Optimisation de Mises à jour XML par typage et projection

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. La projection est une des techniques utilisées pour permettre de réduire les besoins en terme de taille mémoire nécessaire aux moteurs de requêtes XML "in-memory". L'idée sous-jacente à cette technique est simple : étant donnée une requête Q à évaluer sur un document XML D, au lieu de procéder au calcul des réponses de Q sur D, la requête Q est évaluée sur un document D', plus petit que D, obtenu lors du chargement de dernier en mémoire, par élagage des parties de D qui ne sont pas utiles pour Q. Le document Q' qui celui sur lequel la requête Q est calculée, est une projection du document initial Q. Il est souvent de taille drastiquement inférieure à celle de Q. Ceci est dû au fait que les requêtes sont en général très sélectives.
Alors que cette technique de projection a été étudiée et développée assez largement pour l'interrogation de document XML, à notre connaissance, ce type de technique n'a pas été explorée ni appliquée aux mises à jour de documents XML. L'objet de cet article est donc de proposer une technique d'optimisation de mises à jour de documents au format XML, exploitant le typage des documents.

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 unrelevant 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 unrelevant 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, 9, 15, 16], 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 [2], Saxon [4], QuizX/open [3], and eXist [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. For instance, 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 (we are quite confident that similar limitations hold for systems we have not tested yet).

In order to overcome these limitations we are actively working on both issues i) and ii). Concerning the first one, we have soon realized that sound projectors for updates U can be easily obtained by a simple extension of the path extraction algorithm in [7]. Once paths in U have been extracted, the existing type projector inference algorithm in [7] is used to extract a type projector for paths in U, and hence for U since its paths soundly approximate its semantics. Type-projectors obtained in this way are precise enough to ensure great benefits in terms of reduction of main memory consumptions for in-memory engines. Nevertheless, as discussed in the conclusion, we are currently and independently investigating extracting type projector for update expression for improving memory size and execution time requirements.

Due to space limitations and since propagating updates to secondary memory storage is a real new problem, we focus, in this article, on the second issue ii) previously mentioned. Thus we consider the following scenario. 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 π inferred by a projector inference system [7]. The assumption here is that π is a *sound* projector for U (we will characterize soundness), 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 happen 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 $U(D')$.

The main contribution of this paper for solving this problem is an algorithm, called *Merge* that takes as input the projector π, the input D and the partial updated document $U(D')$, and outputs the expected result $U(D)$. This algorithm works in a streaming fashion, and processes D and $U(D')$ in a parallel and a strictly synchronized way. Moreover, it uses a buffer whose size is upper bounded by the maximal depth of D. The use of the projector π and of some specific element labeling, are the two crucial ingredients allowing one to proceed in a streaming manner.

Another positive aspect of our framework lies on the fact that 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 rewriting. 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 [4] 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. Section 2 is devoted to presenting preliminary definitions and notations. Our update query scenario is introduced in Section 3 through a motivating example. Section 3 is also devoted to discussing type projector for update query again through examples. The formal presentation of type projector and of the overall update query mechanism is developed in Section 4 which fully details the merging process including implementation issues. Section 5 summarizes evaluation tests made based on a first implementation of our method. We conclude by discussing some related works and further research directions in Section 6.

2 Preliminaries

Data Model We use a data model which is essentially that used in [11, 7, 10], and defined by the following grammar.

Forest $f ::= () \quad | \quad f, f \quad | \quad t$ **Trees** $t ::= string \quad | \quad l[f]$

An XML data model instance is a forest f, consisting of an ordered sequence of labelled ordered XML *trees* (the empty forest is noted as ()). Each tree t is either an l labeled element $l[f]$ whose content is f, or a string base value *string*. For simplicity, we do not consider attributes[1].

Each sub-tree t of a forest f is associated to a unique identifier $\mathbf{i}$, ranging over N^*, with N being the set of naturals. The identifier $\mathbf{i}$ represents the location of the sub-tree t inside f. ϵ denotes the empty location, and $\mathbf{i} \cdot \mathbf{i}'$ the composition of two locations $\mathbf{i}$, $\mathbf{i}'$.

Definition 1 ($LT(f)$ – $f@\mathbf{i}$ – $loc(f)$ and $trees(f)$).
The set $LT(f)$ is a set of pairs $(\mathbf{i}, t)$ such that t is located at $\mathbf{i}$ inside f. This set is defined as follows.

$$LT(()) = \emptyset \qquad LT(l[f]) = \{(\epsilon, l[f])\} \cup LT(f)$$
$$LT(string) = \{(\epsilon, string)\} \qquad LT(t_1, \ldots, t_n) = \bigcup_1^n \{(i \cdot \mathbf{i}, t') \mid (\mathbf{i}, t') \in LT(t_i)\}$$

We write $f@\mathbf{i}$ for the only tree t such that $(\mathbf{i}, t) \in LT(f)$, if such a tree exists, otherwise $f@\mathbf{i}$ stands for the undefined value **undef**. *We also define $loc(f) = \{\mathbf{i} \mid (\mathbf{i}, t) \in LT(f)\}$ and $trees(f) = \{t \mid (\mathbf{i}, t) \in LT(f)\}$.*

The following definition is needed in order to introduce data projection.

Definition 2. *Given a forest f and a set of locations I, we denote as $f_{|I}$ the forest obtained from f by replacing each subtree $f@\mathbf{i}$, for $\mathbf{i} \in I \cap loc(f)$, with the empty forest ().*

Our notion of data projection is very similar to that of [7]:

Definition 3 (Projection). *Given two forests f and f' we say that f' is a* projection *of f if and only if $f' = f_{|I}$ for a set of locations I.*

We consider input XML trees valid wrt a schema defined by means of the DTD language, which is widely used in practice [8].

For simplicity of notation, we represent DTD as in [12]. So, given a finite set of tag symbols Σ, and the reserved symbol *String*, a DTD over Σ is a tuple $(d,\ s_d)$ where d is a total function from Σ to the set of regular expressions over $\Sigma \cup \{String\}$, and s_d is the root symbol (of course $s_d \in \Sigma$). In the following, given a regular expression r we write $L(r)$ and $S(r)$ to indicate, respectively, the language it generates, and the set of symbols in Σ it contains.

Example 1. The following DTD will be used in the paper as a running example. The starting symbol is *doc*.

doc → *a+* *a* → *b?,c*,d+*
c → *(String | b | f)** *d* → *(b | f)+* *b,f* → *String*

Note that the only *mixed-content* used in this DTD is that for the typing of c elements.

[1] The algorithm we are going to present can be trivially extended to handle them.

In the sequel we will need the following notation: given a tree t, we denote as $rlab(t)$ the label type of the root of t, so $rlab(t) = l$ if $t = l[f]$, while $rlab(t) = String$ if $t = String$.

A tree t is valid wrt d if and only if there exists f' such that $t = s_d[f']$ and for each $\mathbf{i} \in loc(t)$, if $t@\mathbf{i} = l[f]$ then :

- $f = () \Rightarrow \epsilon \in L(d(l))$;
- $f = t_1, \ldots, t_n \Rightarrow rlab(t_1) \cdot \ldots \cdot rlab(t_n) \in L(d(l))$.

We will write $t \in d$ to indicate that t is valid wrt to d.

Update query language As already said, the update language considered here is XQuery Update Facility [5], as it is the main current proposal for XML updates. The only XQuery Update Facility update mechanism we do not consider is *transform* , which we discard for simplicity of treatment, and which we will consider in subsequent stages of this work.

3 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, although type-projector is an important component of our update mechanism, the purpose of the paper is not to present how such projectors are extracted from updates.

The update scenario through an example Let us consider the DTD d specified by the following regular expressions (it is a variant of the DTD of Example 1):

doc → *a+* *a* → *b?,c*,d+* *b, d* → *(b | f)+* *c, f* → *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, in order to keep in main-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^λ 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^λ is not materialized and location identifiers are computed on the fly).

The adorned document t^λ 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:

```
for $x in /doc/a where $x/b return <res> $x/c <res>
```

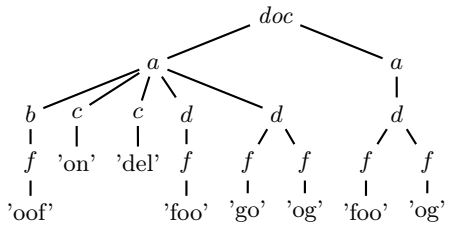
For this query, first the paths `/doc/a/b` and `/doc/a/c//node()` are extracted. Note that in the second one the `//node()` is added, indicating that for building the result we need all the descendants of `c` nodes. Then, the type of nodes traversed by these paths are inferred, thus obtaining the type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

The projection t^{λ}_{τ} of the document t^{λ} with respect to the type projector τ 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.

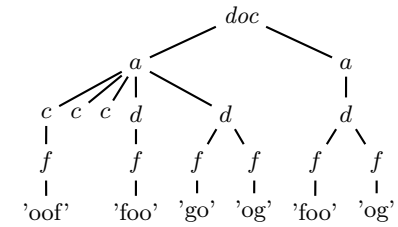
The updated partial document is of course not what is expected as the final result. The last step of the update scenario is dedicated to building the final updated document $U(t)$. In order to do this, the adorned document t^{λ} is merged in a streaming fashion with the, in memory, updated partial document $U({t^{\lambda}}_{|\pi})$. In other words, thanks to the location identifiers, both documents are parsed in a synchronized manner: (i) elements in t^{λ} (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^{λ} 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.

A two level type projector for updates 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. The aim of the following discussion is to motivate and introduce through examples the main features of type projectors for update. Next, the reader should pay attention to the fact that we consider the DTD of Example 1.

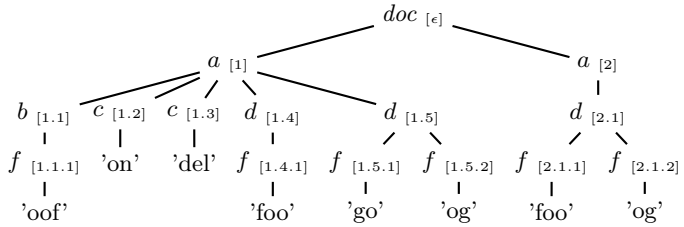
1. String node projection The first point of the discussion is indeed related to update as well as to pure query. So, let us go back to considering the pure query Q previously given and the type projector τ = {*doc, a, b, c, String*} for this query. Although the DTD is different, this type projector is sound. It is able to ensure sensible size reductions, however it is not minimal due to the presence of the String type. The problem comes from the fact that, to ensure efficiency, projection is performed in streaming, by visiting the document according to document order (in a SAX like fashion) and by retaining nodes whose types are in the projector. This implies that when the string children of a b node are met, with the b node a child of an a node, these text nodes are kept in the



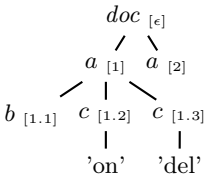
(a) XML document t



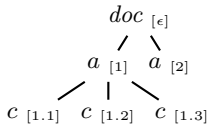
(b) Updated XML document $U(t)$



(c) XML document t adorned with location identifiers



(d) t projected with respect to τ



(e) Partial Update of t wrt τ and U

Fig. 1. Update scenario : motivating example

projection (since String is in the type projector), although they are not needed for evaluating the query Q .

To overcome this problem we adopt the following solution: we consider a *two-level type projector* $\pi = (\tau, \kappa)$ where τ contains all the needed node types but the String type, and κ is a subset of τ containing the types of nodes whose string children are to be kept in the projection, because needed by the query. Of course, using two-level type projectors requires a slight change in the projection process: when textual nodes are encountered during the streaming visit, these ones are retained if and only if their father type is in κ.

For the current example, the two-level projector is (τ, κ) with τ=$\{doc,\ a,\ b,\ c\}$ and κ=$\{c\}$. It enables to project only the necessary string values, which are, for the example, the textual children of c nodes.

The remaining discussion shows that the use of two-level projectors is further motivated by other needs. 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 will distinguish a set of *critical labels* corresponding to the types of nodes for which we require to keep all children. We will collect critical labels in κ and we will require that if $l \in \kappa$ then all children labels of l are in τ .

The *critical* update operations are i) insert and replace updates, plus ii) all those updates potentially touching mixed-content nodes.

2. Insert-into updates Consider the following update U_1:[2]

```
for $x in /doc/a where $x/b
return   for $y in $x/d
         return   insert node <f>some text</f> as last into $y
```

One could think that a minimal and correct projector for U_1 is the following one $\pi_1 = (\{doc, a, b, d\}, \emptyset)$, since the children of d nodes are not apparently touched and needed by the query. According to this choice, after the document has been projected, we would insert the new f element in empty d elements, since in the projected document we do not keep any children of d nodes.

In this case, during the merge phase we would need to recover all the pruned children of d nodes, and put them *before* the new inserted node. Ensuring this is quite hard if none of the children of d nodes is kept, since the merge process is not assumed to use the query update, and only relies on the original and partial updated documents. In other words, the information at hands during merging is not sufficient to recover the order of the d children, the old ones w.r.t. the new one. To solve this problem, we adopt a solution ensuring a good compromise between simplicity and efficiency. This solution consists of keeping in κ all the types touched by the *target* expressions of such insert-into operations; for the

[2] We use the *as last* option here, but the same considerations hold for the other kinds of insert-into updates.

above example we keep d in κ. According to what was previously stated, this implies that all labels (b and f) of children of d are in τ as well. For the current example, the type projector is then specified by $\pi'_1 = (\{doc, a, b, d, f\}, \{d\})$. The projection generated by π'_1 ensures that when a new node is inserted into a d node, in the projected document, it will have all the siblings it should have in the final result with the correct ordering. This entails that the merging phase in this case is immediate: the children of d to be output in the final result are exactly the one occurring in the partial updated document (the process is recursive and we are talking here about the top level elements which are children of a d element).

For the same reasons, the above considerations hold for updates containing the `insert-before/after` or `replace` operators. In the case of `insert-before/after` updates, the critical labels can easily be obtained by first inferring label types for the target expression, and then by considering as critical the father[3] label of these ones (a context-based type system like that provided in [7] allows this kind of type inference, in a very precise way, even in the presence of labels with multiple fathers). For example, consider

```
for $x in /doc/a/d/f insert node <b/> after $x
```

Since the label type corresponding to the target expression is f, its father label d is considered as critical .

Replace updates are in some sense equivalent to a delete followed by an insert-into update.

3. Mixed-Content We now turn to illustrating a second type of situation which requires a careful treatement at the level of the projector in order to facilitate the merging phase. Thus here, we are concerned about elements having mixed-content children.

Consider the following update U_2:

```
for $x in /doc/a/c where $x/text()  return delete nodes $x/b
```

The above query deletes b children of each c element containing some text children. One could say that, in order to have a sound projector for this query only text and b children of c nodes must be retained, while f children can be pruned out. Unfortunately, it may happen, for example, that two text children of a c element are separated by one f element and thus during projection the two text nodes collapse into a unique text node. Moreover, the update performed on the projected document may further collapse text nodes. Of course this situation makes quite complex the merging phase, since it would be quite hard to split the collapsed text nodes back into the original ones, in order to produce the expected updated original document.

Therefore, we solve this problem by enforcing that, as soon as a mixed-content element (like the c element of our example) is touched by the update, every child node of this element (String, b and f children) must be projected. In this way no text node of the mixed content is erroneously collapsed with another node during projection or update processing. Concerning the current example, we will

[3] A label l is a father of a in the DTD d, if $d(l) = r$ and a is used in r.

consider the type projector $\pi'_2 = (\{doc, a, c, b, f\}, \{c\})$. This choice of projector entails a very easy merging phase, as in the case of insert-into update.

Of course, for insert-into like updates and mixed-content, alternative solutions could be adopted for optimizing the memory consumption, but for the moment we prefer to keep things simple, while ensuring at the same time strong improvements in terms of memory and time reduction during update execution, as shown by our test results reported later.

Before concluding the section, we would like to outline the fact that operations like delete and rename do not entail critical labels (of course this is true in the case that these operations do not touch mixed-contents). This is because the type of elements touched by their target expressions are in the projector, and the presence of these types (together with position labels) is sufficient for the merge phase.

4 Update scenario

The forthcoming presentation aims at describing, in an abstract manner, the update scenario introduced in the previous section. We will afterwards explain how the abstract update mechanism is implemented. This section is organized as follows. First, we define the update type projector, then we detail the steps of the update mechanism. This will allow us to address the notion of sound projector.

Update type projector First of all, we formally define projectors:

Definition 4 (Type Projector). *Given a DTD d over the alphabet Σ, a type projector is a pair (τ, κ) such that i) $\tau \subseteq \Sigma$, ii) for each $l \in \kappa$ we have $S(d(l)) \subseteq \tau$, iii) $s_d \in \tau$ and for each $s \in \tau$ there exists $l \in \tau$ such that $d(l) = r$ and s occurs in r.*

As previously mentioned, the above conditions have the following meaning: τ is the set of types (labels) of nodes that are to be kept in a projection of a tree $t \in d$, and κ contains types of nodes that are to be kept in the projection together with *all* their children, some of which may be String children. The last condition ensures that it is never the case that a node is kept into a projection without its father.

It is worth observing that deciding, during the projector inference, which labels have to be collected in κ requires to determine what are the types of nodes selected by the target expressions in the update. This can easily be achieved by adapting several well established type analysis techniques [11, 7, 10].

Definition 5 (Type Driven Projection). *Given a type projector $\pi = (\tau, \kappa)$ for d, and a tree $t \in d$. The π-projection of t is the projection $t_{I(t,\pi)}$ where*

$$I(t, \pi) = \{\mathbf{i} \mid t@\mathbf{i} = l[f] \ \wedge \ l \notin \tau\} \cup$$
$$\cup \{\mathbf{i}.i \mid \neg(t@\mathbf{i}.i = \mathit{string} \ \wedge \ t@\mathbf{i} = l[f] \ \wedge l \in \kappa)\}.$$

Location adornment We consider the document t^λ obtained from t by labelling its elements by their location identifiers. As already mentioned in Section 3, this step is virtual. The implementation subsection will further clarify this point. Formally, t^λ is such that:

- $loc(t) = loc(t^\lambda)$, and
- if $t@\mathbf{i}$=*string* then $t^\lambda@\mathbf{i} = string$, otherwise
  for any $\mathbf{i} \in loc(t)$, $rlab(t@\mathbf{i}) = l$ iff $rlab(t^\lambda@\mathbf{i}) = l_\mathbf{i}$

Trees (forests) enriched with location are called l-trees (l-forests). Note that, the notions of validity wrt a DTD d and of projection remain the same for l-trees (l-forests).

Update projector soundness The key idea for characterizing a sound update projector is to consider as a starting point the document t with location adornment, and to express that, over t^λ, projection and update U commute.

Definition 6 (Sound update projector). *A projector π is correct for the update query U over d if and only if for each $t \in d$, we have: i) π is well-formed with respect to critical updates (as defined in Definition 4 and ii) $U(t^\lambda{}_{|\pi}) = U(t^\lambda)_{|\pi}$.*

In the above definition, one has to pay attention to the fact that once again, when updating t^λ with U, the new elements inserted in t^λ are location-less and adorned with the dummy location $\perp$. The above definition tells that, when an update projector is sound, elements located "out" the location set $I(t, \pi) = I(t^\lambda, \pi)$ are not influential for the update: the query part of the update does not use these elements which are neither updated (touched by an insertion, a renaming, a replacement or a deletion).

The Merge function We are now ready to detail the merging process by defining the function *Merge*. This function takes as input the updated partial document $U(t^\lambda{}_{|\pi})$, the adorned document t^λ, the update projector π, and produces the final result $U(t)$. We have:

$$U(t) = Merge < t^\lambda > < U(t^\lambda{}_{|\pi}) >$$

(for the sake of simplicity, we keep π implicit in the notation of Merge.).

We will describe afterwards an efficient implementation of *Merge* that has been realized to proceed to some evaluation whose results are promising. The function *Merge* is formalized in Figure 2. For the sake of simplicity, we assume that the update projector $\pi = (\tau, \kappa)$ is implicit. The following notations are used: given a location identifier $\mathbf{i}$, we write $\mathbf{i}.j \rhd \mathbf{i}.i$ iff $j = i + 1$ and the closure[4] (resp. reflexive closure) of $\rhd$ is denoted by $\rhd^+$, resp. by $\rhd^*$.

[4] Intuitively, if $\mathbf{i}.j \rhd^* \mathbf{i}.i$ then the node at location $\mathbf{i}.j$ is a right sibbling of the node located at $\mathbf{i}.i$.

Let us now provide some comments concerning the way the *Merge* function proceeds. Basically, the function *Merge* parses in parallel the two l-forests F_λ and F_u. One has to keep in mind that F_λ is a sub-forest of the input well-formed l-tree t^λ and F_u is a sub-forest of the updated partial l-tree $U(t^\lambda{}_{|\pi})$ which is not necessarilly well-formed. Moreover, the following pre-condition holds wrt to π: (†) it is assumed that the parent node of F_u is not critical. The function *DMerge* takes care of forests having a critical parent node and will be presented afterwards. The way the function *Merge* merges the first tree $l_{\mathbf{i}}[f']$ of F_λ with the first tree $l'_{\mathbf{i}'}[f'_u]$ of F_u relies on the location identifiers $\mathbf{i}$ and $\mathbf{i}'$ and on the update projector $\pi = (\tau, \kappa)$ which indicates what are the elements to output in $U(t)$ and also how to synchronize the parsing of both forests. More precisely:

First of all, it should be highlighted here that, because of the pre-condition (†) and because update type projectors are well-formed wrt critical updates, none of the top level trees in F_λ is of type String.
Thus, when (see Line 2) the first subtree of the forest F_u is a `string`, this `string` has neither been projected nor updated, otherwise the father of this string node would have had a *critical* label and *DMerge* would take care of this case. As a consequence, this `string` can not participate to any kind of merging and has to be simply output.

Line 3 deals with the case where the label l of $l_{\mathbf{i}}[f']$ belongs to τ. Thus this tree has been projected but then deleted through U ($\mathbf{i}' \rhd^+ \mathbf{i}$ indicates that, roughly, the element $l'_{\mathbf{i}'}[f'_u]$ comes after the element $l_{\mathbf{i}}[f']$). Thus $l_{\mathbf{i}}[f']$ is not output in $U(t)$.

Line 4 deals with the case where the label l of $l_{\mathbf{i}}[f']$ is not in τ. This tree has not been projected because it was not a target of U, hence it has to be output in $U(t)$.

Lines 5 and 6 take care of the case where the parsing is synchronized over the "same" subtrees: the locations $\mathbf{i}$ and $\mathbf{i}'$ are equals. The labels l and l' may or may not be the same. In the latter case, this means that l has been renamed as l' at location $\mathbf{i}$ by U. In line 5, condition $l \in \tau - \kappa$ indicates that l_i is not a critical label wrt the update U and thus merging the sub-forest f' and f'_u is done by recursively applying *Merge*. In line 6, condition $l \in \kappa$ indicates that, this time, l is a critical label wrt the update U and in such a case, we know that all children, what ever the type, of element l have been projected for ensuring the update correctness. Merging the sub-forests f' and f'_u is then performed slightly differently by the function *DMerge* explained below.

The function *DMerge* proceeds in a way which is very similar to that of *Merge*, but the first differs from this last one due to the following facts: i) parsing F_λ and F_u in parallel is now guided by F_u and ii) cases where $l \notin \tau$ never happen. In a nutshell, the reason for i) is that, since the two sequences F_λ and F_u contains children of the *same* parent node (see case 6), and in F_λ nothing has been pruned out at the top-level (the parent node is critical), then what has to dominate the process of determining the final result is the new structure present in the sequence resulted by the update U, that is F_u. This fact is in turn at the basis

of the soundness of d.2, dealing with the case where the current node t_u on the right hand-side forest is either a text or a newly inserted element. In both cases this node has to be output. Note that this line can be applied in cases where the process may lose synchronization (there is no move on F_λ), if not already lost by previous *DMerge* steps on previous nodes of the two sequences. By a simple case analysis on the first element of F_λ, it is easy to see that synchronization is then recovered by other lines.

The reader can observe that line d.3 covers the same cases as line 3 although she should pay attention to the case when F_λ=*string*, f. In this case, the text node in F_λ is ignored because the *correspondent* text node in F_u, updated or not by U, has, eventually, already been output by a previous application of line d.2 (a simple case analysis can help to see this better in details). The case treated by lines d.4 and d.5 corresponds, respectively, to line 5 and 6 of the *Merge* definition. Finally, the line d.7 corresponds to the case where either every children of a critical node has been deleted, or the critical node already had no child before the update, and no new child has been inserted.

Implementation issues The main point is the way location adorment is handled. Although the l-tree t^λ is made explicit in the above formal scenario, its materialization is not required by the implementation. The adorned projection ${t^\lambda}_{|\pi}$ is directly generated while parsing and projecting the input document t. The location identifiers associated with nodes are introduced by means of a special new attribute, attached to each node. Note that the potential overhead of adding this attribute is mitigated by the size reduction ensured by projection. Recall that, newly inserted elements have no such special attribute, and this is crucial to recognize new elements during the merge phase.

The implementation of the *Merge* step does not need to materialize t^λ either. Indeed, the final updated document $U(t)$ is produced by parsing in parallel the input document t and the partial updated document $U({t^\lambda}_{|\pi})$: the location identifiers for t^λ used in the *Merge* case analysis (Figure 2) are re-computed on the fly.

5 Experiments

In order to validate the effectiveness of our framework, we have implemented both projection and merge algorithms 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; we also indicate for each query the τ and κ components of the type projector.

1. Insert a new `annotation` node for each `closed_auction` not containing any children tagged as `annotation`.

$Merge < F_\lambda > < F_u >=$

1	F_u	if $F_\lambda = ()$
		otherwise assume $F_\lambda = string, f$
2	$string, Merge < f > < F_u >$	
		otherwise assume $F_\lambda = l_{\mathbf{i}}[f'], f$
3	$Merge < f > < F_u >$	if $l \in \tau$ and either $F_u = ()$ or $F_u = {l'}_{\mathbf{i}'}[f'_u], f_u$ and $\mathbf{i}' \rhd^+ \mathbf{i}$
4	$l[f'], Merge < f > < F_u >$	if $l \notin \tau$ and either $F_u = ()$ or $F_u = {l'}_{\mathbf{i}'}[f'_u], f_u$ and $\mathbf{i}' \rhd^* \mathbf{i}$
5	$l'[Merge < f' > < f'_u >], Merge < f > < f_u >$	if $F_u = {l'}_{\mathbf{i}}[f'_u], f_u$ and $l \in \tau - \kappa$
6	$l'[DMerge < f' > < f'_u >], Merge < f > < f_u >$	if $F_u = {l'}_{\mathbf{i}}[f'_u], f_u$ and $l \in \kappa$

$DMerge < F_\lambda > < F_u >=$

$d.1$	F_u	if $F_\lambda = ()$
		otherwise assume $F_u = t_u, f_u$
$d.2$	$t_u, Merge < F_\lambda > < f_u >$	if $t_u = string$ or $t_u = {l'}_\perp[f'_u]$
		otherwise assume $F_u = t_u, f_u$ and $t_u = {l'}_{\mathbf{i}'}[f'_u]$
$d.3$	$DMerge < f > < F_u >$	if $F_\lambda = l_{\mathbf{i}}[f'], f$ with $\mathbf{i}' \rhd^+ \mathbf{i}$ or $F_\lambda = string, f$
$d.4$	$l'[Merge < f' > < f'_u >], DMerge < f > < f_u >$	if $F_\lambda = l_{\mathbf{i}'}[f'], f$ and $l \in \tau - \kappa$
$d.5$	$l'[DMerge < f' > < f'_u >], DMerge < f > < f_u >$	if $F_\lambda = l_{\mathbf{i}'}[f'], f$ and $l \in \kappa$
$d.6$	$l'[f'_u], DMerge < F_\lambda > < f_u >$	
		otherwise assume $F_u = ()$
$d.7$	$()$	

Fig. 2. Definition of *Merge* and *DMerge*

```
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)
```

$\tau_1 = \{$ site, closed_auctions, closed_auction, seller, buyer, itemref, price, date, quantity, type, annotation $\}$
$\kappa_1 = \{$ closed_auction $\}$

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")
```

$\tau_2 = \{$ site, people, person,phone $\}$
$\kappa_2 = \{\}$

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>)
```

$\tau_3 = \{$ site, people, person, name, emailaddress, phone, homepage, creditcard, profile, watches, address, street, city, country, province, zipcode $\}$
$\kappa_3 = \{$ person $\}$

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")
```

$\tau_4 = \{$ site, regions, africa, asia, australia, europe, namerica, samerica, item, location $\}$
$\kappa_4 = \{$ location $\}$

5. Delete all mail items in all regions (africa, asia, australia, europe, namerica, samerica):

```
delete nodes doc("xmark.xml")/site/regions//item/mailbox/mail
```

$\tau_5 = \{$ `site, regions, africa, asia, australia, europe, namerica, samerica, item, mailbox, mail` $\}$
$\kappa_5 = \{\}$

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")
```

$\tau_6 = \{$ `site, regions, africa, asia, australia, europe, namerica, samerica, item, description, parlist, listitem, text, bold, keyword, emph, mailbox, mail, closed_auctions, closed_auction, annotation` $\}$
$\kappa_6 = \{$ `text` $\}$

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)
```

$\tau_7 = \{$`site, people, person, name, emailaddress, phone, address, homepage, creditcard, profile, watches` $\}$
$\kappa_7 = \{$ `person` $\}$

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-meory 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 or equal to 148 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 [1] and we obtained very similar and encouraging results. Increaseing 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 experimentally check that our update evaluation scenario is correct: the results produced by our method have been compared successfully with those obtained with Saxon.

6 Related Works and Conclusions

We have already talked in the introduction about existing approaches investigating projection-based optimizations for XML querying. As already said, we are not aware of any other existing approach on the use of document projection for XML updates. Some other works propose techniques to optimize update

Table 1. Updating Time of Original Files (in seconds)

Or. Size MB	Q1	Q2	Q3	Q4	Q5	Q6	Q7
10	3.84	4.2	4.02	3.72	3.94	4.53	4.06
52	17.31	20.63	22.6	18.6	18	18.73	18.03
104	30.65	40.4	39.02	36.05	34.23	36.84	60.05
128	33.43	45.97	49.33	42.01	41.25	52.13	102.03
148	-	-	-	-	-	-	-

Table 2. Size of Pruned Files in MB

Or. Size MB	Q1	Q2	Q3	Q4	Q5	Q6	Q7
10	0.33	0.15	0.65	0.17	0.24	6.23	0.66
52	1.7	0.78	3.31	0.89	1.2	31.4	3.32
104	3.41	1.6	6.69	1.8	2.4	62.9	6.7
128	4.12	1.93	4.66	2.22	2.99	77.3	8.29
148	4.89	2.2	9.54	2.6	3.5	89.1	9.57
250	8.13	3.8	16.1	4.37	5.75	150	16.2
312	10.1	4.7	20.1	5.5	7.4	188	20.2
1 GB	38.9	18.3	77.7	21	28.3	718	78.3
2 GB	67.7	31.8	109	36.5	49.1	1239.04	135.8

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 [13, 14, 6]. Our work is definitely orthogonal w.r.t. 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 the here introduced framework. First of all, we are defining optimal type rules for type-projector inference, with the aim of reducing as much as possible the size of projected documents. Second, still to minimize size of projected documents, we envision to minimize the presence of critical labels by using some particular kind of query rewriting techniques. Third, of course, we have to formally prove correctness of the merge algorithm; here the purpose was mainly to present the algorithm and some preliminary and promising test results. Finally, we plan to enrich our evaluation test with additional and extensive tests, implying more complex queries and other query engines as well.

Acknowledgments We thank the anonymous referees for their 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.

Table 3. Total Time (Prune + Update + Merge) (in seconds)

Or. Size MB	Q1	Q2	Q3	Q4	Q5	Q6	Q7
10	6.06	5.98	7.2	6.38	6.47	7.3	9.71
52	25.76	25.61	26.18	25.54	25.55	32.13	41.12
104	49.23	49.75	49.63	50.73	49.75	71.24	97.18
128	61.46	62.41	61.55	62.51	62.68	83.38	136.08
148	68.95	69.85	71.16	70.06	69.81	97.58	187.58
250	116.38	119.23	121.27	119.2	113.71	192.31	281.39
312	144.59	145.38	151.38	145.05	143	-	351.29
1GB	550.78	550.74	614.98	538.92	543.01	-	1358.73
2GB	963.57	953.67	1042.48	943.01	917.67	-	2251.59

References

1. eXist. http://exist.sourceforge.net/xquery.html.
2. Galax. `http://www.galaxquery.org`.
3. Qizx/open. `http://www.xmlmind.com/qizx/qizxopen.shtml`.
4. Saxon-sa. `http://saxon.sourceforge.net/`.
5. Xquery update facility 1.0. `http://www.w3.org/TR/2008/CR-xquery-update-10-20080801`.
6. M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Verification of tree updates for optimization. In *CAV*, pages 379–393, 2005.
7. V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. Type-based XML projection. In *VLDB*, pages 271–282, 2006.
8. G. J. Bex, F. Neven, and J. V. den Bussche. DTDs versus XML schema: A practical study. In *WebDB*, pages 79–84, 2004.
9. S. Bressan, B. Catania, Z. Lacroix, Y.-G. Li, and A. Maddalena. Accelerating queries by pruning XML documents. *Data Knowl. Eng.*, 54(2):211–240, 2005.
10. J. Cheney. Flux: functional updates for XML. In *ICFP*, pages 3–14, 2008.
11. D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Static analysis for path correctness of XML queries. *J. Funct. Program.*, 16(4-5):621–661, 2006.
12. W. Gelade, W. Martens, and F. Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. In T. Schwentick and D. Suciu, editors, *ICDT*, volume 4353 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2007.
13. G. Ghelli, K. H. Rose, and J. Siméon. Commutativity analysis in XML update languages. In *ICDT*, pages 374–388, 2007.
14. G. Ghelli, K. H. Rose, and J. Siméon. Commutativity analysis for XML updates. *ACM Trans. Database Syst.*, 33(4), 2008.
15. A. Marian and J. Siméon. Projecting XML documents. In *VLDB '03*, pages 213–224, 2003.
16. M. Schmidt, S. Scherzinger, and C. Koch. Combined static and dynamic analysis for effective buffer minimization in streaming XQuery evaluation. In *ICDE*, pages 236–245, 2007.