A Method for XQuery Transform Implementation Based on Shadow Mechanism

Maria Rekouts, Maxim Grinev, Alexander Boldakov
Institute for System Programming of Russian Academy of Sciences, Russia

Abstract

The XQuery Update Facility has been recently published by the World Wide Web Consortium as a working draft. Among other features, the working draft presents a novel powerful transform expression. A transform expression creates a new modified copy of an XML subtree, where the modifications are specified as update operations. In this paper we investigate the use of transform expressions and provide a method to support transform expressions efficiently: the method avoids copying of the whole modified XML subtree. Our method is based on the idea of shadow mechanism proposed in early work on recovery.

1 Introduction

Over the past several years, both the research and the standards communities have been showing a great interest in extending XQuery [1] with additional advanced facilities. One of the most discussed and, obviously the one XQuery lacked most, is the XQuery Update Facility that has been recently presented by the World Wide Web Consortium as a working draft [3].

The XQuery Update Facility classifies all XQuery expressions into the following two mutually exclusive categories: an updating expression and a non-updating expression. An updating expression is an expression that can modify the state of existing nodes, while a non-updating expression preserves original contents. The XQuery Update Facility presents five new kinds of expressions, called: *insert*, *delete*, *replace*, *rename*, and *transform*. The first four of these are updating expressions, and the last (transform) is a non-updating expression. All of the four updating expressions are quite customary to XQuery users as they all had appeared in one form or another both in research proposals [4, 5], and in software products that support XQuery [6, 7, 8]. While the *transform* expression provides novel powerful XQuery semantics, it has not been studied enough by researches and is not present in today's software products.

A transform expression creates a new modified copy of a node (and its descendants) and allows the user to access both modified and unmodified nodes. This modification is specified as an update operation. Generally speaking, the user can obtain "modified" data without modifying the data itself in a single query. It is worth pointing out, that there is an alternative for transform expressions in XQuery 1.0: the same query can by written by means of recursive function that constructs the modified copies of existing nodes. Comparing the two approaches, transform expression provides the following advantages:

- A query with a transform expression is much easier to write, debug and understand in contrast to a query with a recursive function (in Section 1.2 we give an example of such queries).
- With transform expressions it is possible to avoid bulk copying of data. That is, due to its higher declarative form, transform expressions give implementors the possibility to implement the expression in different ways, in particular in a way that does not carry out real copying of data, or at least minimizes the copying. Otherwise, a query with recursive XML tree traversal and explicit construction of new elements implies straightforward way of execution that leads to copying of data. Optimization of such a query is very challenging, as it is quite a hard task to determine which of the newly created elements are modified and which are copied.

Obviously, supporting transform expressions without performing the actual copying can tremendously increase the speed of its evaluation and save memory resources. This is what the method presented in our paper aims to achieve.

1.1 Outline

In the next section we present a motivating example that demonstrates the effectiveness of using transform expressions for content engineering tasks. In Section 2 we provide our method to support transforms without copying and illustrate it by examples. In Section 3 related work is reviewed. Finally, Section 4 concludes the paper.

1.2 Motivating Example

Our motivating example comes from the field of content engineering [9]. The goal of content engineering is to automate content creation, management and publishing tasks usually performed by hand. One of the basic ideas is *single sourcing*, where there is a single version of content in the content engineering system, and the system provides a set of mechanisms that publish the content to various media. *Identity based markup* technique contributes to single sourcing support by solving the problems with links in the content: instead of explicit references, the content fragments are "labeled" using names that reflect and clarify the meaning of the content [9].

Consider a project where the goal is to create a collection of movie reviews and to publish it to various media: web page and CD-ROM. When published on a web page, each director name in the reviews must reference the director's biography in the internet movie database. When published to a CD-ROM, each director name must reference his biography stored locally on the CD-ROM. To accomplish this task, we use the following identity based markup technique: initially each director in the movie reviews is labeled with `director` tag, that contains this director's identificator as attribute; the information relative to each director is stored in a separate document 'directors.xml' and contains the URI to the biography in the internet movie database and a reference to a local file with biography.

Consider the example structure of the XML documents. In 'reviews.xml' every review node contains review metadata and the text of the review in XHTML format with incorporated `director` tags:

```
<review>
<title>Titanic</title>
<genre>romance</genre>
<text>
...
<p><director id="James Cameron">James Cameron's</director>
194-minute, $200 million film of the tragic voyage is in
the tradition of the great Hollywood epics.</p>
...
</text>
<review>
```

In 'directors.xml' each `director` node is labeled with `id` attribute and contains biography element with different forms of biography references for various media. The `url` element contains the following subelements: the URI to the director's biography intended for publishing on the web page, the `file` element with a path to the biography data to publish on the CD-ROM and the `text` element with brief biography description intended to publish on the printed media.

```
...
<director id="James Cameron">
<biography>
<url>http://en.wikipedia.org/wiki/James_Cameron</url>
<file>/biography/james_cameron.html</file>
<text>
James Francis Cameron (born August 16, 1954)
     is a Canadian-born American film director
     noted for his action/science fiction films,
which are often extremely successful financially...
</text>
</biography>
</director>
...
```

The task is to generate an XHTML version of review text replacing all `director` elements with the hyperlinks according to the directors' data from directors.xml. In the general case we do not know the exact structure of a `review` element, that is, we do not know in advance where the `director` elements will appear within the `review` element. Thus it is necessary to scan over a hierarchy of elements, applying the required transformation at each level of the hierarchy. In XQuery 1.0 this can be accomplished by defining a recursive function. An example of such a function performing the transformation looks as follows:

```
declare function local:traverse-replace($n as node())
as node()
{
typeswitch($n)
case $d as element(director)
return
let $b :=
document("directors.xml")/directors
          /director[@id=$d/@id]/biography
return
<a href="{$b/url/text()}">{$d/text()}</a>
case $e as element()
return element
{ fn:local-name($e) }
{ for $c in $e/(* | @* | text())
return local:traverse-replace($c) }
case $d as document-node()
return document
{ for $c in $d/* return local:traverse-replace($c) }
default return $n
};

for $r in doc("reviews.xml")/reviews
                /review[genre="romantic"]
return
local:traverse-replace($r)
```

In the above query the recursive function 'traverse-replace' scans over the whole data and returns the appropriate result based on the type of the current node. This function reconstructs the whole `review` nodes copying all data. Note that `director` elements constitute a relatively small part of the `review` elements. It means that the recursive function constructs the result preserving most of the data in its original form and transforming only small pieces of it. This leads to an extra overhead incurred by unnecessary copying of data. Optimization of such a query is not trivial and often practically impossible, as it is quite a hard task to determine which of the newly created elements are modified and which are copied. Another disadvantage of transformation queries written via recursive functions is their cumbersome syntax. The XQuery code looses its compactness and becomes hard to understand.

The transform expressions introduced in the XQuery Update Facility allows one to write the required transformation in much more compact way. Moreover, it gives more chances to implement it in an efficient way. The same transformation query written with transform expression looks as follows.

```
for $r in doc("reviews.xml")//review[genre="romance"]
return
transform
copy $rom:=$r
do
for $d in $rom//director
do replace {$d}
with
<a href="{doc('directors.xml')/director[id=$d/id]//url}">
      $d/text()
</a>
return $rom
```

In the above query `for` clause binds the XQuery variable `$r` to a sequence of `review` elements of romantic genre. According to the XQuery Update Facility working draft the semantics of the transform expression for each `review` node are as follows. The `copy` clause produces the copy of the sequence of the source `review` nodes. This copied node sequence consists of the same `review` nodes as original, but with different node identities. The `do` clause contains the updating expression applied to the newly created nodes at the previous step. The update expression replaces all occurrences of the `director` elements with the corresponding hyperlinks. According to the specification, `director` elements are called *target nodes*, and hiperlink elements are called *replacement nodes*. The last `return` clause constructs the result, i.e. in our case just returns modified copies of `review` elements.

2 A Method to Implement Transform Expressions Without Copying

In this section we present our method to implement transform expressions without copying of the data. The method has to address the following challenges caused by the fact that the data is not actually copied:

- In a query that contains a transform expression, the user can access both the source and the copied value. Logically, the source value is not modified by the transform expression, while the copied value is modified. Since we do not physically copy the source value into the copied value, our method must represent the same nodes both as the source value and as the copied value depending on how these values are accessed. In particular, the source value nodes and the copied value nodes must be distinguished in order to support node comparison operations correctly.
- While navigating over the copied value, our method must make sure that the navigation does not leave the bounds of the XML tree that represents the copied value. More precisely, the method must ensure that *parent*, *ancestor*, *following* and *preceding* axes do not lead out of the root of this XML tree.
- While navigating over the copied value, the replacement nodes must be visible instead of the target nodes.

Our method is based on the idea of *data shadow mechanism*, which was first proposed in the early papers on recovery and data versioning where there is an ability to return to original versions [10]. In particular, shadow mechanism was used in early versions of System R [11]. The basic idea of the data shadow mechanism is simple: when the source data is updated, the system does not overwrite the existing data, but stores new (replacement, insertion or deletion) nodes somewhere else on the disk, establishing a correspondence between the existing nodes and the new nodes. Since the old data remains unchanged, there are two versions of the data: the new version and the old version that becomes the *"shadow"* for the new one.

Our method resolves the challenges discussed above and extends the data shadow mechanism by providing the possibility to access both the new and the shadow version of the modified data in the scope of a single query. This is in contrast to the data shadow mechanism used for recovery provided the access only to the new version and the possibility to return to the shadow version.

Now let us proceed to the detailed description. First we present the basic assumptions our method relies on, then we describe the data organization used in our method, and at last define how these data structures must be processed within an XQuery executor to support transform expressions without copying.

2.1 Basic Assumptions

- We assume that at the physical level all nodes of the XML tree are linked via pointers. A pointer to a node is an implementation-dependent object, and its definition is out of the scope of our paper. However, the majority of XQuery implementations operate with pointers to nodes presented in one form or another, for example [6], [13], [14].
- We also assume that at the physical level an XQuery executor operates with pointers to nodes. That is, a query plan operation takes a sequence of pointers as input and produces a sequence of pointers as the result.
- For each transform operation in a single query the executor generates a unique transform identifier (UTI). We use such identifier to mark a node pointer.
- Every update expression can be rewritten into an equivalent replace expression [12]. Thus, without loss

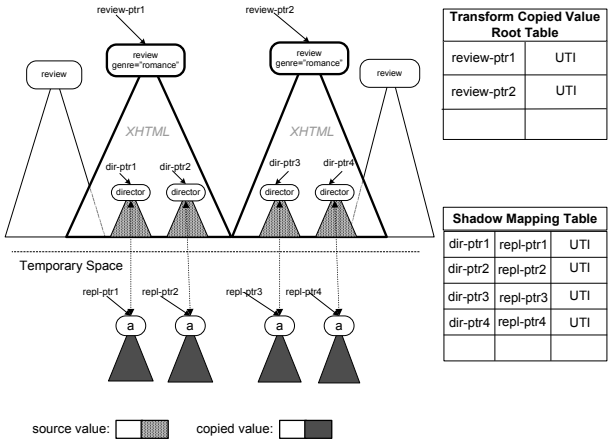

Figure 1. Data organization needed to support transform expressions without copying.

of generality, we present our method for transform expressions where modifications are specified as replace expressions.

2.2 Data Organization

Figure 1 demonstrates the data organization that we use for our method for the example data and transform expression given in Section 1.2.

In Figure 1, the reviews (XML trees with the `review` root node) that are copied in our example transform expression are typeset in bold. *review-ptr1* and *review-ptr2* are the pointers to these `review` root nodes. The review XML trees are painted white. Each 'review' XML tree contains two `director` nodes. These `director` nodes are replaced with the new `a` nodes in the example, and thus, are called the *target nodes*. Target subtrees spanned by the target nodes are painted grey. *dir-ptr1*, *dir-ptr2*, *dir-ptr3* and *dir-ptr4* are pointers to the `director` nodes. The new `a` nodes, the `director` nodes must be replaced with, are built in the temporary space, and are called the *replacement nodes*, the replacement subtrees are painted black. *repl-ptr1*, *repl-ptr2*, *repl-ptr3* and *repl-ptr4* are pointers to the replacement nodes.

As we can see, review XML trees are not physically copied. Thus, the review XML tree nodes (the white tree) are accessed both as the source value, and as the copied value.

The union of all the white and grey tree nodes constitutes the complete source value. The union of the white and black tree nodes constitutes the complete copied value. As the target subtrees remain unchanged and the new replacement subtrees are stored separately, the target subtrees become the shadows for the replacement subtrees.

Our method uses the following two tables: *Transform Copied Value Root Table* (TCVRT) registers all the pointers to the copied value root nodes of all of the transform expressions in the considered statement (each transform expression is identified by the UTI). Thus, for our example TCVRT contains *review-ptr1* and *review-ptr2* with the UTI. *Shadow Mapping Table* (SMT) provides the correspondence between the shadow nodes (the target nodes) and the new nodes (the replacement nodes).

2.3 Execution

Before we describe the execution of transform expressions we present a normal form for general transform expressions where the copied value is returned as a transform result.

```
transform                          let $i:=transform
copy $copied:=source-expr            copy $copied:=source-expr
do update-expr($copied)      =>      do update-expr($copied)
return expression($copied)           return $copied
return expression($copied)
```

The XQuery executor processes queries in a standard fashion [2], however the following extra steps are taken in order to process queries that contain transform expressions:

1. The normalized transform expression is processed as follows:
   - The XQuery executor generates a UTI for each transform operation in a query.
   - The XQuery executor evaluates `source-expr`, obtaining a sequence of pointers as a result of the evaluation. Every pointer of this result sequence is marked with a corresponding UTI.
   - Every pointer of the `source-expr` result sequence is associated with the UTI and the record of the form [pointer, UTI] is added to the TCVRT table.
   - The XQuery executor evaluates target nodes of the `update-expr`. For each target node it creates a sequence of replacement nodes of the `update-expr` in the temporary space. Parent, following-sibling and preceding-sibling pointers of the replacement nodes are set as if they were constructed by the usual update operation, while the target nodes and their parent, following-sibling and preceding-sibling nodes remain unchanged.
   - Records of the form: [target node pointer, its corresponding replacement node pointer, UTI] are added to the SMT table.

2. All XQuery query plan operations can be divided into two groups: operations of the first group construct new items and return pointers to these new items (for example: element constructors, arithmetical operations, logical operations); operations of the second group do not construct new items and return pointers to the existing items, derived from the input pointers (for example: XPath operations, conditional operations such as `if` and `type-switch`, sequences concatenation). For the operations of the second group we introduce *counterparts* that operate on the marked pointers as follows: the result pointers that were derived from marked pointers are also marked by this counterpart operation. In the query plan all operations of the second group are replaced with their counterparts.

3. A new query plan operation `check` is introduced. This operation implements pointer mapping according to the SMT table as follows: `check` takes a sequence of pointers as input, for every pointer `check` looks up this pointer in the target-pointer column of the SMT table, and if it is found returns the corresponding replacement pointer, if it not found, returns the pointer itself. In the query plan `check` operation is applied to the result of every counterpart operation.

4. For node comparison operations (namely `is`, `<<` and `>>`) we introduce their counterparts as follows:

   - if one of the operands of the `is` operation is marked while the other one is unmarked, the operation returns false; in other cases it operates as usual;
   - if the right operand of `<<` is marked and the left one is unmarked, the operation returns false; if the right operand is unmarked and the left one is marked, the operation returns true; if the right operand of `>>` is marked and the left one is unmarked, the operation returns true; if the right operand is unmarked and the left one is marked, the operation returns false.

5. For the operations `parent`, `preceding-sibling` and `following-sibling` that represent the corresponding axes, we introduce their counterparts as follows: for the input pointers that are marked and have their entry in the TCVRT table, the operation returns an empty sequence, otherwise it operates as stated in the 2. We assume, the `ancestor` operation is implemented by means of the `parent` operation, and the `following` and `preceding` operations by means of the `ancestor` operation. Thus, `ancestor`, `following` and `preceding` also operate according to the TCVRT table.

2.4 Method Exemplification

In this section we illustrate the essence of our method by the example queries. The example queries are based on the transform expression presented in Section 1.2, and we explain how the result of the transform expression is further processed. For short, we indicate the transform expression from the motivating example as `{transform-review}`.

The first example demonstrates the navigation over the result of the transform expression, which is a logically copied value, and shows the necessity of `check` operations:

```
{transform-review}/text//*/@href
```

According to our method the query plan shown in Figure 2 (the left one) is generated. In this query plan `check` operations denoted in ovals return their original input sequence without changes. The `check` operation denoted in rhomb substitutes its input pointers to `director` nodes with the pointers to the corresponding `a` nodes. The next example demonstrates the necessity of introducing counterpart operations:

```
({transform-review}, doc('reviews.xml')/reviews/
                review)/*/../..
```

In this query a sequence concatenation operation produces a sequence of both marked and unmarked pointers. The `parent` operation denoted in rhomb operates according to the TCVRT table. For marked input pointers (which are the copied-value roots) it returns an empty sequence. The query plan for this query is shown in Figure 2 (the right one).

Both query plans may be optimized in order to remove unnecessary `check` operations.

3 Related Work

As we mentioned above, the main challenge in implementing transformations via updates consists in providing an efficient way to access both the original and the modified states of the data. The topic of transforming data via updates and related implementation issues have received little attention so far. Implementation methods that were proposed in the literature put some restrictions on accessing the data in at least one of the states.

In a paper devoted to transaction management in the Orion object-oriented system [15], the authors proposed a special type of transactions called hypothetical transactions as a method to support data transformations via updates. In contrast to normal transactions, which can be committed or aborted, hypothetical transactions always abort. Since the changes are never recorded permanently, the user has the freedom of performing updates with the only goal to query the modified state of data. In order to minimize

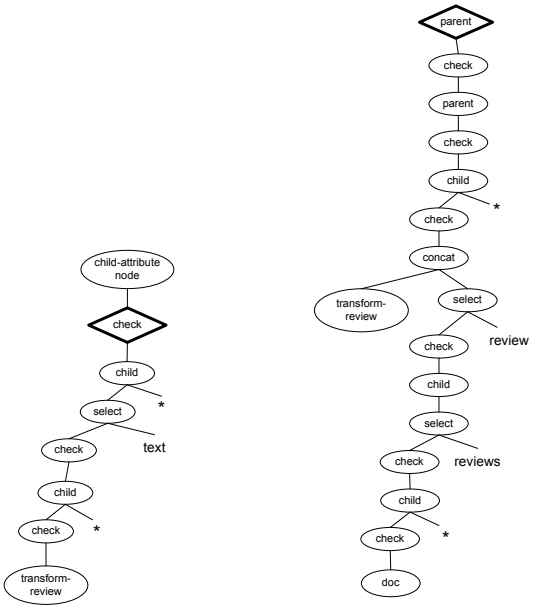

Figure 2. Example query plans

the overhead incurred by abort operation, the authors used shadow mechanism. Because of using shadow mechanism the source values are never updated but in this method they have not been made accessible in the query.

In our previous work [12] we introduced a method to implement transform expressions which is based on serialization mechanism. The method allows obtaining both modified and source values but with the restriction that the modified value can only be processed as a string value.

4 Conclusion

This paper presents a method to support XQuery transform expressions without copying. Our method is based on ideas of shadow mechanism proposed in early papers on recovery. The method is being prototyped in the Sedna XML database system [6].

References

[1] Boag, S., Chamberlin, D., Fernandez, M., Florescu, D., Robie, J., Simeon, J.: XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, (November 2005)

[2] Draper, D., Fankhauser, P., Fernandez, M., Malhotra, A., Rose, K., Rys, M., Simeon, J., Wadler, P.: XQuery 1.0 and XPath 2.0 Formal Semantics. http://www.w3.org/TR/xquery-semantics/, (November 2005)

[3] Chamberlin, D., Florescu, D., Robie, J.: XQuery Update Facility.

http://www.w3.org/TR/xqupdate/, (January 2006)

[4] Benedikt, M., Bonifati, A., Flesca, S., Vyas, A.: Adding Updates to XQuery: Semantics, Optimization, and Static Analysis. Second International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P 2005).

[5] Tatarinov, I., Ives, Z., Halevy, A., Weld, D.: Updating XML. Proceedings of the ACM SIGMOD international conference on Management of data (2001)

[6] Fomichev, A., Grinev, M., Kuznetsov, S.: Sedna: A Native XML DBMS. Proceedings of the Conference on Current Trends in Theory and Practice of Informatics (SOFSEM) (2006)

[7] Mark Logic's xqzone: http://xqzone.marklogic.com/

[8] XML Support in Microsoft SQL Server 2005.

http://msdn.microsoft.com/xml/default.aspx?pull=/library/en-us/dnsql90/html/sql2k5xml.asp

[9] Stilo Whitepapers: Content Engineering.

http://www.stilo.com/download/stilowhitepapers.html

[10] Raymod A. Lorie: Physical Integrity in a Large Segmented Database. ACM Transactions on Database Systems, Vol.2, No. 1, (1977) 91–104

[11] Jim Gray, Paul McJohns, Raymond Lorie et al.: The Recovery Manager of the System R Database Manager. Computing Surveys, Vol. 13, No. 2, (1981)

[12] Boldakov, A., Grinev, M.: Transforming XML Data Using Side-Effect Free Updates. Programming and Computer Software, Vol. 32, No. 2, (2006)

[13] Meng, X., et al.: OrientX: A Schema-based Native XML Database System. Proceedings of the VLDB, (2003), 1057–1060

[14] Fiebig, T., Helmer, S., Kanne, C.-C., Moerkotte, G., Neumann, J., Schiele, R., Westmann, T.: Anatomy of a native XML base management system. The VLDB Journal, Vol. 11, No. 4, (2002)

[15] Jorge F. Garza, Won Kim: Transaction management in an object-oriented database system. Proceedings of the ACM SIGMOD international conference on Management of data (1988)