Projection based optimization for XML updates

Nicole Bidoit and Dario Colazzo and Noor Malla and Marina Sahakyan
Université de Paris-Sud, Laboratoire de Recherche en Informatique, Bâtiment 490, F-91405 Orsay Cedex, France

Abstract. XML projection is one of the main adopted optimization techniques for reducing memory consumption in XQuery in-memory engines. The main idea behind this technique is quite simple: given a query $Q$ over an XML document $D$, instead of evaluating $Q$ on $D$, the query $Q$ is evaluated on a smaller document $D'$ obtained from $D$ by pruning out, at loading-time, parts of $D$ that are irrelevant for $Q$. The actual queried document $D'$ is a projection of the original one, and is often much smaller than $D$ due to the fact that queries tend to be quite selective in general. While projection techniques have been extensively investigated for XML querying, we are not aware of applications to XML updating. The purpose of the paper is to investigate a projection based optimization mechanism for updates.

1 Introduction

XML projection is one of the main adopted optimization techniques for reducing memory consumption in XQuery in-memory engines. The main idea behind this technique is quite simple: given a query $Q$ over an XML document $D$, instead of evaluating $Q$ on $D$, the query $Q$ is evaluated on a smaller document $D'$ obtained from $D$ by pruning out, at loading-time, parts of $D$ that are irrelevant for $Q$. The actual queried document $D'$ is a projection of the original one, and is often much smaller than $D$ due to the fact that queries tend to be quite selective in general.

In order to determine an optimal projection $D'$ several approaches exist [7, 8, 11, 12], and most of them are based on query path extraction: all the paths occurring in $Q$, and expressing the real data-needs for the query, are first extracted and then used to build the projection $D'$. In particular, the type based approach [7] assumes that queried data are typed by a DTD, and uses extracted paths to determine, by means of type inference, what are the type names of the elements really needed for the query; this set of type names is dubbed type-projector. Once a type-projector has been inferred, building the projection $D'$ is a quite efficient and simple operation: $D$ is visited according to document order, by a buffer-less SAX parser, and only elements whose label types are in the type projector are kept in the projection $D'$.

While projection techniques have been extensively investigated for XML querying, we are not aware of any applications to XML updating. At a first glance, such an extension seems to issue at least two challenges: i) a new path extraction mechanism has to be devised in order to deal with update operations, and ii) a technique has to be found in order to make updates persistent. Solving these two challenges, would allow sensible optimizations in terms of memory (and even time) consumption for several in-memory XML querying engines like, just to mention some of them, Galax [3], Saxon [5], Quizx/open [4], eXist [2], and MonetDB/XQuery [1]. All of them share a common modality to perform updates: the input document is first loaded in main memory, then updated, and finally stored back on the disk. As a consequence, each one of these systems have some limitations on the maximal size of documents that can be processed.
We checked that for eXist, Quizx/open and Saxon it is not possible to update documents whose size is greater than 150 MB (no matter the update query at hand) with standard settings and memory limitations.

This article focuses on the issue ii) previously discussed, and on the presentation of the experimental results obtained by an implementation of an update optimization method based on type-projection. Due to space limitation, the method outlined below is not presented in detail. In our projection based update method, for a query update \( U \), the document \( D \) is assumed to be typed by a DTD and the projection \( D' \) of \( D \) is built using a type-projector \( \pi \) inferred by a projector inference system, as proposed in [7]. The assumption here is that \( \pi \) is a sound projector for \( U \) and that the projection \( D' \) aims at keeping in memory only parts of the document \( D \) relevant for evaluating the query update \( U \). The next step of the update mechanism consists of applying the query update \( U \) over the projection \( D' \) building a new document \( U(D') \). As opposed to what happens for a simple query, the document \( U(D') \) is not the final expected result \( U(D) \). In general, they are distinct documents: in particular, all the sub-trees pruned out during the projection phase are obviously missing in \( U(D') \), while they are present in the expected result \( U(D) \). Hence the problem to be solved is how can we efficiently produce \( U(D) \) starting from \( U(D') \). To this end we have defined and implemented a merging algorithm able to produce \( U(D) \) starting from \( U(D') \) and the projector used to obtain \( D' \).

The new technique can be used with any in-memory engine, since it does not require any change in the internal algorithms of the engine itself, nor it requires query re-writing. To make some preliminary tests, we have implemented the proposed projection and merging algorithm in Java. We have considered the popular and largely optimized Saxon system [5] to run some update queries over several XMark documents of growing size. This framework forces us to wait until the partial result \( U(D') \) is stored on second memory storage which is of course not necessary (see future work in last section). Even under such an environment, our evaluation tests show that not only memory consumption is noteworthy optimized, but also total execution time is sometimes drastically reduced.

The article is organized as follows. Our update query scenario is introduced in Section 2 through a motivating example. Section 3 is devoted to the presentation of the evaluation tests based on a first implementation of our method. We conclude by discussing some related works and further research directions in Section 4.

2 Motivating examples and discussion

This section is devoted to introducing and illustrating the update scenario through examples, as well as some of the choices and assumptions made in the formal presentation. Indeed, the second part of this section focuses on the features of the update type projector. Recall, once again, that, the purpose of the paper is to present the experimental results obtained from the implementation of the method.

The update scenario through an example Let us consider the DTD \( d \) specified by the following regular expressions:

\[
\text{doc } \rightarrow \text{ a+ } \\
\text{a } \rightarrow \text{ b?,c*,d+ } \\
\text{b, d } \rightarrow \text{ (b | f)+ } \\
\text{c, f } \rightarrow \text{ String?}
\]

Let us consider the following update query \( U \):

for \( $x \) in /doc/a where \( $x/b \)  
return (delete nodes \( $x/c/text() \), rename node \( $x/b \) as \( 'c' \))
Consider the document \( t \) of Figure 1.a on which the update \( U \) should be applied. As already outlined in the introduction, the update will be performed by first pruning the document \( t \) to keep, in memory, a sub-document of \( t \), as small as possible but of course “sufficient” for evaluating the update. Let us now explain how we proceed.

First, we assume, that the document element nodes are labelled by their location identifiers as depicted by document \( t^\lambda \) in Figure 1.c. The location identifiers are inserted as subscript of the node labels in the picture. Note that this step is virtual and only considered for the purpose of the explanation (in practice \( t^\lambda \) is not materialized and location identifiers are computed on the fly).

The adorned document \( t^\lambda \) is then pruned by projection with respect to a type-projector derived from the update query \( U \) and from the DTD \( d \). As shown in [7], in order to determine a type-projector for a query, here for an update, an important preliminary and basic operation is required: determining the type of nodes used and returned by the query/update. This type inference is made by using the paths extracted by the query.

Indeed, for the current update \( U \), the extraction of the type-projector is the same as for the following pure query \( Q \):

```latex
\begin{verbatim}
for $x$ in /doc/a where $x/b$ return <res> $x/c <res>
\end{verbatim}
```

For this query, first the paths /doc/a/b and /doc/a/c//node() are extracted. Note that in the second one the //node() is added, indicating that for building the result we need all the descendants of c nodes. Then, the type of nodes traversed by these paths are inferred, thus obtaining the type projector \( \tau=\{doc, a, b, c, String\} \). During projection of the input document, only nodes of these types will be retained.

The projection \( t^\lambda \) of the document \( t^\lambda \) with respect to the type projector \( \tau \) is shown in Figure 1.d. The update query \( U \) is then evaluated over the projected document, producing a partial result \( U(t^\lambda|\pi) \) (see Figure 1.e). It is important to note here that the update query performed over the projected document is the original update query \( U \): no rewriting of \( U \) is required.

The updated partial document is of course not what is expected as the final result. The last step of the update scenario is dedicated to building the final updated document \( U(t) \). In order to do this, the adorned document \( t^\lambda \) is merged in a streaming fashion with the, in memory, updated partial document \( U(t^\lambda|\pi) \). In other words, thanks to the location identifiers, both documents are parsed in a synchronized manner: (i) elements in \( t^\lambda \) (for instance the \( d \) elements for our current example) that have been pruned, before the partial update, are recovered during the merge phase, in the right order; (ii) elements in \( t^\lambda \) that have been projected are output in the result with the changes (rename, delete) made by the partial update registered in the document \( U(t^\lambda|\pi) \), in the right order.

**Discussion** In the previous motivating example, for the purpose of illustrating the overall update mechanism, we chose a simple update query leading to a quite obvious type projector. Indeed, the method that has been implemented is based on a refinement of this notion of type projector. First, type projection has been refined in order to handle in an optimized manner string node projection. Secondly, recall that, in our framework, the projected document is updated and then the updated partial document is adequately *merged* with the original input in order to propagate updates. As expected, we soon realized that the more we prune during projection, the more difficult and complex the merge phase is. So, in some cases, we choose to prune less in order to simplify the merge algorithm. Given a DTD and an update query, we distinguish a set of *critical labels* corresponding to the types of nodes for which we require to keep all children.
Fig. 1. Update scenario: motivating example
The critical update operations are i) insert and replace updates, plus ii) all those updates potentially touching mixed-content nodes.

3 Experiments

In order to validate the effectiveness of our method, we have implemented it in Java, and performed several tests by using 7 update queries on XMark documents of growing size. These seven queries cover most of main update operations made available by XQuery Update Facility (insert, rename, replace and delete). Queries used for experiments are indicated below.

1. Insert a new annotation node for each closed_auction not containing any children tagged as annotation:
   
   for $x$ in doc("xmark.xml")/site/closed_auctions/closed_auction
   where not ($x/annotation)
   return (insert node <annotation>Empty Annotation</annotation>
      as last into $x)

2. Rename all phone nodes with personal_phone:
   
   for $x$ in doc("xmark.xml")/site/people/person/phone
   return (rename node $x$ as "personal_phone")

3. Replace each address node where country equals to United States with a new one which has a new city and country:
   
   for $x$ in doc("xmark.xml")/site/people/person/address
   where $x/country/text()="United States"
   return (replace node $x$ with
      <address>
        <street>{$x/street/text()}</street>
        <city>"NewYork"</city>
        <country>"USA"</country>
        <province>{$x/province/text()}</province>
        <zipcode>{$x/zipcode/text()}</zipcode>
      </address>)

4. Replace each location item whose value is United States with USA:
   
   for $x$ in doc("xmark.xml")/site/regions//item/location
   where $x/text()="United States"
   return (replace value of node $x$ with "USA")

5. Delete all mail items in all regions (africa, asia, australia, europe, namesrica, numberica):
   
   delete nodes doc("xmark.xml")/site/regions//item/mailbox/mail

6. Rename each bold child of text node with emph:
   
   for $x$ in doc("xmark.xml")/site//text/bold
   return (rename node $x$ as "emph")
7. Insert a new homepage node with www.{$x/name/text()}Page.com for each person which does not contain an homepage child:

```
for $x in doc("xmark.xml")/site/people/person
    where not($x/homepage)
    return (insert node
        <homepage>www.{$x/name/text()}Page.com</homepage>
    after $x/emailaddress)
```

To run the update queries, we used Saxon SA 9.1.0.6 with 512 MB for the Java heap memory, on a Intel Centrino 2.00GHz laptop with 1 GB of RAM, and running Windows XP. The size of XMark documents considered goes from 10 MB to 2 GB.

We chose Saxon for performing preliminary tests because it is fully XQuery Update Facility compliant, because it easy to install and use, and does not make massive use of indexes (the available main-memory is mostly used for the updated data). In any case, we plan to use other systems in future developments of this work.

Under these settings and without projection, Saxon was not able to update documents whose size is greater than 150 MB, even if most of the queries used for tests are quite selective. On the other hand, thanks to the size reduction ensured by projection, we were able to update documents with size up to 2 GB. The only exception is query Q6, for which our projection-based technique was not able to update documents of size greater than 250 MB; this is due to the fact that this query needs a very large part of the original document (because of a large set of critical labels).

The tables below provide our test results on, respectively, the time needed for updating original documents (Table 1), the size of projected documents (Table 2), and the total time needed to update documents in our framework (Table 3). We do not report tables for pruning and merging time. According to the tests, it turned out that pruning and merging time are about 50% and 40% of the total time, respectively.

These tables, clearly show that our technique succeeds in its primarily purpose: updating very large documents with in-memory systems, in the presence of memory limitations. More extensive tests are needed. Indeed, we made some preliminary tests with eXist [2] and we obtained very similar and encouraging results. Increasing the JVM memory size has also been tried: we made several tests with 1 GB of heap memory and realized that in this case, up to 250 MB, documents can be processed without pruning, but then Saxon took about 20 minutes to terminate the update for the seven considered queries (probably due to intensive swapping). Fortunately, the tests performed reveal that if pruning is used in such cases, the execution time drastically reduces to no more than 4 minutes.

Even if time optimization is not the main purpose of this work, concerning execution time results in Table 3, it is worth observing that the reported values include the time needed to i) load, project and store -on disk- the input document, ii) to update the stored projected document (which includes in turn the time needed to store this partial result), and, finally, iii) the time needed for the merging phase.

Note that, values reported in this table say that execution times with and without projection are almost of the same order, proving that the pruning and merging phase are not too much time consuming. In any case, there are at least two steps that we envision to eliminate in future evolutions of this framework: storing the pruned document on the disk (by directly putting it in main memory for processing), and storing and re-read the partial update pruned document (which is in main memory at the end of the process, and which could be directly merged with the original document on the disk). These two improvements require some kind of strong interaction with the query engine,
and hence will require further implementation efforts; anyway, we realized that they
would probably lead to a reduction of about 50% of the time indicated now in Table 3.
This would imply, that even when projection is not necessary -on memory consumption
basis- to execute the update, using projection can reduce execution time as well.

As a final remark, we would like to stress that, the tests have been used to experi-
mentally check that our update evaluation scenario is correct: the results produced by
our method have been compared successfully with those obtained with Saxon.

<table>
<thead>
<tr>
<th>Table 1. Updating Time of Original Files (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Or. Size MB</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>52</td>
</tr>
<tr>
<td>104</td>
</tr>
<tr>
<td>128</td>
</tr>
<tr>
<td>148</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2. Size of Pruned Files in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Or. Size MB</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>52</td>
</tr>
<tr>
<td>104</td>
</tr>
<tr>
<td>128</td>
</tr>
<tr>
<td>148</td>
</tr>
<tr>
<td>250</td>
</tr>
<tr>
<td>312</td>
</tr>
<tr>
<td>1 GB</td>
</tr>
<tr>
<td>2 GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3. Total Time (Prune + Update + Merge) (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Or. Size MB</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>52</td>
</tr>
<tr>
<td>104</td>
</tr>
<tr>
<td>128</td>
</tr>
<tr>
<td>148</td>
</tr>
<tr>
<td>250</td>
</tr>
<tr>
<td>312</td>
</tr>
<tr>
<td>1 GB</td>
</tr>
<tr>
<td>2 GB</td>
</tr>
</tbody>
</table>
4 Related Works and Conclusions

We are not aware of any other approach using document projection for XML updates. Some other works propose techniques to optimize update query execution time by using static analysis in order to detect independence between several update operations, so that query rewriting techniques can be used for logical optimization [9, 10, 6]. Our work is definitely orthogonal wrt this line of research, and for this reason, fortunately, the two techniques can be combined in order to ensure efficiency in terms of both time and main-memory consumption.

We are currently working on several directions in order to complete and improve our update evaluation method.

Acknowledgments We thank the anonymous referees for valuable comments and constructive criticism. We would also like to thank Bazizi Mohamed Amine for his careful reading of the article and for very useful discussions. This work has been partially funded by Agence Nationale de la Recherche, decision ANR-08-DEFIS-004.

References