntactic peculiarities of a language. The expression of a query in the operations of some logical representation is called query (logical) representation. In this paper we will use the term query meaning its logical representation because all techniques described in this paper are based on a logical representation. We use a logical representation that is very similar to XQuery Core [4]. This logical representation and its features will be described in Section 5.

The approach of implementing query rewriting via rewriting rules is powered by a general theory of rewriting [17]. According to the theory, a logical representation and a set of rewriting rules defined in terms of this logical representation form a rewriting system. Rewriting system can be investigated to determine its properties. The important property of a rewriting system is a normal form property. To define the property we have to define the notion of normal form at first. A normal form of a query (in respect of some rewriting system) is a query representation to which no rules of the rewriting system can be applied. A rewriting system has a normal form property if the application of rules to any query representation reduces it to a normal form. Normal form property ensures that the process of rule application to any query representation will not be infinite and results in some normal form. Now we can pose the rewriting optimization issue formally: define a rewriting system and prove its two properties: the normal form property and that the system implements the required kinds of query rewriting.

Investigating the possibility to define the required rewriting system, we have identified three groups of the XQuery operations that prevent from the implementation of the required kinds of query rewriting for some forms of queries in which operations from these groups are used. These groups are as follows: (1) outer path operations, (2) position-based operations, (2) identity-based operations. In Section 6 we consider a rewriting system for a subset of XQuery with having these operations excluded. We refer to this subset as basic. In Sections 7 we consider the extension of the rewriting system for position-based operations. This extension is orthogonal that is why they are considered separately. We have no optimization solution for queries containing outer path and identity-based operation. In Section 8 we explain the difficulties that we experience developing a rewriting system for these operations.

Logical representation of XQuery queries

This section contains the definition of an XQuery logical representation in terms of which rewriting rules presented in the following sections are defined. The logical representation is a set of operations defined in terms of XML Data Model [1] by means of mapping each operation to the expression in XQuery Core for which semantics is formally defined in [4]. The notation $[\text{expr}]$ means that the rules of mapping to XQuery Core are applied to the expression expr.

Operations of the logical representation are divided into three named groups. The name of a group is used to refer to the operations of this group from the other sections. The groups are divided into classes of operations which have the similar properties. Groups and classes are presented by the following subsections and sub subsections respectively.

For brevity, we omit operations for which optimization rules are similar to those, proposed in the following sections, for other operations from the same class. Nevertheless, the classification of the operations is designed to be complete in the sense that any XQuery operation defined in [2, 3] falls into one of the classes defined below.

5.1 Operations of the basic subset

Meta-operations

Meta-operations control the evaluation of query expression. The distinguishing feature of meta-operations that they do not produce new items by their own. They are used, for instance, to express repetition or to combine the input sequences into new sequences.

The return operation applies a given function \( f(x| e2) \), where \( x \) is the parameter name and \( e2 \) is a function body depending on \( x \) to each item of the given sequence (denoted by \( e1 \)). The function results are concatenated into the output sequence.

\[
[[\text{return}(e1, f(x| e2))]] = \text{for } x \text{ in } [[e1]] \text{ return } [[e2]]
\]

The seq operation combines input sequences into a single sequence.

\[
[[\text{seq}(e1, e2, \ldots, en)]] = \text{op:concatenate}([[e1]], \text{op:concatenate}([[e2]], \ldots, [[en]]))
\]
The if operation returns the result of e2 if the effective boolean value of e1 is true and the result of e3 otherwise. About the notion of effective boolean value, see below in this section.

\[[\text{if}(e1, e2, e3)]\] = let fs:new := [[e1]] return if fs:new then [[e2]] else [[e3]]

The ts operation\(^1\) applies a given function of a specific form to each item of a given sequence. Such function consists of one or more case clauses and a default clause. The function chooses one of several expressions to evaluate based on the dynamic type of the input item. Each case clause specifies a type pattern (denoted by TP\(i\)) followed by a return expression. The value of the ts operation is the value of the corresponding expression (denoted by ei) in the first case clause such that the type of the input item matches the type pattern in the case clause. If the type of the input item does not match to any type pattern specified in the case clauses, the value of the ts operation is the value of the expression in the default clause (this expression is denoted by en+1).

\[[\text{ts}(e, f(x | \text{cases(case(TP\(1\), e1), ... , case(TPn, en), def(en+1))})]] =

for $y$ in [[e]]
return typeswitch ($y$)
case TP\(1\) $x$ return [[e1]]
... case TP\(n\) $x$ return [[en]] default $x$ return [[en+1]]

Such operations on sequences as fn:insert-before, fn:remove, fn:subsequence, fn:unordered, and fn:item-at can be also treated as meta-operations but, for brevity, we do not include them in the logical representation.

Constructors

XQuery provides constructors that can create XML structures within a query. In this paper we consider only one of them, namely element constructor because the others are not essential with the respect to the optimization techniques proposed in this paper.

The element constructor (denoted by element) creates an XML element. The first argument specifies the name of the constructed element. The second argument specifies the its content.

\[[\text{element}(e-name, e)]\] = element { [[e-name]] } { [[e]] }

Inner Path operations

Path operations allows traversing an XML tree in the directions determined by axises. XQuery supports a number of axises, for instance, child, descendant, following-sibling, parent, and preceding-sibling. These axises can be classified into two classes: (1) inner axises that are directed inside the content of an item — child and descendant; (2) outer axises that are directed outside the content — all the other axises such as following-sibling, parent, and preceding-sibling. We refer to path operations that implement inter axises as inner path operations. The definition of the inner path operations is as follows.

\[[\text{child}(e, test)]\] =
for $x$ in [[e]]
return $x$/child::[[test]]

\[[\text{descendant}(e, test)]\] =
for $x$ in [[e]]
return $x$/descendant::[[test]]

\[[\text{test}]\] = elem(name) | text()

Quantifiers

Quantifiers support existential and universal quantification.

\[[\text{some}(e1, f(x| e2))]\] =
xf:not (xf:empty (for $x$ in [[e1]]

return if [[e2]] then $x$ else ()))

\[[\text{every}(e1 | f(x| e2))]] =
xf:empty (for $x$ in [[e1]]

return if xf:not([[e2]]) then $x$ else ())

Accessors

Accessors are used to get the property of XML nodes. In this paper, we consider only two accessors, namely name and node-kind.

The name accessor returns the name for XML element node kinds. For other kinds of nodes it returns the empty sequence.

\[[\text{name}(e)]\] = fn:name([[e]])

The node-kind accessor returns a string representing the node's kind. In this paper, we consider only the following node kinds: "element", "text".

\[[\text{node-kind}(e)]\] = fn:node-kind([[e]])

Operations over sequences of atomic values

The evaluation of all operations from this class consists of two steps. At the first step the atomization procedure defined in [2] is implicitly applied to all arguments of these operations. Atomization is a kind of type conversion. It is defined over sequence and consists in casting every item of the sequence into some atomic type. Thus the result of atomization is a sequence of atomic typed values or a dynamic error if some item cannot be cast to the atomic type. At the second step operations are computed over atomized arguments. We will refer to the operations of this class as avo (avo stands for atomic value operation). The avo class includes arithmetical operations, operations on

\(^1\)ts stands for typeswitch
Arity-based operations over sequences

Operations from this class possesses the following property: the result of operation is determined only by the number of items (i.e. arity) in the input sequence and does not depend on the items themself. For brevity, we include only one such operation (i.e. empty, see its definition below) in the logical representation. The examples of other operations that fall into this class are fn:exist, fn:zero-or-one, fn:exactly-one.

\[
[[\text{empty}(e)]] = \text{fn:empty}([[e]])
\]

EBV-based operations (EBV-op for short)

This set of operations possesses the following property: before computing these operations the effective boolean value (EBV) procedure defined in [2] is implicitly applied to all arguments of these operations. EBV is a kind of type conversion. It casts a sequence into Boolean according to some rules. For example, an empty sequence or the singleton sequence of zero-length string are treated as false whereas any non-singleton sequence is treated as true. EBV-based operations are not(e), and(e1,e2), and or(e1,e2). The mapping of these operations to XQuery Core is straightforward.

5.1.1 Auxiliary operations

1. seq-string(e) applies fn:string to each item of the input sequence e, then concatenates the strings into one.

2. seq-string2untypedAtomic(e) applies fn:string to each item of the input sequence e, then concatenates the strings into one, and casts the result string to type xdt:untypedAtomic defined in [2].

3. marked-name-type(e)
   - If e returns a value of type QName, that QName is cast to a special string type that we will call marked-name.
   - If e returns a string and that string can be cast to type QName defined in [2], that string is cast to marked-name.
   - If e returns a string but that string cannot be cast to QName, an dynamic error is raised as it is defined in [2].
   - If e does not return a QName or a string, a type error is raised as it is defined in [2].

4. marked-node-kind-type(e)
   - If e returns a string, that string is cast to a special string type that we will call marked-node-kind.
   - If e does not return a string, a dynamic error is raised as it is defined in [2].

5. extended-name(e)
   - If e returns a value of type marked-name, that value is cast to QName.
   - In other cases the result is fn:name(e).

6. extended-node-kind(e)
   (a) If e returns a value of type marked-node-kind, that value is cast to String.
   (b) In other cases the result is fn:node-kind(e).

5.2 Outer path operations

Outer path operations are path operations that implement outer axises (see the comments to the definition of inner path operations above).

5.3 Identity-based operations

The common property of the operations from this group is that they are based on the notion of unique identity. Support for unique identity allows distinguishing XML nodes that are equal by value but that are from different positions in the XML tree. It also allows ordering XML nodes in document order. Document order is defined in [2] by means of setting an order of XML tree traversal. The example of identity-based operations is distinct-nodes (i.e. eliminates duplicate nodes from the input sequence exploiting unique identity feature of the nodes) or node-before (i.e. tests whether the node occurs in document order before the other node).

5.4 Position-based operations

XQuery supports the notion of dynamic context. Dynamic context consists of a number of components the value of which can be accessed from the query expression by means of calls to the predefined functions. Some components of dynamic context are global and their values are the same for the whole query while others called focus are local and their values depends on the position in a query. Focus includes the following components: current item, index (i.e. order number) of the current item in the sequence processed and index of the last item in the sequence processed. The value of these components are accessed by the predefined functions fn:context-item(), fn:position(), fn:last() respectively. Focus is only available for predicates of the XPath expression.
6 Optimization rules for the basic subset

In this section we describe a rewriting system for the basic subset of the logical representation introduced in the previous section. We will refer to this system as basic rewriting system (BRS for short). We begin with a list of the BRS rules, then we consider an example of BRS application to an XQuery query and formally prove the normal form property of BRS and that BRS implements the required kinds of query rewriting. In order to make it easy to understand the rewriting rules defined below we divide them into several groups and provide comments on each group.

The first group of rules unnest meta-operation expressions by distributing and commuting computations. Each rule from this group actually abbreviates a number of rules, since the context variable stands for a number of different expressions. The notation \( E[\mathbf{e0}] \) stands for one of the five expressions given with \( E[e0] \) for a number of different expressions. The notation provides comments on each group.

The first group of rules unnest meta-operation expressions by distributing and commuting computations. Each rule from this group actually abbreviates a number of rules, since the context variable \( E \) stands for a number of different expressions. The notation \( E[\mathbf{e0}] \) stands for one of the five expressions given with expression \( e0 \) replacing the sign ? that appears in each of the alternatives.

\[
E[?] := return(? , f(x|e)) \\
| child(? , e) \\
| descendant(? , e) \\
| some(? , e) \\
| every(? , e) \\
| ts(? , e)
\]

\( E[\text{seq}(e1,\ldots , e_n)] \Rightarrow \text{seq}(E[e1],\ldots , E[e_n]) \) (1)

Rule 1 unnest \( \text{seq} \) expressions, since the \( E \) expression distributes over list concatenation. For instance, this rule does not hold if \( E \) would be an aggregate function. Rule 1 is a consequence of the automatic list flattening feature of the XQuery language. This rule is valid for all of the expressions denoted by \( E \) except quantifier expressions (i.e. \( \text{some}, \text{every} \)). For the latter the following two rules must be applied.

\[
\begin{align*}
\text{some}(\text{seq}(e1,\ldots , e_n), f) & \Rightarrow \\
\text{or}(\text{some}(e1, f), \ldots , \text{some}(e_n, f)) & \quad (1a) \\
\text{every}(\text{seq}(e1, \ldots , e_n), f) & \Rightarrow \\
\text{and}(\text{every}(e1, f), \ldots , \text{every}(e_n, f)) & \quad (1b)
\end{align*}
\]

\( E[\text{return}(e1, f(x|e2))] \Rightarrow \text{return}(e1, f(x|E[e2])) \) (2)

Rule 2 unnest \( \text{return} \) expressions nested in \( E \). As the previous rule this one holds because of the implicit list flattening feature of the language. For instance, one of the expansions of Rule 2 is the following, when \( E \) is taken to be \( \text{return}(?, f(x|e)) \).

\[
\begin{align*}
\text{return}(\text{return}(e1, f(x|e2)), f(x|e)) & \Rightarrow \\
\text{return}(e1, f(x|\text{return}(e2, f(x|e))))
\end{align*}
\]

Rule 2 is also not valid for quantifier expressions. In case of quantifiers the following rule must be applied.

\[
\begin{align*}
\text{quantifier}(\text{return}(e1, f(x|e2)), f(x|e)) & \Rightarrow \\
\text{quantifier}(e1, f(x|\text{quantifier}(e2, f(x|e)))) & \quad (2a)
\end{align*}
\]

The following two rules unnest \( \text{if} \) and \( \text{ts} \) expressions by commuting them.

\[
\begin{align*}
E[\text{if}(e1, e2, e3)] & \Rightarrow \text{if}(e1, E[e2], E[e3]) & \quad (3) \\
E[\text{ts}(e0, f(x) \text{ cases}(\text{case}(t1, e1), \ldots , \text{case}(tn, en), \text{def}(en+1))))], \text{test}) & \Rightarrow \\
\text{ts}(e0, f(x) \text{ cases}(\text{case}(t1, E[e1]), \ldots , \\
\text{case}(tn, E[en]), \text{def}(E[en+1])))} & \quad (4)
\end{align*}
\]

The second group of rules eliminates iterations over singleton sequences by means of the proper substitutions of the singleton sequence expression into the body of iterative operations. Under iterative operations we mean operations that take a function as argument. The context variable \( E \) stands for iterative operations (in each iterative operation, its body is denoted by \( e \)). The notation \( e1{x->e2} \) stands for \( e1 \) with \( e2 \) substituted to all free occurrences of \( x \) inside \( e1 \).

\[
E[?] := \text{return}(?, f(x|e)) \\
| \text{ts}(?, f(x|e)) \\
| \text{quantifier}(?, f(x|e))
\]

The following rule is an abbreviation for Rules 5,6,7 that can be constructed by replacing \( e \) with one of the singleton sequence expressions \( \text{avo}(\ldots), \text{element}(\ldots) \), and variable respectively.

\( E[e1] \Rightarrow e1{x->e1} \) (5,6,7)

The third group of rules eliminates element constructors to which inner path operations are applied. As defined in Section 5, there are two inner path operations, namely \text{child} and \text{descendant}. As also discussed in Section 5, the content of an XML element is sufficient to compute the result of the application of a inner path operation to the element. The content of an XML element built by element constructor is determined by the second argument of the constructor. Due to these features of inner path operation and element constructor, the application of the former to the latter can be rewritten so that the element constructor disappears. The main idea behind such rewriting rules is to build a predicate that will be used to select XML items satisfying the axis test (specified as the second argument of an path operation) and to use this predicate for restricting the result of the content-determining expression (i.e. the second argument of the constructor). Below is the rewriting rule for \text{child} operation.

\[
\text{child}(\text{element}(e1, e2), \text{test}) \Rightarrow \\
\text{return}(e2, f(x|\text{if}(\text{predicate}, x(), ()))) & \quad (8)
\]

... In Rule 8, \text{predicate} is built by \text{test} that can be, as defined in Section 5, one of the following kinds \text{elem}(\text{name}), \text{attr}(\text{name}), or \text{text}(\text{name})

\[
\begin{align*}
\text{elem}(\text{name}) & \Rightarrow \text{name}(x) = \text{"name"} \quad \text{and} \quad \text{node-kind}(x) = \text{"element"} \\
\text{text}(\text{name}) & \Rightarrow \text{node-kind}(x) = \text{"text"}
\end{align*}
\]
The rewriting rule for the descendant operation can be defined as a recursive application of Rule 8.

descendant(element(e1,e2), test) =>
seq(return(e2,f(x) if(predicate,x,()))),
  (9)
  descendant(e2,test))

Before going on to the next group of rules, we give the reader informal explanation on how the rules helps to implement the required kinds of query rewriting. Formal consideration of the matter will be given at the end of this section. In order to implement the required kinds of query rewriting we have to rewrite the query to a form where accessors and predicates are not computed over to the results of element constructors. Because predicate is a logical expression defined via logical, arithmetical and other operations, we can state more general requirement to the result of the rewriting: non-meta-operations (i.e. accessors, logical operations, arithmetical operations, and so on except meta-operations) are not computed over the values of element constructors. Meta-operations are excluded from the operation list because of their property to only control the evaluation not contributing to the production of new values. The requirement is fulfilled when the following points are satisfied: (1) the expressions that present the arguments of the operations (argument expression for short) do not contain element constructors; (2) the expressions, the results of which are bound to variables during query evaluation, do not contain element constructors. The application of the rules listed above guarantees that the first point is completely satisfied but the second point is satisfied only with respect to inner path operations (i.e. the argument expressions of inner path operations do not contain element constructors except in the argument expressions of non-meta-operations (i.e. logical, arithmetical, accessor, etc.). The next group of rules (together with those defined above) guarantees that the second point is satisfied with respect to all non-meta-operations from basic subset but not only inner path operations.

The fourth group of rules eliminates element constructors from the argument expressions of accessors, operations over sequences of atomic values, arity-based operations over sequences, and EBV-based operations. According to the classification given in Section 5 the argument of all these operations can be break down into three categories: (1) to which the procedure of effective boolean value is applied; (2) to which the procedure of atomization is applied; (3) which are arguments of accessors. Rewriting rules are defined below for each of the categories. The rule names are EBV, ATOMIZE, and ACCESSOR, respectively. Informally speaking, the main idea behind these rules is to traverse the argument expression of an operation passing through meta-operations until an element constructor is encountered and replace the constructor with some other expression depending on the category into which the operation falls. The rules are specified as recursive functions. All these functions take two parameters and return the rewritten expression. The first parameter is an expression to which procedure is applied. The second parameter is a procedure state. The state is used to inform whether the function processes an argument expression of some non-meta-operation (in this case state is "active") and, especially in ACCESSOR function, what is the name of the non-meta-operation (in this case state is an operation name). The functions are specified in a pseudo language. This language consists of two operators SWITCH and MATCHES. The semantics of SWITCH is the same as that of traditional programming languages. MATCHES tests whether an expression matches a pattern defined in terms of the logical representation.

The first rule eliminates element from the subexpressions to which the procedure of effective boolean value defined in [2] is applied. The following function called EBV implements this rule. EBV is applied to a query with the second parameter STATE equal to "passive".

FUNCTION EBV(expr, STATE)
BEGIN
SWITCH
CASE expr MATCHES return(e1,f(x | e2)) :
  return(e1, f(x | EBV(e2,STATE)))
CASE expr MATCHES seq(e1,...,en) :
  seq(EBV(e1,STATE),...,EBV(en,STATE))
CASE expr MATCHES if(e1,e2,e3) :
  if(EBV(e1,"active"),EBV(e2,STATE),EBV(e3,STATE))
CASE expr MATCHES ts(e1,fx)
  cases(casest(t1|e1),...,casest(tn|en),default(en+1))) :
    ts(e1,fx) cases(caset(t1|EBV(e1,STATE)),...,case(tn|EBV(en,STATE)),
      default(EBV(en+1,STATE))))
CASE expr MATCHES ELEMENT(e1,e2) : true
CASE expr MATCHES and(e1,e2) :
  and(EBV(e1,"active"),EBV(e2,"active"))
CASE expr MATCHES or(e1,e2) :
  or(EBV(e1,"active"),EBV(e2,"active"))
CASE expr MATCHES not(e) : not(EBV(e,"active"))
CASE expr MATCHES some(e1,f(x | e2)) :
  some(e1,f(x | EBV(e2,"active")))
CASE expr MATCHES every(e1,f(x | e2)) :
  every(e1,f(x | EBV(e2, "active")))
DEFAULT /*expr MATCHES any-op(e1,...,en)*/
any-op(EBV(e1, "passive"),...,EBV(en, "passive"))
END SWITCH
END

The second rule eliminates element from the subexpressions to which the procedure of atomization defined in [2] is applied. The following function called ATOMIZE implements this rule. ATOMIZE is applied to a query with the second parameter STATE equal to "passive".
The rewriting gets complicated by reason that the expression is not equivalent to this one.

```
return(e,f(x|name-e))
```

According to the type definition of the `name` argument, there must be a dynamic error when the `return` expression is evaluated in non-singleton sequence. The latter expression does not arise any error in this case. It is a consequence of the strict typing feature of XQuery. To adhere to the semantics of strict typing, the initial expression should be rewritten as follows.

```
extended-name(
    return(e,f(x|cast-to-marked-name-type(name-e))))
```

cast-to-marked-name-type casts the value of `name-exp` to QName type (dynamic error is arise if the value cannot be cast to the type)⁴ and then to the special type called marked-name. extended-name operation extends `name` accessor as follows: in case its argument value is of marked-name type, it cast the value back to QName type; it operates as `name` accessor in other cases. Exploiting the approach discribed allows us to eliminate element constructors from the argument expression of `name` accessor and get the equivalent expression as the result of the rewriting.

### An example of rules application

To give a reader to understand how the rules of BRS work we consider the application of BRS to our motivating example given in Section 2. The expression of the query in terms of the logical representation is as follows.

```
return(  
  elem(bookstore),  
  f($bs)  
    element(bookstore,  
      seq(child($bs, elem(name)),  
        child($b,elem(authors)),  
        child($b,elem(pub-year)),  
        return(child($b, elem(book)),  
          f($b)  
            element(book,  
              seq(child($b, elem(title)),  
                child($b,elem(authors)),  
                child($b,elem(price)),  
                child($b,elem(price))  
                  * 0.5))))))))
```

Let us first focus on rewriting Subexpression 1. In this subexpression the path operation, namely outermost `child`, is applied to the result of transformation expressed by the construction of the XML element named `bookstore`. Rewriting of the subexpression will demonstrate the implementation of projection of transformation. This subexpression can be rewritten sequentially applying Rule 2 with `E` taken to `child(?,e)` and Rule 8 into the following expression.

⁴This type is used in XQuery to present names of XML elements.
The transformation has been projected by means of statically computing the child operation. As a result of this bookstore is no longer constructed in the rewritten query. But the expression can be simplified further by sequentially applying Rule 1 with $E$ taken to be $\text{return(?,e)}$ and Rule 2 with the same expansion of $E$. The result is as follows.

```
return(
    child(child (doc ("db.xml"), elem(stores)),
    elem(bookstore)),
    f($b|
    return(seq(child($b, elem(name)),
        return(child($b, elem(book)),
        f($v|if(node-kind($v)="element"
            and name($v)="book",$v,()))),
        return(child($b, elem(book)),
        f($b|
            if(
            node-kind(element(book,
            seq(child($b, elem(title)),
            child($b, elem(pub-year)),
            child($b, elem(authors)),
            element(price, child($b, elem(price)) * 0.5))))="element"
            and
            name(element(book,
            seq(child($b, elem(title)),
            child($b, elem(pub-year)),
            child($b, elem(authors)),
            element(price, child($b, elem(price)) * 0.5)))="book",
            $v,()))))))
```

Applying ACCESSOR to the above expression, we get $\text{cast-to-marked-node-kind("element")="element" and } \text{cast-to-marked-name("book")="book"}$ instead of Subexpression 2 and 3, respectively. Now we can apply static computation to simplify the expression being rewritten. Both latter subexpressions can be statically computed according to the definition of $\text{cast-to-marked-node-kind, cast-to-marked-name}$ and $\text{= operations into true}$. As a result, the if-statement condition containing Subexpressions 2 and 3 gets equal to $\text{true(})$ and $\text{true()}$. The latter expression can also be statically computed into true. It allows us to replace the if-statement with its first branch. We can also simplify the subexpression $\text{return(child($b, elem(name)),}
    f($v|if(node-kind($v)="element" and name($v)="book",$v,())))$ using type inference based optimization proposed in [8] as follows. Applying type inference we find out that the type of $\text{$v$ is element(name)*}$ that allows computing the expression $\text{name($v$)="book"}$ into false, the if-expression into $\text{$v$}$ and, therefore, the whole return expression into the empty sequence $\text{()}$. The result of the ACCESSOR application and the simplification is as follows.

```
return(
    child(child (doc ("db.xml"), elem(stores)),
    elem(bookstore)),
    f($b|
    return(child($b, elem(book)),
    f($b|
        if(child($b, elem(title)) = "Seven Years in Tibet",
        element(book,
        seq(child($b, elem(title)),
        child($b, elem(pub-year)),
        child($b, elem(authors)),
        element(price, child($b, elem(price)) * 0.5))))),
    )))
```

Assuming Subexpression (1) of the original expression having been rewritten as described above we can sequentially apply to it Rule 2 with $E$ taken to be $\text{return(?,e)}$, Rule 2 with $E$ taken to be $\text{child(?,e)}$, Rule 8, Rule 1 with $E$ taken to be $\text{child(?,e)}$, Rule 7 with $E$ taken to be $\text{return(?,e)}$, ACCESSOR, and some simple static computation in the same manner to push the predicate $\text{child($b$,elem(title)) = "Seven Years in Tibet"}$ down the constructor of the book element. In the result we get the following expression in which the book elements are constructed only for those books that are titled Seven Years in Tibet. The XQuery representation of the expression is given in Section 2.

```
return(
    child(child (child (doc ("db.xml"), elem(stores)),
    elem(bookstore)), elem(book))
    f($book|
    if(child($book, elem(title)) = "Seven Years in Tibet",
    element(book,
    seq(child($book, elem(title)),
    child($book, elem(pub-year)),
    child($book, elem(authors)),
    element(price, child($book, elem(price)) * 0.5)),
    ))))
```

**Formal Statements**

In this subsection we give formal consideration of BRS. Because the proofs of the following theorems are rather long to be presented in this paper, we refer the reader to the extended version of the paper [12].

**Theorem 1**

$\text{BRS has the property of a normal form.}$
As discussed in Section 4 normal form property ensures the correctness of the rewriting system: the process of rule application to any query representation will not be infinite and results in some normal form.

**Theorem 2**

In a normal form only meta operations and element can be applied to the results of element and only in positions marked by ?:

\[
\text{return}(e, f(x|?)), \quad \text{seq}(? , \ldots , ?), \quad \text{if}(e, ?, ?), \quad \text{ts}(e, f(x|\text{cases}(\text{case}(t_1, ?), \ldots , \text{case}(t_n, ?), \text{def}(e))), \text{element}(e, ?)).
\]

Theorem 2 proves that meta operations and element have the following very important property. The value of the marked parameters is not analyzed by these operations and just returned without any modification but maybe as a part of some new structure (e.g. when sequence is concerned). This leads to the following corollaries.

1. All projections are performed by BRS. Assume that it is not so. It follows that there is an XML element constructed during the computation that does not present in the result. But it is a contrary to the statement of the theorem.

2. All predicates are pushed down XML element constructors by BRS. Let us assume that it is not so. It means that some predicate in the query is applied to an item that is an XML element or contains an XML element as its part. To check whether the item satisfies the predicate, the content of the item has to be analyzed. But it is a contrary to the statement of the theorem.

These corollaries prove that BRS implements the required kinds of query rewriting.

**7 Optimization rules for the extension of basic subset to position-based operations**

Extending the set of basic operations with position-based operations (see Section 5 for the definition of position-based operations) leads to the two problems.

The first problem is that the results of the position-based operations depend on the context in which the operation is called. The context is implicitly generated and passed as implicit argument to position-based operations. This breaks the referential transparency property of XQuery (that tends to be functional). The rewriting of a query with position-based operation may result in non-equivalent query because the operation can change the position in the query and leave its context. To solve the problem context generation operations are introduced in this section. The results of the context generation operations are bound explicitly to iterator variables (i.e. the formal parameters of the function-argument of return) and all position-based operations are replaced with the variables. The rewriting rules for basic operations are extended to handle queries with context generation operations. Though using context generation operations allows one to rewrite many queries with position-based operations there is a number of query examples (with the last() operation) for which the required kinds of query rewriting cannot be accomplished. This is the second problem. We will show that this is a language-inherent problem and it cannot be solved by developing any rewriting system.

To extend the logical representation defined in Section 5 with context generation operations we have to add a new structure to the XML Data Model [1]. We refer to this structure as unit. Unit is an ordered set of items. Arity of an unit is a number of items in this unit. Unit allows combining items (defined in the XML Data Model) into one structure and constructing sequences of such structures. We use square brackets to denote an unit. For example, [a, b] is an unit of two items a and b.

Just as in the XML data model an item is equal to the sequence of one object that is this item, we identify item with the unit of one object that is this item. This allows us to consider all operations of XQuery closed on sequences of units of one item.

Using the notion unit, context generation operations can be defined. Context generation operations conceptually differ from the others in having an internal state that can be shared between several occurrences of a context generation operation in the query representation. Internal state presents a variable of type integer. The sharing of internal state is implemented by passing the name of an internal state variable as an argument to a context generation operation. All internal state variables are initialized to zero before the query is evaluated. During query evaluation, internal state variables are reinitialized to zero according to the following rules. Before a function-argument of the operations return, ts, some, every is evaluated, all internal state variables that occur in the function body are set to zero. The use of internal state variables will help us to implement the required kinds of query rewriting for many queries with the occurrences of position-based operations. Context generation operations (con-gen for short) are defined as follows.

**con-gen1(e, m)** - where e is a sequence of items and m is the name of some internal state variable - iterates over items of sequence e replacing each item with unit of the form [this item, m+1] and m is set to m+1.

**con-gen2(e, m)** - where e is a sequence of items and m is the name of some internal state variable - iterates over items of sequence e replacing each item with unit of the form [this item, m+1, count(e)] and m is set to m+1.
As mentioned in Section 5, position-based operations can be used only within the predicate of the XPath expressions. Let us define the rules of translation from an XPath expression with position-based operations into the logical representation using context generation operations. The application of the operations into the logical representation using conflation from an XPath expression with position-based XPath expressions. Let us define the rules of translations can be used only within the predicate of the position-based operation \( \text{last()} \).

In the first case the translation rule is as follows.

\[
\{e1/test[e2]\} =
\text{return (e1, f(v)}
\]

\[
\text{return(con-gen1(child (v, \{\{text\}\}), m),}
\]

\[
\text{e2'=\{e2\}\{item->x, position()->1\}}
\]

In the second case the translation rule is as follows.

\[
\{e1/test[e2]\} =
\text{return (e1, f(v)}
\]

\[
\text{return(con-gen2(child (v, \{\{test\}\}), m),}
\]

\[
f(x, i, \text{iff(e2', e, ()())}),
\]

\[
e2'=\{e2\}\{item->x, position()->1, last()->c\}
\]

The occurrence of context generation operations in a query representation can prevent from applying some of the rewriting rules specified in Section 6. To determine these rules, let us notice that according to the translation rules listed above the only position of context generation operation occurrence is that marked by \(?\) in \(\text{return(?}, f)\). Consequently we have to revisit only those rewriting rules in which the first argument of return is analyzed. These rules are 1, 2, 3, 4, 5, 6, 7 with \(E\) taken to be \(\text{return(?}, e)\). Some of these rules can be modified to define additional ones that accomplish the same kinds of query rewriting even in case of the occurrence of context generation operations. The additional rules are listed below. The rules that are valid for \(\text{con-gen1}\) are also valid for \(\text{con-gen2}\).

\[
\text{return(con-gen2(e1, m), f(x, i, c|e2))}
\]

\[
(\text{var?}(e1)) \Rightarrow \text{ (10)}
\]

\[
e2'=e2|\{e2\}\{item->x, i->1, c->1\}
\]

\[
\text{return(con-gen1(seq(e1, ..., en), m), f}) \Rightarrow \text{ (11)}
\]

\[
\text{seq(return(con-gen1(e1, m), f),}
\]

\[
\text{...}, \text{return(con-gen1(en, m), f))}
\]

Notice that the rule above is correct only for \(\text{con-gen1}\) and is not correct for \(\text{con-gen2}\).

\[
\text{return(con-gen1(return(e, f(x1| e1))),}
\]

\[
f(x2|e2)) \Rightarrow \text{ (12)}
\]

\[
\text{return(e, f(neu-var|}
\]

\[
\text{return(con-gen1(e1', m), f(x2|e2)))),}
\]

\[
e1'=e1|x->\text{neu-var}
\]

Notice that the rule above is valid only for \(\text{con-gen1}\) and is not valid for \(\text{con-gen2}\).

\[
\text{return(con-gen2(element(e1, e2), m),}
\]

\[
f(x, i, c|e3)) \Rightarrow \text{ (13)}
\]

\[
e3|x->\text{element(e1, e2), i->1, c->1}
\]

\[
\text{return(con-gen2(if(c, e1, e2), m),}
\]

\[
f(x, i, l|e3))\Rightarrow \text{ (14)}
\]

\[
\text{if(c, return(con-gen2(e1), f(x, i, l|e3)),}
\]

\[
\text{return(con-gen2(e2), f(x, i, l|e3)))}
\]

\[
\text{return(con-gen1}
\]

\[
\text{(ts(e, f(x1)cases(case(t1,e1),}
\]

\[
\text{...},
\]

\[
\text{case(tn, en),}
\]

\[
\text{def(en+1))))), m),}
\]

\[
f(x, i, l|e3)) \Rightarrow \text{ (15)}
\]

\[
ts(e, f(\text{neu-varcases(}
\]

\[
case(t1, return(con-gen1(e1'), m), f(x, i, l|e3))),
\]

\[
\text{...}
\]

\[
\text{case(tn, return(con-gen1(en+1'), m), f(x, i, l|e3))}))
\]

\[
\text{e1'=e1|x->\text{neu-var}}
\]

As we have mentioned above Rules 11, 12 are not valid with \(\text{con-gen1}\) replaced with \(\text{con-gen2}\). In this case if the second argument of the "outer" return contains a predicate, it will not be pushed down \(\text{seq}\) and the "inner" \(\text{return}\). It does not allow us to implement the required kinds of query rewriting in this case. It is easy to see that this problem cannot be solved by developing any rewriting system. This is the case because the occurrence of \(\text{con-gen2}\) means that to evaluate the predicate, one has first to evaluate \(\text{seq}\) or the "inner" \(\text{return}\). That is why the predicate cannot be pushed down \(\text{seq}\) or the "inner" \(\text{return}\).

8 Problems with rewriting identity-based operations and outer path operations

Extending the set of basic operations with identity-based operations (such as for example \(\text{union}\)) prevents from accomplishing the required kinds of query rewriting. For example, let us consider the following expression \(\text{return(union(e1, ..., en), f(x | if(pred, element(...), (),)))}\). If \(\text{seq}\) was instead of \(\text{union}\) we would apply Rule 1 with \(E\) taken to be \(\text{return(e, f(x|e1)}\) that is defined in Section 6 to push \(\text{pred}\) down \(\text{seq(return(e1, f(x | if(pred , element(...), (),)))}, ...
\]

\[
\text{return(en, f(x | if(pred, element ..., ()))))}\). In case of \(\text{union}\) this rule is not valid because of the following reason. \(\text{union}\) removes duplicates by unique identity and if such a rule is applied, \(\text{element}\) that regenerates unique identity will be applied to duplicates before \(\text{union}\) and the result of the rewritten expression may contain more elements then the original one.

The main problem with outer path operations is as follows. In contract to rewriting expressions with inner path operations, in case of outer path operations the analysis of the expression is not sufficient to make optimization decision. For example, despite we know that
the constructed element has no parent, we may not rewrite the expression \( \text{parent}(\text{element}(\text{name2}, e2)) \) into the empty sequence, because this expression might be the result of rewriting of the following one \( \text{parent}(\text{child}(\text{element}(\text{name1}, \text{element}(\text{name2}, e2)), \text{elem}(\text{name2}))) \) that is obviously not equal to the empty sequence.

We have no complete solutions for rewriting expressions with identity-based and outer path operations. Our partial solutions to the problem seems to be too cumbersome for practical implementation. The research on solutions for the problem is the subject of future work but we do not believe that any reasonable solution can be found.

9 Conclusion

In our previous papers [11, 13] we have proposed a general framework for XQuery rewriting-based optimization and stated promising kinds of rewriting-based optimization. The focus of this paper is optimization techniques for transformational queries: push predicates down XML element constructors and projection of transformation. The main contribution of this paper is a complete set of rewriting rules that fully implements these techniques.

All optimization techniques described in this paper have been fully implemented as an optimizer of XML-based virtual integration server called BizQuery [6]. The optimizer is implemented in Scheme. The size of implementation is about 6500 lines of code. We have conducted a lot of experiments with queries evaluated over more than 1Gb of data with and without optimization. The results of the experiments have shown that the proposed techniques improve query performance by orders of magnitude in a large number of common cases and the overheads incurred due to the query transformations are negligible compared with the time to execute complex queries. Unfortunately the limits on the size of the paper do not allow us to provide measures for practical queries because the size of such queries is too big to be presented in the paper. The measures for toy queries are not quite illustrative.

The focus of future work is the other kinds of rewriting-based optimization considered in [11]. These kinds are as follows: push predicate down iterative operations, simplify query on the basis of data schema, make query representation as declarative as possible, inline user-defined XQuery functions.

References


