Projection for XML Update Optimization.

Mohamed-Amine Baazizi  
Leo Team  
Univ. Paris-Sud - INRIA  
Saclay  
bazizi@lri.fr

Nicole Bidoit  
Leo Team  
Univ. Paris-Sud - INRIA  
Saclay  
bidoit@lri.fr

Dario Colazzo  
Leo Team  
Univ. Paris-Sud - INRIA  
Saclay  
colazzo@lri.fr

Noor Malla  
Leo Team  
Univ. Paris-Sud - INRIA  
Saclay  
Nour.Malla@lri.fr

Marina Sahakyan  
Leo Team  
Univ. Paris-Sud - INRIA  
Saclay  
Marina.Sahakyan@lri.fr

ABSTRACT

While projection techniques have been extensively investigated for XML querying, we are not aware of applications to XML updating. This paper investigates a projection based optimization mechanism for XQuery Update Facility expressions in the presence of a schema. This paper includes a formal development and study of the method as well as experiments testifying its effectiveness.

Categories and Subject Descriptors

H.2 [Database Management]: Systems—Query processing

1. INTRODUCTION

XML projection is a well-known optimization technique for reducing memory consumption of XQuery in-memory engines. The main idea behind this technique is quite simple: given a query $q$ over an XML document $t$, instead of evaluating $q$ over $t$, the query $q$ is evaluated on a smaller document $t'$ obtained from $t$ by pruning out, at loading-time, parts of $t$ that are not relevant for $q$. The queried document $t'$, a projection of the original one, is often much smaller than $t$ due to selectivity of queries.

In order to determine an optimal projection of $t$ several approaches exist [11, 12, 15, 19]. Most of them are based on query path extraction: all the paths expressing the data-needs for the query $q$ are first extracted and then used for projecting $t$. In particular, the type based approach [11] assumes that documents are typed by a DTD and combines path extraction with type inference, to determine the type names (labels) of the elements required for the query. This set of type names is dubbed type-projector, and used at loading time to prune out elements whose type labels do not belong to it.

While projection techniques have been extensively investigated for XML querying, we are not aware of any application to XML updating, although several XML querying engines like Galax [2], Saxon [7], QizX [5, 4], and eXist [1] perform updates in main-memory: 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 has some limitations on the maximal size of documents that can be processed. For instance, we checked that for eXist, QizX/open [5] 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.

XML projection, as described above, cannot be applied directly for updating XML documents. Obviously, updating a projection of a document $t$ is not equivalent to updating the document $t$ itself: the pruned out sub-trees will be missing.

In this paper, we develop a type based optimization technique for updates. Our update scenario is designed as follows for an update $u$ and a document $t$ typed by a DTD $D$. First, the projection $t'$ of $t$ is built using a type-projector $\pi$. Second, the update $u$ is performed over the projection $t'$, yielding the partial result $u(t')$. We would like to emphasize that no rewriting of the update $u$ is required. The last step, called Merge, parses in a streaming and synchronized fashion both the original document $t$ and $u(t')$ in order to produce the final result $u(t)$. For the sake of efficiency, the Merge step is designed so that (a) only child position of nodes and the projector $\pi$ are checked in order to decide whether to output elements of $t$ or of $u(t')$ and (b) no further changes are made on elements after the partial updated document $u(t')$ has been computed: output elements are either elements of the original document $t$ or elements of $u(t')$. It should be noted that the revalidation issue is not considered in this paper.

Contributions. The main contributions of the paper are:

i) A new 3-level type projector for updates: the first issue is to deal with update expressions; the second issue is related to the choices (a) and (b) for Merge; these choices have a significant impact on the specification of the type-projector; the next section develops motivating examples. Interestingly enough, the new 3-level type projector designed for updates provides interesting improvements for pure queries.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2011 ACM 978-1-4503-0528-0/11/0003 ...$10.00
Related Work. The approach here presented introduces substantial novelties wrt the type based approach for queries presented in [11]. As it will be explained in Sec. 2, we adopt a three-level projector, while the projector proposed in [11] is one level. A three level projector, allows to optimize (minimize) the size of projections. In particular, it allows to avoid keeping in the projection useless text nodes that would be kept with the technique proposed in [11]: this can result into substantial improvements since in many cases large parts of documents consist of textual content.

Other works propose techniques to optimize update 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 [10] [17] [5] [9]. Our work is definitely orthogonal wrt this line of research, and indeed, the two techniques can be combined in order to increase the efficiency in terms of time.

Some recent works [13] [14] addressed the problem of translating an XQuery update expression \( u \) into a pure query expression \( Q_u \), with the aim of executing the update \( u \) via the query \( Q_u \). The advantages of these approaches are that updates can be executed even if the XQuery engine only deals with queries, and well established query-optimization techniques can be adopted to optimize update execution. A peculiar characteristic of these approaches [13] [14] is that the query \( Q_u \) needs to select and return all nodes that are not updated, while those which are updated are selected and processed to compute new nodes. As a consequence, using standard projection techniques [11] [15] for the query \( Q_u \) would lead to no improvement, since the whole document would be projected.

It is worth observing that, although not directly, existing projection techniques [11] [15] could be used for a single update, provided that the projected document is used only to compute the update pending list, so that this last one can be then propagated to the input document in a streaming fashion. Such approach would require some techniques similar to those here developed in order to: opportune determine the projection, and make node identity persistent in order to propagate, in the second phase, the calculated update pending list. This approach has two drawbacks. Firstly, it does not allow to use XML querying engines in a straight manner as we propose to do: controlling the two phase evaluation of XML updates would become necessary. Secondly, this approach would perform very inefficiently in the quite frequent case where a bunch of \( n \) updates has to be executed, according to a given order, because each update would need to be fully processed one after the other entailing the document to be processed/parsed \( n \) times. Our approach is different and allows to evaluate the \( n \) updates by processing our method just once: a global projector can be easily inferred (it is sufficient to consider the union of each update projector); the \( n \) updates are evaluated on the global projection wrt the specified order; finally, the updates are propagated on the original document in a single pass, using the Merge function. As testified by our tests (Section 6), this results in a much more efficient processing.

Organization. The article is organized as follows. Section 2 introduces the main features of our method through examples. Section 3 brings all necessary notation and definitions. Section 4 provides a formal presentation of our method although the inference of the update type projector is addressed separately in Section 5. Section 6 formally states soundness and completeness of our method; it carefully outlines the proof of the main result. The implementation and experiments of the method are reported in Section 6 just before concluding and developing future research directions in Section 7.

2. MOTIVATING EXAMPLES

This section is devoted to introducing and illustrating, through examples, the main features of our method and especially of the update type projector. The choices and assumptions made in the formal presentation are motivated.

Merge explained on a simple example. Let us consider the example in Fig. 1 and assume that the partial updated document \( u(t') \) has been produced by first pruning the original document \( t \) leading to \( t' \) and then updating \( t' \) with \( u \). In order to produce the final result \( u(t) \), we parse and merge the initial document \( t \) and the partial updated document \( u(t') \).
Notice that each node of the initial document $t$ is adorned with its label $(a, b, \ldots)$ and with an identifier $i$ inside square brackets $(1, 1.1, \ldots)$. A node of a document $t$ whose identifier is $i$ is next denoted by $t \therefore i$. We make the choice that the identifier of a node in $t$ gives its position in $t$ according to document order. In the projection $t'$ of $t$, the identifier of a projected node is kept and thus may no more correspond to the position of the node in $t'$ (it is the case, for instance, of the node $t' \therefore 1.4$ in Fig. 1.5). In the partial updated document $u(t')$, new identifiers are assigned to inserted or replaced nodes (see next examples).

Concerning our example, while merging $t$ and $u(t')$, nothing special happens until the nodes $t \therefore 1$ and $u(t') \therefore 1$, both labelled $a$, have been parsed. At this point, the two nodes examined by $\text{Merge}$ are: the first child node $t \therefore 1.1$ labelled $b$ of $t \therefore 1$, and the first child node $u(t') \therefore 1.4$ labelled $d$ of $u(t') \therefore 1$. Because the child rank 4 of $u(t') \therefore 1.4$ is strictly greater than the child rank 1 of $t \therefore 1.1$ and because the label $b$ belongs to the projector $\pi$, indicating that the node $t \therefore 1.1$ has been projected in $t'$, the node $t \therefore 1.1$ is not output (it has been deleted by the update $u$), the original document $t$ is further parsed. The next two nodes examined are: $t \therefore 1.2$ labelled $c$ and $u(t') \therefore 1.4$ labelled $d$. Once again, the child rank 4 of $u(t') \therefore 1.4$ is strictly greater than the child rank 2 of $t \therefore 1.2$, however this time, the label $c$ does not belong to the projector $\pi$ (the node $t \therefore 1.2$ was not needed for the partial update and thus not projected in $t'$) and thus the node $t \therefore 1.2$ is output in the final result, the original document $t$ is further parsed. The process will continue parsing $t$ and $u(t')$ until both documents are fully scanned. Note that, positions of nodes (more precisely child rank) in the initial document play a crucial role in the $\text{Merge}$ process.

Dealing with insertion. Consider the update $u_1$ specified by for $x$ in /doc/a return insert as last <e>"new"<e/> into $x$s with the same DTD $D$ and document $t$ of Fig 1.1 and 1.4. Intuitively, the path corresponding to data relevant for the update $u_1$ is /doc/a and the types of nodes traversed by this path are $\pi_1 = \langle \text{doc}, a \rangle$. The projection $\pi_1(t)$ of $t$ is given below as well as the partial update $u_1(\pi_1(t))$. Recall that node identifiers in $\pi_1(t)$ correspond to node identifiers in $t$, the same holds for unchanged nodes in $u_1(\pi_1(t))$, and that new (inserted or replaced) nodes in $u_1(\pi_1(t))$ are given new identifiers. In the table below, $i$ and $\Gamma$ are new identifiers.

Let us proceed merging the initial document $t$ and the partial result $u_1(\pi_1(t))$ in order to produce the final result $u_1(t)$. After visiting the root nodes of the two documents, the two nodes examined by $\text{Merge}$ are: $t \therefore 1.1$ labelled $b$ and the new node $u_1(\pi_1(t)) \therefore 0$ labelled $c$. Here, the new identifier $i$ conveys no information about child rank of the new node and even if the projector tells us that the node $t \therefore 1.1$ has been projected out, there is no way to decide whether it has to be output before the inserted node or vice-versa. Recall here the assumption made for $\text{Merge}$: information about the update $u_1$ is not available.

In order to solve this problem, related to insertion, we modify the projector. The new projector for the update $u_1$ takes into account that the path /doc/a is the target of an insertion. As such, the projector $\pi_1$ will have 2 components: the type $\text{doc}$ of category ‘node only’ and the type $a$ of category ‘one level below’. Applying this new projector to a document proceeds as follows: the nodes labelled by types of category ‘node only’ are projected; the nodes labelled by types of category ‘one level below’ are projected together with each of their children.

For our example, applying the projector $\pi_1 = \langle \text{doc}, \text{node} \rangle$ with $\text{doc} = \langle \text{doc}, a \rangle$ and $\text{node} = \langle a \rangle$ to the document $t$ leads to the document $\pi_1(t)$ depicted in the table above together with the partial update $u_1(\pi_1(t))$. Since now the new nodes are inserted in a projection containing all their siblings, it is easy to check that the documents $t$ and $u_1(\pi_1(t))$ can be merged in a valid, simple and efficient way.

We would like to stress that our projector avoids unnecessary node projection: the projection of all children of a ‘one level below’ node is forced but, and this is important, without requiring the labels of these children to be part of the projector. Finally, of course, the reader should not confuse projecting all children of a ‘one level below’ node with projecting all its descendants.

Dealing with String and mixed-content. We are now going to slightly modify the DTD $D$ by redefining the rule for $b$ as $b \rightarrow (\text{String} | c)^*$ and consider the update $u_2$ specified by for $x$ in /doc/a where $x$b/text()="foot" return delete $x$/d. Intuitively, /doc/a/d and /doc/a/b/text() are the paths corresponding to data relevant for the update $u_2$. The associated types are $\pi_2 = \langle \text{doc}, a, b, \text{String}, d \rangle$. Let us consider the document $t_2$ given below and its projection $\pi_2(t_2)$. Notice that projecting $t_2$ wrt $\pi_2$ has the side effect to concatenate the two Strings ‘fo’ and ‘ot’ and consequently, the node $u_2(\pi_2(t_2)) \therefore 1.4$ labelled $d$ is deleted when the update $u_2$ is applied on the projected document $\pi_2(t_2)$. Recall the assumption that $\text{Merge}$ is not supposed to change the elements parsed in $t_2$ and $u_2(\pi_2(t_2))$ and has only access to the projector. Thus, we cannot expect that merging the initial document $t_2$ and the partial updated result $u_2(\pi_2(t_2))$ will produce the final updated document.
tion. The new projector \( \pi_{u_2} \) generated for the example will have 2 components: \( \pi_{no}=\{doc, a, d\} \) of category ‘node only’ and \( \pi_{nb}=\{b\} \) of category ‘one level below’. This example is well suited to stress that the notion of projection presented in the paper allows one for a better precision. On the one hand, the projector \( \pi_{u_2} \) allows us to prune out \( c \) children of a nodes as the table above shows. Indeed, we could have solved the problem, in a syntactic manner, by extending the extracted path /doc/a/b/text() to /doc/a/b/text()/parent :: node()/child :: node() leading (by type inference) to a simple projector \{doc, a, b, c, d, String\} which in fact projects the whole document \( t_2 \). On the other hand, the projector \( \pi_{u_2} \) allows us to restrict the projection of text nodes to children of \( b \) nodes. To better illustrate this, let us assume that \( doc \) is now defined by \( doc\rightarrow (a \mid String)\ast \), then applying the simple projector \{doc, a, b, c, d, String\} inferred by \( \Pi \) would lead to project all text children of \( a \) although not useful for the update. This last point is a significant improvement wrt [11] in reducing the size of the projected document as our experiments will show and can also benefit to pure queries.

**Dealing with element extraction.** Consider the DTD \( D \) and the update \( u_3 \) for \( \$x \) in /doc/a/b return replace \( \$x/b \) with \( \$x/d \). First, it is clear that replace updates have to be treated like insert wrt to the target path \( \$x/b \): replace is a delete followed by an insert. Second, because the path /doc/a/d is meant to return the element copied at the target node computed by /doc/b, the complete subtrees rooted at nodes of type \( d \) have to be completely projected. For this update, we propose to generate a projector \( \pi_{u_3} \) composed of three sets of types: \( \pi_{no}=\{doc\} \) of category ‘node only’, \( \pi_{nb}=\{a\} \) of category ‘one level below’, and \( \pi_{nb}=\{d\} \) of category ‘everything below’ (abbreviated ‘\( \forall \) below’).

Let us explain the behavior of the 3-level type projector \( \pi \) the category ‘everything below’: a node labelled by a type of this category is projected together with its sub-forest. Indeed, applying the projector \( \pi_{u_3} \) on the document \( t \) of Fig. 14 produces almost the whole document with the exception of the String ‘oof’ which is pruned out.

Once again, this third feature of our projector brings more precision and efficiency wrt [11]: it allows us for optimizing the projection (by avoiding to include in the projector the types of the nodes in the subtree of a ‘\( \forall \) below’ node) and it accelerates the projection itself.

## 3. PRELIMINARIES

### Data Model

The data model is essentially that of [10] and XML documents are represented using the notion of store.

Next, \( I, J, K \) designate sets (id-set) or lists (id-seq) of identifiers denoted by \( i, j, ... \) (\( () \) denotes the empty id-set; \( I' \) denotes id-seq composition, and the intersection of \( I \) and \( J \) preserving the order in the id-set \( I \) is denoted by \( I \cap J \).

A store \( \sigma \) over the id-set \( I \) is a mapping associating each identifier \( i \in I \) with either an element node \( a[i] \) or a text node \( text[s] \) where \( a \) is a label, \( J \) is an id-seq of identifiers in \( I \) (the ordered list of children) and \( s \) is a string. We define:

- \( lab(\sigma) = \{a[i] \mid i \in I \land \sigma(i) = a[i]\} \) and
- \( String(\sigma) = \{i \mid text(\sigma(i)) \neq \emptyset\} \).

- \( child(\sigma, I) = \{(j \mid 3E i : I, \sigma(i) = a[i], \text{ and } j \in J)\} \)
- \( roots(\sigma) = \{(i \mid \neg \exists j : I, j \in child(\sigma, \{j\}))\} \).

### Update Query Language

The update language we consider is the one proposed in [10], a large core of XQuery Update Facility. The effect of an update \( u \) over an XML document is defined in two steps. A first evaluation of \( u \) produces a sequence of atomic update operations. After checking some properties over these atomic updates, they are ordered and finally applied over the document. Next, we introduce the minimal syntactical and semantic ingredients useful for the presentation.

Atomic updates are defined as follows:

- atom\_mp := ins(l, i, a) \mid del(l) \mid rep(l, \tilde{I}) \mid ren(i, a)

- direction := \( \leftarrow \mid \rightarrow \mid \downarrow \mid \uparrow \mid \gamma \)

The insertion of a set of elements, given by their root identifiers \( I \), targets a node \( a \); it uses a direction parameter \( \delta \) to specify whether to insert before (\( \leftarrow \)), after (\( \rightarrow \)) a node, or into the child list of a node in first (\( \downarrow \)), last (\( \uparrow \)) or arbitrary position (\( \gamma \)).

Deletion or renaming of a subtree \( t \) uses the identifier \( i \) of its root.

Due to space limitation, we do not present the syntax of the query language underlying update expressions. The path axes considered are: child, descendant, parent and ancestor.

The syntax of updates is given by:

- \( u := () \mid \text{insert } q \leftarrow qo \mid \text{del } qo \mid \text{replace } q0 \mid \text{rename } q \leftarrow qo \mid \text{with } q \mid \text{for } x \in q \text{ return } u \mid \text{let } x = q \text{ return } u \)

Obviously, above, \( q \) \ and \( qo \) are queries where \( qo \) is called the target query expression. For instance, the update expression \( \text{insert } q \leftarrow qo \) requires to insert a copy of (the result of) \( q \) in position \( \delta \) relative to the result of \( qo \). In each case the target expression is assumed to evaluate to a single node (identifier) and if not, the evaluation fails.
Semantics of update expressions is defined as in [10]. Now, we outline the definition structure of the judgements introduced in [10]. Below, the stores $\sigma$, $\sigma'$ are forests and $\gamma$ is a variable environment. Queries and updates are also assumed to be closed.

$\sigma, \gamma \mapsto \sigma', I$ : the evaluation of the query $q$ over the forest $\sigma$ under the environment $\gamma$ leads to the new store $\sigma'$ (an extension of the initial store $\sigma$) together with the id-seq $I$ which is the list of identifiers of answer element roots.

$\sigma, \gamma \mapsto u \sim \sigma'$ : applying an update $u$ over $\sigma$ produces the store $\sigma'$; intermediate steps are: producing an atomic update pending list, checking properties of the atomic updates (out of the scope of the paper), and applying the atomic updates.

$\sigma, I \vdash \sigma', I'$ : the copying judgment extends the initial store $\sigma$ by copying each of the subtrees identified by the list $I$ of their roots to a fresh subtree, collecting the root identifiers of the new subtrees in the list $I'$ (the fresh subtree is built with new identifiers).

$\sigma, u \sim \sigma, \omega :$ the evaluation of the update $u$ over $\sigma$ given the environment $\gamma$, starts by producing an update pending list $\omega$ (a list of atomic updates) and a new store $\sigma'$; the store $\sigma'$ extends $\sigma$ with the new or copied elements generated by this phase and required later for evaluating the pending list;

$\sigma \vdash \omega \sim \sigma'$ : applying the update pending list $\omega$ on $\sigma$ produces the new store $\sigma'$.

Of course, $u(t)$ denotes the store $t'$ such that $t(\cdot)\sim u \sim t'$. Given a query path $P$ over a tree $t$, the evaluation is assumed to start at the root of the document (recall that $P=\rel P$). As it does not touch the store, we write $t(\cdot)=P \Rightarrow t, I$.

4. UPDATE MECHANISM

Let us recall the main steps of the update scenario for an update expression $u$ and a document $t$. Step 1: an update type projector $\pi$ is inferred for $u$; the notion of update type projector is defined below: the inference of the type projector is described in Section 5.

Step 2: the update $u$ is evaluated over the projected document $\pi(t)$ producing a partial result $u(\pi(t))$; Step 3: the fully updated document $u(t)$ is built by merging the initial document $t$ and $u(\pi(t))$; this step is detailed below.

Update type projector. First of all, we formally define 3-level type projectors:

**Definition 4.1 (Type Projector).** Given a dtd $(D, sD)$ over the alphabet $\Sigma$, a type projector $\pi$ is a triple $(\pi_{no}, \pi_{olb}, \pi_{eb})$ such that $\pi$ also denotes $\pi_{no} \cup \pi_{olb} \cup \pi_{eb}$;

i) $\pi \subseteq \Sigma$;

ii) $\pi_{no}$, $\pi_{olb}$ and $\pi_{eb}$ are pairwise disjoint, and

iii) $sD \in \pi$ and for each $b \in \pi$ there exists $a \in \pi$ such that $D(a)=b$ and $b$ occurs in $r$.

The $\pi_{no}$ (resp. $\pi_{olb}$ and $\pi_{eb}$) component of $\pi$ contains ‘node only’ types (resp. ‘one level below’ and ‘two level’ types). Notice that condition iii) ensures some closure property wrt to the DTD $D$: label $a \in \pi$ cannot be disconnected from the root label $s_0$ although it does not need to be connected in all possible manners (see projector $\pi_{no}$ below). Notice that the $String$ type itself never belongs to a type projector $\pi$; as explained in Section 2 a string is projected "indirectly" when its parent node type is of category ‘olb’ or ‘eb’.

4. UPDATE MECHANISM

Let us recall the main steps of the update scenario for an update expression $u$ and a document $t$. Step 1: an update type projector $\pi$ is inferred for $u$; the notion of update type projector is defined below: the inference of the type projector is described in Section 5.

Step 2: the update $u$ is evaluated over the projected document $\pi(t)$ producing a partial result $u(\pi(t))$; Step 3: the fully updated document $u(t)$ is built by merging the initial document $t$ and $u(\pi(t))$; this step is detailed below.

Update type projector. First of all, we formally define 3-level type projectors:

**Definition 4.1 (Type Projector).** Given a dtd $(D, sD)$ over the alphabet $\Sigma$, a type projector $\pi$ is a triple $(\pi_{no}, \pi_{olb}, \pi_{eb})$ such that $\pi$ also denotes $\pi_{no} \cup \pi_{olb} \cup \pi_{eb}$;

i) $\pi \subseteq \Sigma$;

ii) $\pi_{no}$, $\pi_{olb}$ and $\pi_{eb}$ are pairwise disjoint, and

iii) $sD \in \pi$ and for each $b \in \pi$ there exists $a \in \pi$ such that $D(a)=b$ and $b$ occurs in $r$.

The $\pi_{no}$ (resp. $\pi_{olb}$ and $\pi_{eb}$) component of $\pi$ contains ‘node only’ types (resp. ‘one level below’ and ‘two level’ types). Notice that condition iii) ensures some closure property wrt to the DTD $D$: label $a \in \pi$ cannot be disconnected from the root label $s_0$ although it does not need to be connected in all possible manners (see projector $\pi_{no}$ below). Notice that the $String$ type itself never belongs to a type projector $\pi$; as explained in Section 2 a string is projected "indirectly" when its parent node type is of category ‘olb’ or ‘eb’.

![Figure 2: Type projection: an example.](image-url)
The reader may notice that no move on line c.2 deals with the case where the current parsed tree essentially guided by t has been pruned out. Hence, if roots(F_i)=∅, otherwise assume F_i=t_o f_i.

Line 3 takes care of synchronization on the nodes r_i; these nodes can only differ by their labels because of some renaming. In that case, the tree TreeMerge(t_i | t_u) is output. The root of the tree TreeMerge(t_i | t_u) is labelled by lab(r_i) and its subforest is defined by:

\[
\begin{align*}
\text{Merge}(F_i | F_u) &= F_u & \text{if roots}(F_i)=\emptyset, \\
& \text{otherwise assume } F_u=t_o f_u.
\end{align*}
\]

Line 4 takes care of the case where the current parsed tree is labelled by lab(r_i) and its subforest is defined by:

\[
\begin{align*}
\text{Merge}(F_i | F_u) &= F_u & \text{if roots}(F_i)=\emptyset, \\
& \text{otherwise assume } F_u=t_o f_u.
\end{align*}
\]

\[
\begin{align*}
t_o \circ \text{Merge}(f_i | F_u) &= \text{if } \sigma_i(r_i) = \text{text}[\text{st}], \\
& \text{otherwise assume } \sigma_i(r_i) = a[J], \\
& \text{Merge}(f_i | F_u) & \text{if } a \in \pi \text{ and either roots}(F_u)=\emptyset \text{ or } F_u=t_o f_u \text{ with } r_i > r_u,
\end{align*}
\]

\[
\begin{align*}
\text{TreeMerge}(t_i | t_u) &= \text{if } \sigma_i(r_i) = \text{text}[\text{st}] \text{ or } \text{new}(r_i) = \text{true}, \\
& \text{otherwise assume } \sigma_i(r_i) = b[K] \text{ and } F_i=t_o f_i.
\end{align*}
\]

\[
\begin{align*}
\text{CMerge}(F_i | F_u) &= F_u & \text{if roots}(F_i)=\emptyset, \\
& \text{otherwise assume } F_u=t_o f_u.
\end{align*}
\]

\[
\begin{align*}
t_u \circ \text{CMerge}(F_i | F_u) &= \text{if } \sigma_u(r_u) = \text{text}[\text{st}] \text{ or } \text{new}(r_u) = \text{true}, \\
& \text{otherwise assume } \sigma_u(r_u) = b[K] \text{ and } F_i=t_o f_i.
\end{align*}
\]

\[
\begin{align*}
\text{TreeMerge}(t_i | t_u) &= \text{if } \sigma_i(r_i) = \text{text}[\text{st}] \text{ or } \text{new}(r_i) = \text{true}, \\
& \text{otherwise assume } \sigma_i(r_i) = b[K] \text{ and } F_i=t_o f_i.
\end{align*}
\]

5. UPDATE TYPE PROJECTOR

This section focuses on the inference of a type projector π given an update u and a DTD D. The extraction of the type projector is decomposed into three steps. First, we proceed to the path extraction from u. Three categories of paths are extracted from u: paths whose targets correspond to ‘one level below’ and ’r below’ nodes. Second, for each category of paths, the DTD D is used in order to derive the labels traversed by these paths and their target labels. The last step is technical and meant to enforce the pairwise disjointness of the projector components (Def. 4.2).

Update Path extraction. It is obvious from the syntax of updates that queries are first class components of updates. An update u may be decomposed into two parts: the context part used to proceed to some navigation and the action part specifying changes to be made over the document. The action part itself can be further decomposed into a source query in charge of building elements to be copied at some position in the document and a target query collecting nodes where changes (insertion, deletion, replacement or renaming) have to be made. To illustrate this, let us consider
the update $u_3$ of Sec2 its context part contains the query /doc/a, its action part is built with the source query $s/d$ and the target query $s/b$. As a consequence, specifying path extraction for updates requires first specifying path extraction for queries.

Our path extraction for queries generalizes where two kinds of paths are distinguished: used paths corresponding to navigation and side queries (for instance, for checking an existential condition) but not involved in building the query result itself; returned paths specifying root nodes of answer elements. Here, for the purpose of our study, among the different groups of paths (here the fact that the action is an insertion or deletion), we have

<table>
<thead>
<tr>
<th>Category</th>
<th>Judgement</th>
<th>Path set derived</th>
</tr>
</thead>
<tbody>
<tr>
<td>node only</td>
<td>$\Gamma, u \sim_{\text{node}} P$</td>
<td>$P_{\text{node}}$</td>
</tr>
<tr>
<td>one level below</td>
<td>$\Gamma, u \sim_{\text{del}} P$</td>
<td>$P_{\text{del}}$</td>
</tr>
<tr>
<td>$\forall$ below</td>
<td>$\Gamma, u \sim_{\text{su}} P$</td>
<td>$P_{\text{su}}$</td>
</tr>
</tbody>
</table>

Judgments for update path extraction.

The guideline to understand the rules for update path extraction relies on analyzing the five categories of paths extracted for queries wrt the environment (context, source, target) of the query path within the update. Let us illustrate this with the update $u_3$. The node returned path $/doc/a$ is inferred from the context query of $u_3$ and, thus, for the update $u_3$, $/doc/a$ is inferred as a node only path. The node returned path $/doc/a/b$ is inferred from the insert target of $u_3$ and as such, for the update $u_3$, $/doc/a$ is inferred as a one level below path (here the fact that the action is an insertion is used to derive $/doc/a/b$ from $/doc/a/b$). Finally, the node returned path $/doc/a/d$ is inferred from the insert target of $u_3$ leading to derive, for $u_3$, $/doc/a/d$ as a $\forall$ below path.

We now present informally and not exhaustively the general analysis underlying path extraction rules for updates given in Fig. Each case considered below is specified by a query path $P_q$ extracted from a query component of an update $u$, more precisely it is specified by the category of $P_q$ and the environment of $q$ within $u$. The presentation relies on the frames in Fig. Each frame shows the target $i$ of a query path $P_q$ and possibly the target $j$ of the corresponding update path $P_u$ when $P_u$ differs from $P_q$ when $P_u=P_q$ we have $i=j$. Each case explains how the path $P_u$ and its category are derived. The nodes that need to be projected are surrounded by dashed lines (a rectangle corresponds to all siblings of a node and a triangle to all its descendants).

1. Assume that $P_q$ is a node used query path extracted from $q$ occurring in the context of the update $u$, then $P_q$ is a node only update path for $u$. The same will be derived if $q$ is a source or target query in $u$ (see Frame 1).
2. Assume that $P_q$ is a $\forall$ below used query path extracted from either the context, source or target of $u$, then $P_q$ is a $\forall$ below update path for $u$ (see Frame 2).
3. Assume that $P_q$ is a string used query path extracted from
any environment of q, then \( P_u = P_{\pi} / \text{parent} \), node() is a one level below update path for \( u \). This case (mixed-content data) has been motivated in Sec. 2 (see Frame 3 where i has been replaced by st in order to represent that the target of \( P_q \) is a string).

4. Let us turn to the case where \( P_q \) is a node returned query path extracted from the query q of the action \( \text{del}(q) \) of u. This case is quite simple: it is unnecessary to project the siblings of node i and thus \( P_q \) is inferred as a node only update path for u. Note that the case where a string node is deleted is captured as an access to a string followed by delete and as such falls into case 3. above.

5. Now consider that \( P_q \) is a node returned query path extracted from the query q of \( u = \text{insert} \ q \ \delta \ q \ \delta \ q \) with \( \delta \in \{\leftarrow, \rightarrow\} \).

Then (see Frame 5), as motivated in Sec. 2, all siblings of node i need to be projected to give Merge the ability to recover the nodes in valid order: thus \( P_q = P_{\pi} / \text{parent} \), node() is derived as a one level below update path. The same conclusion is obtained when \( u = \text{replace} \ q \) with \( \delta \).

6. Finally, consider that \( P_q \) is a returned query path extracted from the query q of \( u = \text{insert} \ q \ \delta \ i \) with \( \delta \in \{\\rightleftharpoons, \downarrow\} \).

This time, the insertion possibly adds a new child to i. Thus, \( P_q \) itself is derived as a one level below update path.

**Path Type Inference.** This step relies on the type inference rules of [11] which are not reported here. It starts with the three sets of path expressions \( P_{\text{no}}, P_{\text{olb}}, \) and \( P_{\text{eb}} \) inferred for \( u \). For each set \( P \), it produces a pair \((T, C)\) of sets of types.

- Considering path expressions in \( P \) as (simple) queries, \( T \) collects all labels of the answer roots for these queries. Formally, for any \( \pi \in D \) and \( P \in P \), assuming \( t, (i)=P \Rightarrow t, i: J \).

\[
\text{if } i \in J \text{ and } \sigma(t(i)) = a[I] \text{ then } a \in T
\]

Note that here, we only infer element labels. The String type is not considered in the inferred type because, in our setting, projecting a String node is a side effect of marking its parent node label as one level below.

For the update \( u_3 \), from \( P_{\text{no}} = \{/doc/a\}, P_{\text{olb}} = \{/doc/a\} \) and \( P_{\text{eb}} = \{/doc/a/d\} \), we derive: \( T_{\text{no}} = \{a\}, T_{\text{olb}} = \{a\} \) and \( T_{\text{eb}} = \{d\} \).

- Besides inferring an answer type set \( T \), we also infer a context type set \( C \) containing labels of all ancestors of nodes in the sequence \( I \) output by \( P \). As explained in [11] the use of context types is crucial to ensure precision of the projector. Formally, for any \( \pi \in D \) and \( P \in P \), assuming \( t, (i)=P \Rightarrow t, i: J \).

\[
\text{if } i \in \text{idmatch}(t, J) \text{ and } \sigma(t(i)) = a[I] \text{ then } a \in C
\]

where given a set \( J \) of identifiers, \( \text{idmatch}(t, J) \) collects, for each \( j \in J \) the node identifiers along the (concrete) paths from the root to \( j \), excluding the identifier \( j \).

For the update \( u_3 \), we derive \( C_{\text{no}} = \{doc\}, C_{\text{olb}} = \{doc\}, \) and \( C_{\text{eb}} = \{doc, a\} \).

So type inference actually produces a pair \( \Sigma = (T, C) \). As for query path and update path extraction, type inference rules specify the judgement \( \Sigma \vdash \rho: P : \Sigma \) where \( \Sigma' = (T, C) \) is a starting environment. This judgement means that given a DTD \( D \), starting from the labels in \( T \) and the context \( C \), the path \( P \) generates the labels \( T \) with its context \( C \).

The main theorem satisfied by type inference rules is the following:

**Theorem 5.1** ([11]). Let \( (D, s_D) \) be a DTD, \( P \) a path, and \( t \in D \). If \( t, (i)=P \Rightarrow t, J \) and \( (\{s_D\}, \{\}) \vdash \rho \) then:

\[
T_P \subseteq T \quad \text{and} \quad C_P \subseteq C
\]

where \( T_P = \{a \mid a \in J \text{ and } \sigma(t(j)) = a[I']\} \), and \( C_P = \{a \mid a \in \text{idmatch}(t, J) \} \).

**Type-projector inference.** The final step of the update type projector derivation starts with the three pairs of type sets \((T_{\text{no}}, C_{\text{no}}), (T_{\text{olb}}, C_{\text{olb}}), (T_{\text{eb}}, C_{\text{eb}})\) inferred for \( u \). This step is quite straightforward although it contributes to the efficiency and precision of the type projector by enforcing pairwise disjointness of the 3 components of the type projector. First, it is rather immediate to see that types in \( C_{\text{no}} \) are all of the category ‘node only’. Then, it is also obvious (from the definition of the projection) that types of category ‘\( \uparrow \) below’ do not need to be kept neither in the category ‘one level below’ nor in the category ‘node only’ and similarly, types of category ‘one level below’ do not need to be kept in the category ‘node only’.

**Definition 5.2 (Type Projector Extraction).** The type projector \( \pi = (\pi_{\text{no}}, \pi_{\text{olb}}, \pi_{\text{eb}}) \) for \( u \) is given by:

\[
\begin{align*}
\pi_{\text{no}} & = T_{\text{no}} \cup C_{\text{no}} \cup C_{\text{olb}} \cup C_{\text{eb}} \\
\pi_{\text{olb}} & = T_{\text{olb}} \cup \pi_{\text{eb}} \\
\pi_{\text{eb}} & = T_{\text{eb}}
\end{align*}
\]

The type projector for the update \( u_3 \) is given in Section 2.

At this point of the presentation, we would like to highlight how simple it is to use our framework to execute a sequence of updates \( u_1, \ldots, u_n \).

Indeed, it suffices to generate each projector \( \pi_i \) for \( u_i \) and build the global projector \( \pi \) as the union of the \( \pi_i \) (enforcing in the obvious manner disjointness of the 3 projector components). Given a document \( t \in D \), the updated document \( u_n(...(u_1(t))) \) is obtained by first projecting \( t \) wrt \( \pi \) then applying successively \( u_1, \ldots, u_n \) on the projection \( \pi(t) \), and finally merging the initial document with the partially updated document \( u_n(...(u_1(\pi(t)))) \).

**Main Results.** The main result states that the update scenario based on the 3-level type projection is sound and complete. Formally:

1. The reader should not confuse the single update \( u_1, \ldots, u_n \) with the sequence of updates \( u_1, \ldots, u_n \). Here we focus on the latter case.
Theorem 5.3. Let \( u \) be an update over \( D \) and \( \pi \) be the inferred type projector for \( u \). Then for each \( p \)-tree \( t \in D \), we have: \( \text{Merge}(t \mid u(\pi(t))) \sim u(t) \).

Above, value equivalence \( \sim \) (formally defined later on) captures the idea that the two processes return the same document up to node identifiers.

The previous result strongly relies on the fact that the inferred type projector is sound which is formally stated by:

**Theorem 5.4** (Soundness of update type projector). With the same assumption as in Theorem 5.3, we have: \( u(\pi(t)) \sim \Pi_J(u(t)) \) where \( J = \text{dom}(u(t)) \setminus \{\text{dom}(t) - K(t, \pi)\} \).

This result corresponds to some kind of commutative diagram (see Fig. 8) involving projection and the update \( u \); roughly, it tells that updating the projection is equivalent to projecting the update of \( t \). The reader should pay attention on the way \( u(t) \) is projected wrt \( J \). The set \( J \) contains the identifiers of \( u(t) \) that have to be kept during the projection \( \pi \) and still in \( u(t) \). Said differently, the above result states that elements in \( t \) located "out" of the position set \( K(t, \pi) \), captured by \( \text{dom}(t) - K(t, \pi) \), are not influential for the update: the queries of the update do not use these elements which are neither updated (touched by an insert, a rename, a replace or a delete).

The proof of the main Theorem 5.3 is decomposed in two main steps. First we prove Theorem 5.4 stating that elements pruned out by the projector set \( K(t, \pi) \) are not influential for the update \( u \). Then, assuming Theorem 5.4, we show that \( \text{Merge} \) builds the document \( u(t) \).

**Sketch of proof of Theorem 5.4.** Recall that the semantics of updates is specified in two steps: (1) producing an update pending list, (2) applying the elementary updates of the pending list after some test and reordering. The proof of Theorem 5.4 (soundness wrt update) essentially relies on the intermediate semantics given by update pending list. The intermediate result 5.5 below is the core of the proof of 5.4. In order to state this result, we need some preliminary definitions. The purpose of the first definition is to check whether two lists of trees are equal up to identifiers.

**Definition 5.5** (Value equivalence). Let \( \sigma \) and \( \sigma' \) be stores over \( I \) and \( I' \) resp. Let \( J \) and \( J' \) be two id-seqs such that \( J \subseteq I \) and \( J' \subseteq I' \). The value equivalence \( (J, \sigma) \sim (J', \sigma') \) is recursively defined by:

- \((I, \sigma)\sim(\emptyset, \sigma')\) always holds;
- \((I, \sigma)\sim(J', \sigma')\) iff \((J, \sigma)\sim(J', \sigma')\) and
  - \(\sigma(i) = s[K]\) implies \(\sigma'(i) = s[K']\) and \((K, \sigma)\sim(K', \sigma')\);
  - \(\sigma(i) = \text{text}(st)\) implies \(\sigma'(i) = \text{text}(st)\)

Value equivalence can be extended to a pair of forests \( f \) and \( f' \). We write \( f \sim f' \) for \((\text{roots}(f), \sigma_f) \sim (\text{roots}(f'), \sigma_{f'})\).

The purpose of the second definition is to check whether two update pending lists are equal, once again up to identifiers.

**Definition 5.6** (Update list equivalence). Let \( \sigma \) and \( \sigma' \) be stores over \( I \) and \( I' \) resp. Let \( \omega \) and \( \omega' \) be two atomic update lists. The equivalence \( (\omega, \sigma) \sim (\omega', \sigma') \) is recursively defined, in the obvious manner, from the base cases given below:

- \((\text{ins}(J, \delta, i), \sigma)\sim(\text{ins}(J', \delta, j), \sigma')\) iff \(i = j\) and \((J, \sigma)\sim(J', \sigma')\);
- \((\text{del}(i), \sigma)\sim(\text{del}(j), \sigma')\) iff \(i = j\);
- \((\text{rep}(i, J), \sigma)\sim(\text{rep}(j, J'), \sigma')\) iff \(i = j\) and \((J, \sigma)\sim(J', \sigma')\);
- \((\text{ren}(i, a), \sigma)\sim(\text{ren}(j, b), \sigma')\) iff \(a = b\) and \(i = j\).

The two following definitions establish what is meant for a projector set to be sound wrt to a path expression, in the one hand and wrt to an update, in the other hand. As in the rest of the presentation, we make the choice here not to detail what happens for pure queries that is what is a sound projector set for a pure query, although it is of course a component of the proof.

**Definition 5.7** (Sound projector for a path). A pair of id-set \((K_T, K_C)\) is a sound projector for the path expression \( P \) on the p-tree \( t \) iff

- \(t \mid P \Rightarrow t\) implies \(\Pi_K(t, \omega) \Rightarrow K_T\), and
- \(J \subseteq K_T\).

Recall that the set of identifiers \( J \) in \( t \), \( \omega \Rightarrow t \) captures identifiers of answer roots for \( P \). Intuitively, in the above definition, the id-set \( K_T \) is a super set of \( J \) and thus captures, at least, the answer roots for \( P \), while the id-set \( K_C \) captures, at least, identifiers in \( \text{idmatch}(t, J) \). Here, the distinction between \( K_T \) and \( K_C \) is necessary to proceed to the right treatment of targets of path matching \( P \) depending on the category they belong to.

**Definition 5.8** (Sound projector for an update). An id-set \( K\) is a sound projector for the update (pending list of) \( u \) on the p-tree \( t \) iff \(t \mid u \Rightarrow \sigma, \omega \) implies \(\Pi_K(t, \omega) \Rightarrow \sigma', \omega' \sim (\omega, \sigma)\).

Of course, soundness wrt to an update projector set is expressed based on the intermediate semantics given by update pending list. Indeed, the proof of Theorem 5.7 is based on showing that for a given document, the projector set \( K(t, \pi) \) is sound. Formally:

**Lemma 5.9.** Let \( D \) be a DTD and \( u \) be an update with its inferred type projector \( \pi \). For any tree \( t \in D \): \( K(t, \pi) \) is a sound projector set for the update (pending list of) \( u \) on the tree \( t \in D \).

The proof of this lemma relies on showing that the update path inference is sound wrt update. Formally, we show that:

**Theorem 5.10** (Soundness of Path Inference 1). Let us consider the id-set \( K = K_{\ref{no}} \cup K_{\ref{ah}} \cup K_{\ref{eh}} \) defined below. We have that \( K \) is a sound projector for the (pending list of the) update \( u \) on the p-tree \( t \).

- Let us assume that \((\Gamma, u) \trianglelefteq P_{\ref{no}}\) with \(P_{\ref{no}} = (P_{\ref{no}, 1}, \ldots, P_{\ref{no}, n})\) and consider for \(i = 1, k_{\ref{no}}\), a sound projector set \((K_{T_{\ref{no}}}, K_{C_{\ref{no}}})\) for \(P_{\ref{no}, i}\).

Then, \(K_{\ref{no}} = \cup_{i=1}^{k_{\ref{no}}} (K_{T_{\ref{no}}} \cup K_{C_{\ref{no}}})\).
• Let us assume that \((\Gamma, u) \sim_{\mathcal{H}} P_{\mathcal{O}}\) with 
\[ P_{\mathcal{O}} = \{P_{\mathcal{O}_1}, \ldots, P_{\mathcal{O}_k}\} \] 
and consider for \(i = 1, k_{\mathcal{O}}\), a sound projector set \(K_{\mathcal{O}_i}, K_{C_{\mathcal{O}_i}}\) for \(P_{\mathcal{O}_i}\). Then, 
\[ K_{\mathcal{O}} = \bigcup_{i=1}^{k_{\mathcal{O}}} (K_{\mathcal{O}_i} \cup K_{C_{\mathcal{O}_i}} \cup \text{child}(t, K_{\mathcal{O}_i})) \]

• Let us assume that \((\Gamma, u) \sim_{\mathcal{H}} P_{\mathcal{E}}\) with 
\[ P_{\mathcal{E}} = \{P_{\mathcal{E}_1}, \ldots, P_{\mathcal{E}_k}\} \] 
and consider for \(i = 1, k_{\mathcal{E}}\), a sound projector set \(K_{\mathcal{E}_i}, K_{C_{\mathcal{E}_i}}\) for \(P_{\mathcal{E}_i}\). Then, 
\[ K_{\mathcal{E}} = \bigcup_{i=1}^{k_{\mathcal{E}}} (K_{\mathcal{E}_i} \cup K_{C_{\mathcal{E}_i}} \cup \text{desc}(t, K_{\mathcal{E}_i})) \]

Indeed, we start by proving the following intermediate result which is, in some sense, more precise:

**Lemma 5.11** (Soundness of Path Inference II). Let us consider the id-set \(K = K_{\mathcal{O}} \cup K_{\mathcal{E}} \cup K_{\mathcal{H}}\) defined below. We have that \(K\) is a sound projector for the (pending list of the) update \(u\) on the p-tree \(t\).

• Let us assume that \((\Gamma, u) \sim_{\mathcal{H}} P_{\mathcal{O}}\) with 
\[ P_{\mathcal{O}} = \{P_{\mathcal{O}_1}, \ldots, P_{\mathcal{O}_k}\} \] 
and for \(i = 1, k_{\mathcal{O}}\), \(t(\mathcal{O}_i) = \text{child}(t, P_{\mathcal{O}_i})\). Then, 
\[ K_{\mathcal{O}} = \bigcup_{i=1}^{k_{\mathcal{O}}} (J_{\mathcal{O}_i} \cup \text{idmatch}(t, J_{\mathcal{O}_i} \cup \text{child}(t, J_{\mathcal{O}_i}))) \]

• Let us assume that \((\Gamma, u) \sim_{\mathcal{H}} P_{\mathcal{E}}\) with 
\[ P_{\mathcal{E}} = \{P_{\mathcal{E}_1}, \ldots, P_{\mathcal{E}_k}\} \] 
and for \(i = 1, k_{\mathcal{E}}\), \(t(\mathcal{E}_i) = \text{child}(t, P_{\mathcal{E}_i})\). Then, 
\[ K_{\mathcal{E}} = \bigcup_{i=1}^{k_{\mathcal{E}}} (J_{\mathcal{E}_i} \cup \text{idmatch}(t, J_{\mathcal{E}_i} \cup \text{child}(t, J_{\mathcal{E}_i}))) \]

In order to prove Theorem 5.10, the above result (Lemma 5.11) is combined with type inference (see Theorem 5.1).

**Sketch of proof of Theorem 5.1** To conclude this section, we would like to highlight that the proof of Theorem 5.3 and more precisely the part showing that \(\text{Merge}\) builds the updated document \(u(t)\) uses the fact that the type projector \(\pi\) is initially applied over a p-tree. Once again, p-tree is specified such that node identifiers correspond to node positions. Intuitively, Lemma 5.9 expresses that the update pending list generated for the projected document \(\Pi_{K(\Gamma, u)}(t)\) and \(u\) targets the same positions for performing changes as the update pending list generated for the initial document and \(u\).

From this, proving that \(\text{Merge}\) builds a valid result \(u(t)\) from \(t\) and \(u(t')\) does not present any deep difficulty although it is technically involved. It suffices to proceed recursively to a careful case study.

### 6. Implementation & Experiments

**Implementation issues.** In order to validate the effectiveness of our method, we have implemented both projection and merge algorithms in Java. The only technical gap between the formal method and its implementation concerns node identifiers or positions. Although made explicit in the formal scenario, the implementation does not materialize positions in the input document \(t\); it is not necessary. Positions are generated on the fly while parsing \(t\), during projection and during Merge. Indeed, for each node, the implementation generates its rank among its siblings: full node position is not necessary. In \(\pi(t)\), this rank is stored by means of a special new attribute for node only/one level below nodes and by means of another new attribute for \(\forall\) below node.

The potential overhead due to these special attributes is mitigated by the size reduction ensured by projection. The use of two distinct attributes is required for technical reason related to insertion and replace updates and also to the way source elements are copied during their execution.

The algorithm \(\text{Merge}\) is implemented by means of two threads, parsing resp. \(t\) and \(\pi(t)\). These threads are defined in terms of classes obtained by extending existing SAX parser classes. The two threads interact with each other according to the Producer-Consumer pattern.

**Experiments.** Several tests have been performed using our Java implementation and 7 updates on XMark documents of growing size. These updates, together with their associated projectors, are reported in the following, and cover the main update operations made available by XQuery Update Facility (insert, rename, replace and delete). All experiments were performed on a 2.53 Ghz Intel Core 2 Duo machine (2 GB main memory) running Mac OSX 10.6.4.

The sizes of projected documents are reported in Fig. 8.

**Figure 8:** Documents size reduction after pruning

The first kind of tests aims at detecting memory limitations of four popular query processors implemented in Java: Saxon EE 9.2.0.2, QizX Free-Engine-3.2.0, eXist 1.2.5 and MXQuery 0.6.0. We set to 512 MB the Java virtual machine memory, while the size of XMark documents considered goes from 50 MB to 2 GB. The sizes of largest documents these processors could update without projection are reported in Fig. 9. For this test, we used the less memory consuming update U4. Three out of four systems cannot deal with documents whose size is greater than 150 MB, while QizX is able to process documents whose size is slightly higher than the Java virtual memory size (this is due to some efficient techniques adopted by QizX for compacting internal document representation).

The second kind of tests evaluate our projection based technique. We focused on two systems Saxon and QizX, and used the whole set of 7 updates. In both cases, tests

![Table of maximal input sizes](image-url)

**Figure 9:** Maximal input sizes
show that our technique can ensure great improvements. In all figures, missing value for time means memory failure.

Concerning Saxon, tests results are synthesized in Fig. 10.1 and 10.2, reporting, respectively, total execution time by using and by not using projection. They clearly show that our technique succeeds in its primarily purpose: making possible to update very large documents with in-memory systems, in the presence of memory limitations. Note that the total time in the case of projected documents (Fig. 10.2 and 10.4) includes time for i) projecting the input, ii) storing the projection, iii) updating the projection and storing it, and iv) performing the final merge. Nevertheless, for documents that can be updated even without projection, execution time with projection remains comparable to that without projection. This is because the time spent for projection, merging and so on is recovered by a faster update process thanks to a significantly smaller size of the projected document (Fig. 10.3). Also observe that for U5, Saxon with projection was not able to update documents for size greater than 1 GB (due to memory failure). The projector of this update reveals that this is due to its low selectivity.

QuizX shows less severe memory limitations. Total execution times are reported in Fig. 10.3 and 10.4. We still have great improvements in terms of memory: with projection, we can update up to 2GB for all queries, while without projection the limit is 520 MB. However, for QuizX, projection also ensures sensible total execution time reduction. This is in part due to the fact that QuizX needs a significant time to build auxiliary indexes at loading time. This improvement in terms of execution time also testifies the effectiveness of our design choices at the projector, path extraction (Sec. 2 and 3), and Merge function level. For the 52MB document, we have the following reductions of execution times, expressed in percentages: U1 (45.4%), U2 (60.3%), U3 (74.3%), U4 (72.2%), U5 (45.2%), U6 (63.6%), U7 (24%). We had similar percentages for documents of other sizes.

A last kind of tests we made concerns the computation of a unique projection for all the updates, executed in the following order: U1, U2, U3, ..., U7. The document has been projected once, then all the updates have been evaluated on the projection, and finally Merge has been executed once to obtain the final document. With Saxon and QuizX this took, respectively, 82 and 64 seconds on the 128MB document. For this document, the sum of total times needed to project, updating and merging for each single update was much higher, respectively 181 and 194 seconds for Saxon and QuizX.

The updates and the corresponding projectors.

| U1 | site, closed.auctions, annotation | closed.auct. | 0 |
| U2 | site, people, address | person, country, street, province, zipcode | 0 |
| U3 | site, regions, africa, asia, australia, europe, namerica, samerica, item | location | 0 |
| U4 | site, regions, africa, asia, australia, europe, namerica, samerica, item, mailbox, mail | 0 | 0 |
| U5 | site, regions, africa, asia, australia, europe, namerica, samerica, item, mailbox, mail, bold, mailbox, mail, item, description, text, open.auctions, open.auction, closed.auctions, closed.auction, annotation, parlist | 0 | 0 |
| U6 | site, people, homepage | person, name | 0 |
| U7 | site, people | person, name, country | address |

U3. for $x$ in $doc/site/regions/item/location where $x/text()="United States" return (replace value of node $x$ with "USA")

U4. delete nodes $doc/site/regions/item/mailbox/mail

U5. for $x$ in $doc/site/text/bold return node $x$ as "emph"

U6. for $x$ in $doc/site/people/person where not($x/homepage) return insert node <homepage>www.$x/name/text().com</homepage> after $x/emailaddress

U7. for $x$ in $doc/site/people/person, for $y$ in $doc/site/people/person where $x/name = $y/name and not ($y/address) and $x/country='Malaysia' return insert node $x/address after $y/emailaddress

7. CONCLUSIONS AND FUTURE WORKS

To the best of our knowledge, the technique we have presented here is the first XQuery update optimization technique based on the use of projection and schema information. One of its main distinctive features is a new notion of projector allowing to strictly minimize the resulting projection, and to efficiently propagate updates from the updated projection to the initial database. Another distinctive feature is that the proposed framework can be exploited without changing any internal part of the query/update engine.

In order to have a more efficient implementation, we plan to eliminate: (i) storing the pruned document on the disk, and (ii) storing and re-reading the partial update pruned...
Updating without projection using Saxon

Updating with projection using Saxon

Updating without projection using Qizx

Updating with projection using Qizx

Figure 10: Results of the tests performed on Saxon and Qizx

document. This requires some strong interaction with the update processor, and hence further implementation efforts; anyway, we realized that this would probably lead to a reduction of about 50% of the time indicated now in Table 10.2. This would also imply, that even when projection is not necessary, it can reduce execution time for Saxon as well (for QizX we already have sensible improvements in terms of time).

We are currently working on several directions in order to further reduce the size of projected documents. One of the goal is to replace the one level below projection process by a less greedy one. This could be done by further refining the update expression analysis, taking into account the kind of insertion occurring in the update. Relaxing the “no rewriting of update” assumption, on which our update scenario has been built, is another interesting direction of investigation.

We are also currently investigating how projection based update can be applied to temporal XML documents in order to ensure a compact storage.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. This work has been partially funded by the Codex project, Agence Nationale de la Recherche, decision ANR-08-DEFIS-004.

8. REFERENCES


