Projection for XML Update Optimization.

Mohamed-Amine Baazizi
Leo Team
Univ. Paris-Sud - INRIA
Saclay
baazizi@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, 18, 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.

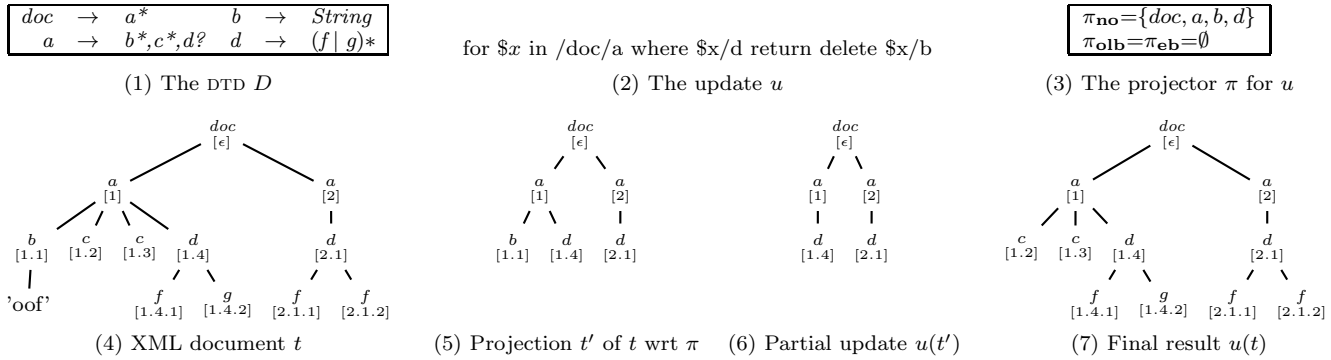
EDBT 2011, March 22–24, 2011, Uppsala, Sweden.

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 π. 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 π 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.



(1) The DTD D (2) The update u (3) The projector π for u

(4) XML document t (5) Projection t' of t wrt π (6) Partial update $u(t')$ (7) Final result $u(t)$

Figure 1: A motivating example of the Update Scenario

ii) A new path extraction mechanism required for the derivation of the update type projector.

iii) Design and implementation of a simple and thus efficient algorithm *Merge*, to make updates persistent. The *Merge* algorithm uses a buffer whose size is upper bounded by the maximal depth of the input document t.

iv) Extensive experiments whose results validate the effectiveness of the proposed approach. We have implemented the projection and merging algorithms in Java and considered several popular systems to perform tests.

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 [16, 17, 8, 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, 18] 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, 18] 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: opportunely 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 5 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

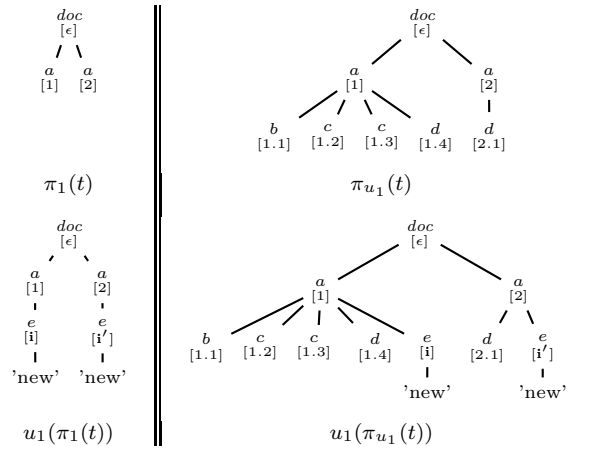
This section is devoted to introducing and illustrating, through examples, the main features of our method and especially of the update type projector. The choices and assumptions made in the formal presentation are motivated.

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, ...) and with an identifier **i** inside square brackets (1, 1.1, ...). A node of a document t whose identifier is **i** is next denoted by t@**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'@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@1 and $u(t')$@1, both labelled a, have been parsed. At this point, the two nodes examined by *Merge* are: the first child node t@1.1 labelled b of t@1, and the first child node $u(t')$@1.4 labelled d of $u(t')$@1 . Because the child rank 4 of $u(t')$@1.4 is strictly greater than the child rank 1 of t@1.1 and because the label b belongs to the projector π, indicating that the node t@1.1 has been projected in t', the node t@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@1.2 labelled c and $u(t')$@1.4 labelled d. Once again, the child rank 4 of $u(t')$@1.4 is strictly greater than the child rank 2 of t@1.2, however this time, the label c does not belong to the projector π (the node t@1.2 was not needed for the partial update and thus not projected in t') and thus the node t@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 *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` 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 π_1=$\{doc, a\}$. 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 **i**$'$ are new identifiers.



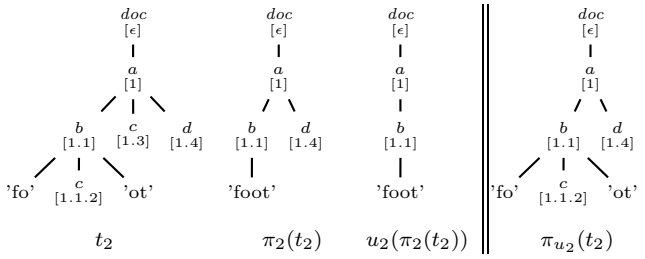
Let us proceed to merging the initial document t and the partial result $u_1(\pi_1(t))$ in order to produce the final result $u_1(t)$. After visiting the root nodes of the two documents, the two nodes examined by *Merge* are: t@1.1 labelled b and the new node $u_1(\pi_1(t))$@**i** labelled e. Here, the new identifier **i** conveys no information about child rank of the new node and even if the projector tells us that the node t@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 *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 π_{u_1} will have 2 components: the type 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 π_{u_1}=($\pi_{\mathbf{no}}, \pi_{\mathbf{olb}}$) with $\pi_{\mathbf{no}}$=$\{doc\}$ and $\pi_{\mathbf{olb}}$=$\{a\}$ to the document t leads to the document $\pi_{u_1}(t)$ depicted in the table above together with the partial update $u_1(\pi_{u_1}(t))$. Since now the new nodes are inserted in a projection containing all their siblings, it is easy to check that the documents t and $u_1(\pi_{u_1}(t))$ 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 $b \rightarrow (String \mid c)*$ 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 π_2=$\{doc, a, b, String, d\}$. Let us consider the document t_2 given below and its projection $\pi_2(t_2)$. Notice that projecting t_2 wrt π_2 has the side effect to concatenate the two *Strings* 'fo' and 'ot' and consequently, the node $u_2(\pi_2(t_2))$@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 *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.



The problem here is due to mixed-content nodes and solved by modifying the projector in the same way as for inser-

tion. The new projector π_{u_2} generated for the example will have 2 components: $\pi_{\mathbf{no}}$=$\{doc, a, d\}$ of category 'node only' and $\pi_{\mathbf{olb}}$=$\{b\}$ 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 π_{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 π_{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 $doc \rightarrow (a \mid String)*$, then applying the simple projector $\{doc, a, b, c, d, String\}$ inferred by [11] 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 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 π_{u_3} composed of three sets of types: $\pi_{\mathbf{no}}$=$\{doc\}$ of category 'node only', $\pi_{\mathbf{olb}}$=$\{a\}$ of category 'one level below', and $\pi_{\mathbf{eb}}$=$\{d\}$ of category 'everything below' (abbreviated '$\forall$ *below*').

Let us explain the behavior of the 3-level type projector wrt the category 'everything below': a node labelled by a type of this category is projected together with its sub-forest. Indeed, applying the projector π_{u_3} on the document t of Fig. 1.4 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 it-self.

3. PRELIMINARIES

Data Model. The data model is essentially that of [10] and thus 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 $\mathbf{i}$, $\mathbf{j}$ ...; () denotes the empty id-seq; $I{\cdot}I'$ denotes id-seq composition, and the intersection of I and J preserving the order in the id-seq I is denoted by $I_{|J}$.

A *store* σ over the id-set I is a mapping associating each identifier $\mathbf{i}{\in}I$ with either an element node $a[J]$ or a text node $text[st]$ where a is a label, J is an id-seq of identifiers in I (the ordered list of children) and st is a string. We define:
– $lab(\mathbf{i})$=a if $\sigma(\mathbf{i})$=$a[J]$, and $lab(\mathbf{i})$=$String$ if $\sigma(\mathbf{i})$=$text[st]$,
– $child(\sigma, I)$=$\{\mathbf{j} \mid \exists \mathbf{i}{\in}I, \sigma(\mathbf{i})$=$a[J]$ *and* $\mathbf{j}{\in}J\}$,
– $roots(\sigma)$=$\{\mathbf{i} \mid \neg\exists \mathbf{j}, \mathbf{i}{\in}child(\sigma, \{\mathbf{j}\})\}$.

Given a store σ over I, the *projection* on $J{\subseteq}I$ of σ, is a store over J, denoted $\Pi_J(\sigma)$, defined by: for each $\mathbf{j}{\in}J$, if $\sigma(\mathbf{j})$=$a[K]$ then $\Pi_J(\sigma)(\mathbf{j})$=$a[K_{|J}]$ otherwise $\sigma(\mathbf{j})$=$text[st]$ and $\Pi_J(\sigma)(\mathbf{j})$=$\sigma(\mathbf{j})$. The reader should pay attention to the fact that the domain and the "co-domain" of the *projection* on J of σ is J.

We only consider stores corresponding to XML forests and trees. A forest f over I is given by a pair (J, σ) where σ is as above and J=$roots(\sigma)$. We write $dom(f)$ for I and σ_f for σ and $f{\circ}f'$ for the concatenation of two disjoint forests f and f'.

Similarly, a tree t over I is given by (r_t, σ_t) where r_t is the root identifier of the store t over I that is, $roots(\sigma_t)$=$\{r_t\}$. The *subforest of* t, denoted $subfor(t)$, is defined by $\Pi_{I\setminus\{r_t\}}(t)$.

For the sake of the formal presentation, the identifiers used in the definition of a store are sometimes giving the position of the nodes in the XML document (see the motivating example of Section 2). Such stores are called *p-stores*.

We consider XML trees valid wrt a schema defined by means of the DTD language, which features the core mechanisms of mainstream schema languages.

DTDs are defined as in [15]: given a finite set of labels 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 $s_D \in \Sigma$ is the root symbol. Given a regular expression r, the language generated by r, resp. the set of symbols in Σ occurring in r, is denoted by $\mathscr{L}(r)$, resp. $S(r)$. We denote $t{\in}D$ the fact that t is valid wrt D.

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 :

$$\begin{array}{lll} atom\_up & ::= & \mathrm{ins}(\mathrm{I}, \delta, \mathbf{i}) \mid \mathrm{del}(\mathbf{i}) \mid \mathrm{repl}(\mathbf{i},\mathrm{I}) \mid \mathrm{ren}(\mathbf{i}, a) \\ direction & ::= & \leftarrow \mid \rightarrow \mid \downarrow \mid \swarrow \mid \searrow \end{array}$$

The insertion of a set of elements, given by their root identifiers I, targets a node $\mathbf{i}$; it uses a direction parameter δ to specify whether to insert before ($\leftarrow$), after ($\rightarrow$) a node, or into the child list of a node in first ($\swarrow$), last ($\searrow$) or arbitrary position ($\downarrow$). Deletion or renaming of a subtree t uses the identifier $\mathbf{i}$ 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:

$$\begin{array}{lll} u & ::= & () \mid \texttt{insert}\ q\ \delta\ q_0 \mid \texttt{del}\ q_0 \mid \texttt{replace}\ q_0\ \texttt{with}\ q \mid \\ & & \texttt{rename}\ q\ \texttt{as}\ a \mid u, u' \mid \texttt{if}\ q\ \texttt{then}\ u_1\ \texttt{else}\ u_2 \mid \\ & & \texttt{for}\ x\ \texttt{in}\ q\ \texttt{return}\ u \mid \texttt{let}\ x\ \texttt{=}\ q\ \texttt{return}\ u \end{array}$$

Obviously, above, q and q_0 are queries where q_0 is called the target query expression. For instance, the update expression `insert` $q\ \delta\ q_0$ requires to insert a copy of (the result of) q in position δ relative to the result of q_0. 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 σ, σ' are forests and γ is a variable environment. Queries and updates are also assumed to be closed.

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

– $\sigma, \gamma \models u \rightsquigarrow \sigma'$: applying an update u over σ produces the store σ'; 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.

– $\sigma, I \overset{copy}{\vdash} \sigma', I'$: the copying judgment extends the initial store σ 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).

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

– $\sigma \models \omega \rightsquigarrow \sigma'$: applying the update pending list ω on σ produces the new store σ'.

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

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 π for u is inferred and t is projected wrt π. 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, s_D) *over the alphabet* Σ*, a type projector* π *is a triple* $(\pi_{\mathbf{no}}, \pi_{\mathbf{olb}}, \pi_{\mathbf{eb}})$ *such that (*π *also denotes* $\pi_{\mathbf{no}} \cup \pi_{\mathbf{olb}} \cup \pi_{\mathbf{eb}}$*):*
i) $\pi \subseteq \Sigma$*,*
ii) $\pi_{\mathbf{no}}$*,* $\pi_{\mathbf{olb}}$ *and* $\pi_{\mathbf{eb}}$ *are pairwise disjoint, and*
iii) $s_D \in \pi$ *and for each* $b \in \pi$ *there exists* $a \in \pi$ *such that* $D(a) = r$ *and* b *occurs in* r*.*

The $\pi_{\mathbf{no}}$ (resp. $\pi_{\mathbf{olb}}$ and $\pi_{\mathbf{eb}}$) component of π contains 'node only' types (resp. 'one level below' and '∀ below' types). Notice that condition iii) ensures some closure property wrt to the DTD D: label $a \in \pi$ cannot be deconnected from the root label s_d although it does not need to be connected in all possible manners (see projector π_4 below). Notice that the *String* type itself never belongs to a type projector π: as explained in Section 2, a string is projected "indirectly" when its parent node type is of category 'olb' or 'eb'.

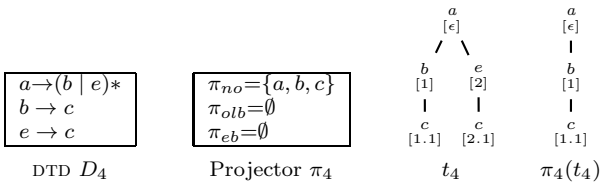


Figure 2: Type projection: an example.

DEFINITION 4.2 (TYPE PROJECTION). *Let us consider the* DTD (D, s_D)*, the type projector* $\pi = (\pi_{\mathbf{no}}, \pi_{\mathbf{olb}}, \pi_{\mathbf{eb}})$ *and the document* $t \in D$ *with* $roots(t) = \{r_t\}$ *and* $subfor(t) = F$*. The projection of* t *wrt* π*, denoted* $\pi(t)$*, is the tree* $\Pi_{K(t,\pi)}(t)$ *where* $K(t, \pi)$ *is recursively defined by:*
– *if* $lab(r_t) \notin \pi$ *then* $K(t, \pi) = \emptyset$*,*
– *if* $lab(r_t) \in \pi_\alpha$ *then* $K(t, \pi) = \{r_t\} \cup K_\alpha(F)$ *for* $\alpha \in \{\mathbf{no}, \mathbf{olb}, \mathbf{eb}\}$ *with:*
$K_\alpha(F) = \emptyset$ *if* $F = ()$ *and otherwise, assuming* $F = t' \circ F'$*,*
$K_{\mathbf{no}}(F) = K(t', \pi) \cup K_{\mathbf{no}}(F')$*,*
$K_{\mathbf{olb}}(F) = K(t', \pi) \cup K_{\mathbf{olb}}(F')$ *if* $lab(r_{t'}) \in \pi$
$K_{\mathbf{olb}}(F) = \{r_{t'}\} \cup K_{\mathbf{olb}}(F')$ *if* $lab(r_{t'}) \notin \pi$
$K_{\mathbf{eb}}(F) = dom(F)$*.*

Let us consider in Fig. 2 the DTD D_4 and the update projector π_4. This projector is well-defined: the type c is connected to the root type a by b. For the document t_4, the set $K(t_4, \pi_4)$ is $\{\epsilon, 1, 1.1\}$. Observe that, although $c \in \pi_{no}$, we have $2.1 \notin K(t_4, \pi_4)$ because $e \notin \pi_4$.

The closure property iii) of definition 4.1 entails that the result of a type projection is a well-formed tree although it may not conform to the DTD D:

PROPERTY 4.3. $\Pi_{K(t,\pi)}(t)$ *is a tree.*

The merge phase. The task of *Merge* is to build the result $u(t)$ of the update u over t starting from the initial p-tree t and the updated partial tree $u(\pi(t))$. The main assumption here is that the input document t is a p-store, implying that node identifiers correspond to node positions in the document. The function *Merge* uses this information, the 3-level projector π and nothing else. Positions are not materialized in the input document but generated at loading time (See Sec. 6).

Finally, for the purpose of insert and replace operations, it is assumed that the update u generates 'new' (not already used in t) identifiers .

The functions *Merge* and *CMerge* are formalized in Fig. 3 and Fig. 4. For the sake of simplicity, the update projector π is kept implicit in the specification.

The functions *Merge* and *CMerge* have to be thought of as mechanisms parsing in parallel two forests: F_i belonging to the initial p-tree t and F_u belonging to the updated partial tree $u(\pi(t))$; parsing synchronization is captured by the fact that the parent nodes of F_i and F_u are assumed to share the same identifier; because of projection and update, F_u contains identifiers belonging to t, besides the new ones due to insert and replace operation.

The two functions differ on the following pre-conditions:
– *Merge* assumes that (†) the parent node n of the forest F_i is of category 'node only' which implies that, because of synchronization, i) none of the top level trees in F_u is of type *String*, ii) root identifiers of top level trees in F_u belong to F_i that is $roots(F_u) \subseteq roots(F_i)$.

1	$Merge(F_i \mid F_u) =$	F_u	if $roots(F_i)=\emptyset$, **otherwise** assume $F_i=t_i \circ f_i$
2		$t_i \circ Merge(f_i \mid F_u)$	if $\sigma_{t_i}(r_{t_i})=text[st]$, **otherwise** assume $\sigma_{t_i}(r_{t_i})=a[J]$,
3		$Merge(f_i \mid F_u)$	if $a \in \pi$ and either $roots(F_u)=\emptyset$ or $F_u=t_u \circ f_u$ with $r_{t_u} > r_{t_i}$
4		$TreeMerge(t_i \mid t_u) \circ Merge(f_i \mid f_u)$	if $a \in \pi$, $F_u=t_u \circ f_u$ and $r_{t_i}=r_{t_u}$
5		$t_i \circ Merge(f_i \mid F_u)$	if $a \notin \pi$

Figure 3: The function *Merge*

c.1	$CMerge(F_i \mid F_u) =$	F_u	if $roots(F_i)=\emptyset$,
c.1′		$()$	if $roots(F_u)=\emptyset$, **otherwise** assume $F_u=t_u \circ f_u$
c.2		$t_u \circ CMerge(F_i \mid f_u)$	if $\sigma_{t_u}(r_{t_u})=text[st]$ or $new(r_{t_u})=true$, **otherwise** assume $\sigma_{t_u}(r_{t_u})=b[K]$ and $F_i=t_i \circ f_i$
c.3		$CMerge(f_i \mid F_u)$	if $\sigma_{t_i}(r_{t_i})=text[st]$ or $\sigma_{t_i}(r_{t_i})=a[J]$ with $a \in \pi$ and $r_{t_u} > r_{t_i}$
c.4		$TreeMerge(t_i \mid t_u) \circ CMerge(f_i \mid f_u)$	if $a \in \pi$, $\sigma_{t_i}(r_{t_i})=a[J]$, and $r_{t_i}=r_{t_u}$
c.5		$t_i \circ Merge(f_i \mid f_u)$	if $a \notin \pi$ and $\sigma_{t_i}(r_{t_i})=a[J]$

Figure 4: The function *CMerge*

– *CMerge* considers that (††) the node n is of category 'one level below' which implies that each node in $roots(F_i)$ has been projected and that $roots(F_u)$ are exactly the top level nodes of F_u that have to be output by *CMerge*.

The function *Merge* proceeds as follows:
Line 2 takes care of the case where the current parsed tree t_i of F_i is a *String*. The assumption † entails that it has been pruned out by π. Thus, the *String* t_i is simply output.
Line 3 deals with the case where the label a of the root r_{t_i} of t_i belongs to π (thus a subtree of t_i has been projected) and r_{t_i} does not occur in F_u (the projection of t_i has been deleted by the update). When F_u is not empty, this latter fact is identified by comparing the identifiers of the currently parsed nodes (which are positions in F_i): $r_{t_u} < r_{t_i}$ indicates that the tree t_i comes after the tree t_u in the forest F_i. Thus t_i is not output.
Line 4 takes care of synchronization on the nodes r_{t_u} and r_{t_i}: these nodes can only differ by their labels because of some renaming. In that case, the tree $TreeMerge(t_i \mid t_u)$ is output. The root of the tree $TreeMerge(t_i \mid t_u)$ is labelled by $lab(r_{t_u})$ and its sub-forest is defined by:

$$\begin{array}{ll} Merge(subfor(t_i) \mid subfor(t_u)) & if\ lab(r_{t_i}) \in \pi_{\mathbf{no}} \\ CMerge(subfor(t_i) \mid subfor(t_u)) & if\ lab(r_{t_i}) \in \pi_{\mathbf{olb}} \\ subfor(t_u) & if\ lab(r_{t_i}) \in \pi_{\mathbf{eb}} \end{array}$$

Note that, in case of $lab(r_{t_i}) \in \pi_{\mathbf{eb}}$: $TreeMerge(t_i \mid t_u) = t_u$.
Finally, line 5 deals with the case where the label a of t_i root does not belong to the projector π implying that t_i has been pruned out. Hence t_i is output.

Recall that the function *CMerge*, specified in Fig. 4, is built assuming (††). Parsing F_i and F_u in parallel is thus essentially guided by F_u, as opposed to *Merge*.
Line c.2 deals with the case where the current parsed tree t_u of F_u is either of type *String* or a newly inserted element. This latter case is identified by checking whether the identifier r_{t_u} is new ($\notin dom(t)$). Hence, the tree t_u is output. The reader may notice that no move on F_i is performed: a simple case analysis shows that synchronization is recovered through other cases.
Line c.3 is similar to line 3, although it should be paid attention to the sub-case where the root of t_i is of type *String*: t_i is then ignored because the *corresponding String* element in F_u (updated or not by u) has, eventually, already been output by a previous application of line c.2.
Lines c.4, c.5 are the dual of lines 4, 5 of the *Merge* definition. The reader should pay attention to line c.5 where, although implicit, the equality $r_{t_i}=r_{t_u}$ holds (as opposed to the case "line 7" of *Merge*): even if $a \notin \pi$, because of (††), the node identified by $r_{t_i}=r_{t_u}$ is in both forests F_i and F_u.

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 'node only' nodes, resp. to 'one level below' and '∀ 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

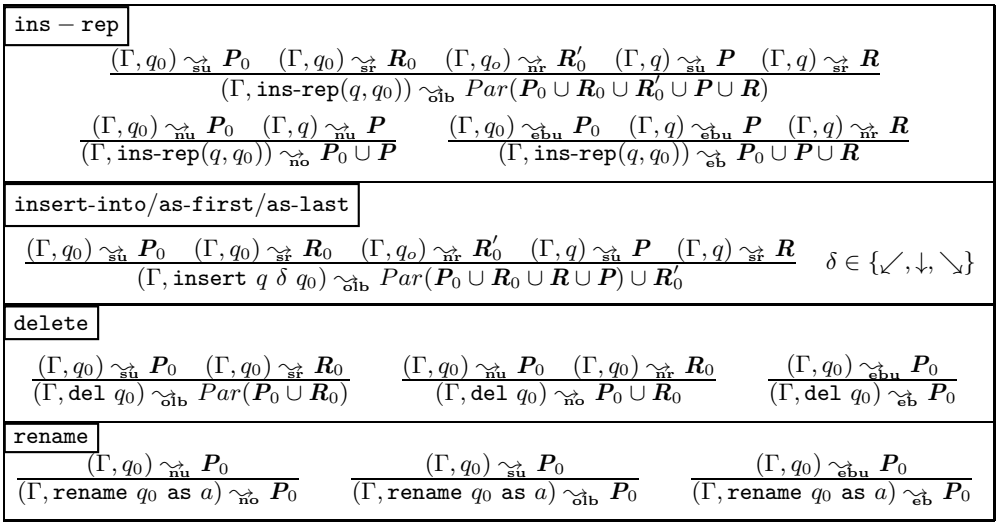


Figure 5: Path extraction for updates

the update u_3 of Sec.2: its context part contains the query /doc/a, its action part is built with the source query $x/d and the target query $x/b. As a consequence, specifying path extraction for updates requires first specifying path extraction for queries.

Our path extraction for queries generalizes [18] 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 *used* paths, we further differentiate between (i) the ones targeting nodes needed by the query (*node used paths*), (ii) the ones targeting a string (*string used paths*) and finally, (iii) the ones capturing the root nodes of elements used by the query (∀ *below used paths*). Among *returned* paths, we differentiate between those returning roots of answer elements (*node returned paths*) and those returning text (*string returned paths*).

Next, the expressions (judgments) of the form $(\Gamma, do)\rightsquigarrow_{cat} X$ should be read: given the environment Γ, the paths X of category *cat* are derived from the query/update *do*. The table below introduces judgments for query path extraction. Due to space limitation, query path extraction itself is not presented.

Category	Judgement	Category	Judgement
node returned	$(\Gamma,q)\rightsquigarrow_{\mathtt{nr}}\boldsymbol{R}$	*node used*	$(\Gamma,q)\rightsquigarrow_{\mathtt{nu}}\boldsymbol{P}$
string returned	$(\Gamma,q)\rightsquigarrow_{\mathtt{sr}}\boldsymbol{R}$	*string used*	$(\Gamma,q)\rightsquigarrow_{\mathtt{su}}\boldsymbol{P}$
		∀ *below used*	$(\Gamma,q)\rightsquigarrow_{\mathtt{ebu}}\boldsymbol{P}$

Judgements for query path extraction.

On the basis of the query path classification, we now concentrate on path extraction for updates. The three categories of paths that are going to be considered have already been introduced in Sec. 2: *node only*, *one level below* and ∀ *below*. The corresponding judgement notation are presented below while the associated rules for elementary updates are partially provided in Fig. 5. Here, $\mathtt{ins\text{-}rep}(q,q_0)$ is used to designate either an insertion of the form `insert` $q\ \delta\ q_0$ with $\delta\in\{\leftarrow,\rightarrow\}$ or a replace `replace` q_0 `with` q. Given a set of paths $\boldsymbol{P}$, $Par(\boldsymbol{P})$ denotes the set of paths $\{P/parent :: node() \mid P\in\boldsymbol{P}\}$.

Category	Judgement	Path set derived
node only	$(\Gamma,u)\rightsquigarrow_{\mathtt{no}}\boldsymbol{P}$	$\boldsymbol{P}_{\mathbf{no}}$
one level below	$(\Gamma,u)\rightsquigarrow_{\mathtt{olb}}\boldsymbol{P}$	$\boldsymbol{P}_{\mathbf{olb}}$
∀ *below*	$(\Gamma,u)\rightsquigarrow_{\mathtt{eb}}\boldsymbol{P}$	$\boldsymbol{P}_{\mathbf{eb}}$

Judgements 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 from /doc/a/b). Finally, the node returned path /doc/a/d is inferred from the insert source of u_3 leading to derive, for u_3, /doc/a/d as a ∀ *below* path.

We now present informally and not exhaustively the general analysis underlying path extraction rules for updates given in Fig. 5. 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. 6. 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 **j**=**i**). 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 ∀ *below used* query path extracted from either the context, source or target of u, then P_q is a ∀ *below* update path for u (see Frame 2).
3. Assume that P_q is a *string used* query path extracted from

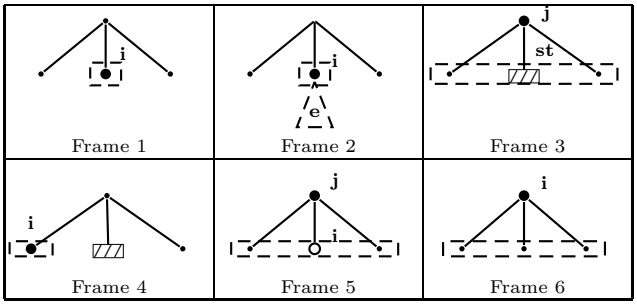


Figure 6: Update path extraction: analysis.

any environment of q, then $P_u{=}P_q/parent :: node()$ 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 $del(q)$ 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 $u{=}$`insert` $q'\ \delta\ q$ 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 $P_u{=}P_q/parent :: node()$ is derived as a *one level below* update path. The same conclusion is obtained when $u{=}$`replace` q `with` q'.

6. Finally, consider that P_q is a *node returned* query path extracted from the query q of $u{=}$`insert` $q\ \delta$ **i** with $\delta{\in}\{\swarrow,\downarrow,\searrow\}$. 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 $\boldsymbol{P}_{\mathbf{no}}$, $\boldsymbol{P}_{\mathbf{olb}}$ and $\boldsymbol{P}_{\mathbf{eb}}$ inferred for u. For each set $\boldsymbol{P}$, it produces a pair (T, C) of sets of types.

– Considering path expressions in $\boldsymbol{P}$ as (simple) queries, T collects all labels of the answer roots for these queries. Formally, for any $t{\in}D$ and $P{\in}\boldsymbol{P}$, assuming $t, ()\models P \Rightarrow t, J$:

$$\textit{if } \mathbf{i}{\in}J \textit{ and } \sigma_t(\mathbf{i}){=}a[I'] \textit{ then } a{\in}T$$

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 $\boldsymbol{P}_{\mathbf{no}}{=}\{/doc/a\}$, $\boldsymbol{P}_{\mathbf{olb}}{=}\{/doc/a\}$ and $\boldsymbol{P}_{\mathbf{eb}}{=}\{/doc/a/d\}$, we derive: $T_{\mathbf{no}}{=}\{a\}$, $T_{\mathbf{olb}}{=}\{a\}$ and $T_{\mathbf{eb}}{=}\{d\}$.

– 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 $t{\in}D$ and for $P{\in}\boldsymbol{P}$, assuming $t, ()\models P \Rightarrow t, J$:

$$\textit{if } \mathbf{i}{\in}idmatch(t, J) \textit{ and } \sigma_t(\mathbf{i}){=}a[I'] \textit{ then } a{\in}C$$

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

For the update u_3, we derive $C_{\mathbf{no}}{=}\{doc\}$, $C_{\mathbf{olb}}{=}\{doc\}$, and $C_{\mathbf{eb}}{=}\{doc, a\}$.

So type inference actually produces a pair $\Sigma{=}(T, C)$. As for query path and update path extraction, type inference rules specify the judgement $\Sigma'{\vdash_D} P : \Sigma$ where $\Sigma' = (T_c, C_c)$ is a starting environment. This judgement means that given a DTD D, starting from the labels in T_c and the context C_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 $t, ()\models P \Rightarrow t, J$ and $(\{s_D\}, \{\})\vdash_D P : (T, C)$ then:*

$$T_P \subseteq T \textit{ and } C_P \subseteq C$$

where $T_P{=}\{a \mid \mathbf{j}{\in}J \textit{ and } \sigma_t(\mathbf{j}){=}a[I']\}$, and
$C_P{=}\{a \mid \mathbf{j}{\in}idmatch(t, J) \textit{ and } \sigma_t(\mathbf{j}){=}a[I']\}$.

Type-projector inference. The final step of the update type projector derivation starts with the three pairs of type sets $(T_{\mathbf{no}}, C_{\mathbf{no}})$, $(T_{\mathbf{olb}}, C_{\mathbf{olb}})$ and $(T_{\mathbf{eb}}, C_{\mathbf{eb}})$ 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_α are all of the category 'node only'. Then, it is also obvious (from the definition of the projection) that types of category '$\forall$ 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 $\pi{=}(\pi_{\mathbf{no}}, \pi_{\mathbf{olb}}, \pi_{\mathbf{eb}})$ for u is given by:*
Let $\tau_{\mathbf{no}}{=}T_{\mathbf{no}} \cup C_{\mathbf{no}} \cup C_{\mathbf{olb}} \cup C_{\mathbf{eb}}$. Then :

$$\begin{array}{lcl} \pi_{\mathbf{no}} & = & \tau_{\mathbf{no}} - (\pi_{\mathbf{eb}} \cup \pi_{\mathbf{olb}}), \\ \pi_{\mathbf{olb}} & = & T_{\mathbf{olb}} - \pi_{\mathbf{eb}}, \textit{ and} \\ \pi_{\mathbf{eb}} & = & T_{\mathbf{eb}} \end{array}$$

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 $u_1, ..., u_n$.[1] Indeed, it suffices to generate each projector π_i for u_i and build the global projector π as the union of the π_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 π then applying successively $u_1, ..., u_n$ 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 $u_1, ..., u_n$ with the sequence of updates $u_1, ..., u_n$. Here we focus on the latter case.

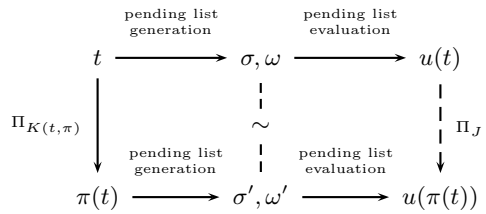


Figure 7: Soundness of the update type projector.

THEOREM 5.3. *Let u be an update over D and π be the inferred type projector for u. Then for each p-tree $t{\in}D$, we have: $Merge(t \mid u(\pi(t))) \sim u(t)$.*

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 $J{=}dom(u(t)){-}[dom(t){-}K(t,\pi)]$.*

This result corresponds to some kind of commutative diagram (see Fig. 7) 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 which include, of course, all new identifiers introduced by the update u and the identifiers used by the projection π 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 $dom(t) - K(t,\pi)$, 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 *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 5.9 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 σ and σ' 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:*
- $((),\sigma){\sim}((),\sigma')$ *always holds,*
- $(\mathbf{i}{\cdot}J,\sigma){\sim}(\mathbf{i}'{\cdot}J',\sigma')$ *iff* $(J,\sigma){\sim}(J',\sigma')$ *and*
  - $\sigma(\mathbf{i}){=}a[K]$ *implies* $\sigma(\mathbf{i}'){=}a[K']$ *and* $(K,\sigma){\sim}(K',\sigma')$,
  - $\sigma(\mathbf{i}){=}text[st]$ *implies* $\sigma(\mathbf{i}'){=}\sigma(\mathbf{i}){=}text[st]$.

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

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 σ and σ' be stores over I and I' resp. Let ω and ω' 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:*
- $(ins(J,\delta,\mathbf{i}),\sigma){\sim}(ins(J',\delta,\mathbf{j}),\sigma')$ *iff* $\mathbf{i}{=}\mathbf{j}$ *and* $(J,\sigma){\sim}(J',\sigma')$,
- $(del(\mathbf{i}),\sigma){\sim}(del(\mathbf{j}),\sigma')$ *iff* $\mathbf{i}{=}\mathbf{j}$
- $(repl(\mathbf{i},J),\sigma){\sim}(repl(\mathbf{j},J'),\sigma')$ *iff* $\mathbf{i}{=}\mathbf{j}$ *and* $(J,\sigma){\sim}(J',\sigma')$
- $(ren(\mathbf{i},a),\sigma){\sim}(ren(\mathbf{j},b),\sigma')$ *iff* $a{=}b$ *and* $\mathbf{i}{=}\mathbf{j}$

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*
- $t,()\models P \Rightarrow t,J$ *implies* $\Pi_K(t),()\models P \Rightarrow \Pi_K(t),J$ *where $K{=}K_T{\cup}K_C$, and*
- $J{\subseteq}K_T$.

Recall that the set of identifiers J in $t,()\models P \Rightarrow t,J$ 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 $idmatch(t,J)$. Here, the distinction between K_T and K_C is necessary to proceed to the right treatement 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 $t,()\models u \Rightarrow \sigma,\omega$ implies $\Pi_K(t),()\models u \Rightarrow \sigma',\omega'$ and $(\omega,\sigma){\sim}(\omega',\sigma')$.*

Of course, soundness wrt to update of a projector set is expressed based on the intermediate semantics given by update pending list. Indeed, the proof of Theorem 5.4 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 infered type projector π. 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 I). *Let us consider the id-set $\mathbf{K}{=}\mathbf{K_{no}} \cup \mathbf{K_{olb}} \cup \mathbf{K_{eb}}$ defined below. We have that $\mathbf{K}$ is a sound projector for the (pending list of the) update u on the p-tree t.*

- *Let us assume that $(\Gamma,u) \rightsquigarrow_{\mathbf{no}} \mathbf{P_{no}}$ with $\mathbf{P_{no}}{=}\{P^1_{\mathbf{no}},...,P^{k_{\mathbf{no}}}_{\mathbf{no}}\}$ and consider for $i{=}1..k_{\mathbf{no}}$, a sound projector set $(KT^i_{\mathbf{no}},KC^i_{\mathbf{no}})$ for $P^i_{\mathbf{no}}$. Then, $\mathbf{K_{no}}{=}\cup_{i=1..k_{\mathbf{no}}}(KT^i_{\mathbf{no}} \cup KC^i_{\mathbf{no}})$.*

- *Let us assume that* $(\Gamma, u) \rightsquigarrow_{\mathbf{olb}} \mathbf{P_{olb}}$ *with* $\mathbf{P_{olb}}=\{P^1_{\mathbf{olb}}, ..., P^{k_{\mathbf{olb}}}_{\mathbf{olb}}\}$ *and consider for* $i=1..k_{\mathbf{olb}}$, *a sound projector set* $(KT^i_{\mathbf{olb}}, KC^i_{\mathbf{olb}})$ *for* $P^i_{\mathbf{olb}}$. *Then,* $\mathbf{K_{olb}}=\cup_{i=1..k_{\mathbf{olb}}} (KT^i_{\mathbf{olb}} \cup KC^i_{\mathbf{olb}} \cup child(t, KT^i_{\mathbf{olb}}))$
- *Let us assume that* $(\Gamma, u) \rightsquigarrow_{\mathbf{eb}} \mathbf{P_{eb}}$ *with* $\mathbf{P_{eb}}=\{P^1_{\mathbf{eb}}, ..., P^{k_{\mathbf{eb}}}_{\mathbf{eb}}\}$ *and consider for* $i=1..k_{\mathbf{eb}}$, *a sound projector set* $(KT^i_{\mathbf{eb}}, KC^i_{\mathbf{eb}})$ *for* $KP^i_{\mathbf{eb}}$. *Then,* $\mathbf{K_{eb}}=\cup_{i=1..k_{\mathbf{eb}}} (KT^i_{\mathbf{eb}} \cup KC^i_{\mathbf{eb}} \cup desc(t, KT^i_{\mathbf{eb}}))$

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* $\mathbf{K}=\mathbf{K_{no}} \cup \mathbf{K_{olb}} \cup \mathbf{K_{eb}}$ *defined below. We have that* $\mathbf{K}$ *is a sound projector for the (pending list of the) update* u *on the p-tree* t.

- *Let us assume that* $(\Gamma, u) \rightsquigarrow_{\mathbf{no}} \mathbf{P_{no}}$ *with* $\mathbf{P_{no}}=\{P^1_{\mathbf{no}}, ..., P^{k_{\mathbf{no}}}_{\mathbf{no}}\}$ *and for* $i=1..k_{\mathbf{no}}$, $t, () \models P^i_{\mathbf{no}} \Rightarrow t, J^i_{\mathbf{no}}$. *Then,* $\mathbf{K_{no}}=\cup_{i=1..k_{\mathbf{no}}} (J^i_{\mathbf{no}} \cup idmatch(t, J^i_{\mathbf{olb}})$.
- *Let us assume that* $(\Gamma, u) \rightsquigarrow_{\mathbf{olb}} \mathbf{P_{olb}}$ *with* $\mathbf{P_{olb}}=\{P^1_{\mathbf{olb}}, ..., P^{k_{\mathbf{olb}}}_{\mathbf{olb}}\}$ *and for* $i=1..k_{\mathbf{olb}}$, $t, () \models P^i_{\mathbf{olb}} \Rightarrow t, J^i_{\mathbf{olb}}$. *Then,* $\mathbf{K_{olb}}=\cup_{i=1..k_{\mathbf{olb}}} (J^i_{\mathbf{olb}} \cup idmatch(t, J^i_{\mathbf{olb}}) \cup child(t, J^i_{\mathbf{olb}}))$
- *Let us assume that* $(\Gamma, u) \rightsquigarrow_{\mathbf{eb}} \mathbf{P_{eb}}$ *with* $\mathbf{P_{eb}}=\{P^1_{\mathbf{eb}}, ..., P^{k_{\mathbf{eb}}}_{\mathbf{eb}}\}$ *and for* $i = 1..k_{\mathbf{eb}}$, $t, () \models P^i_{\mathbf{eb}} \Rightarrow t, J^i_{\mathbf{eb}}$. *Then,* $\mathbf{K_{eb}}=\cup_{i=1..k_{\mathbf{eb}}} (J^i_{\mathbf{eb}} \cup idmatch(t, J^i_{\mathbf{eb}}) \cup desc(t, J^i_{\mathbf{eb}}))$

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 5.3. To conclude this section, we would like to highlight that the proof of Theorem 5.3, and more precisely the part showing that *Merge* builds the updated document $u(t)$ uses the fact that the type projector π is initially applied over a p-tree. Once again, p-tree are specified such that node identifiers correspond to node positions. Intuitively, lemma 5.9 expresses that the update pending list generated for the projected document $\Pi_{K(t,\pi)}(t)$ 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 *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 *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 [6]. 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.

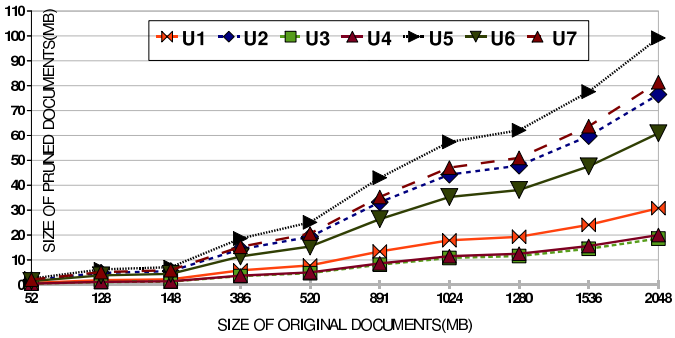


Figure 8: Documents size reduction after pruning

The first kind of tests aims at detecting memory limitations of four popular query processors implemented in Java: Saxon EE 9.2.0.2 [7], QizX Free-Engine-3.2.0 [4], eXist 1.2.5 [1] and MXQuery 0.6.0 [3]. 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 evaluates our projection based technique. We focused on two systems Saxon and QizX, and used the whole set of 7 updates. In both cases, tests

	Saxon	QizX F-E	eXist	MXQuery
MB	128	580	148	52

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. 8). 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.

QizX 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 QizX, projection also ensures sensible total execution time reduction. This is in part due to the fact that QizX 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 5), 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 QizX this took, respectively, 82 and 64 seconds on the 128MB document. For this document, the sum of total times needed to projecting, updating and merging for each single update was much higher, respectively 181 and 194 seconds for Saxon and QizX.

The updates and the corresponding projectors.

```
U1.  for $x in $doc/site/closed_auctions/closed_auction
where not ($x/annotation) return
insert node <annotation>Empty Annotation</annotation>
as last into $x

U2.for $x in $doc/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>)
```

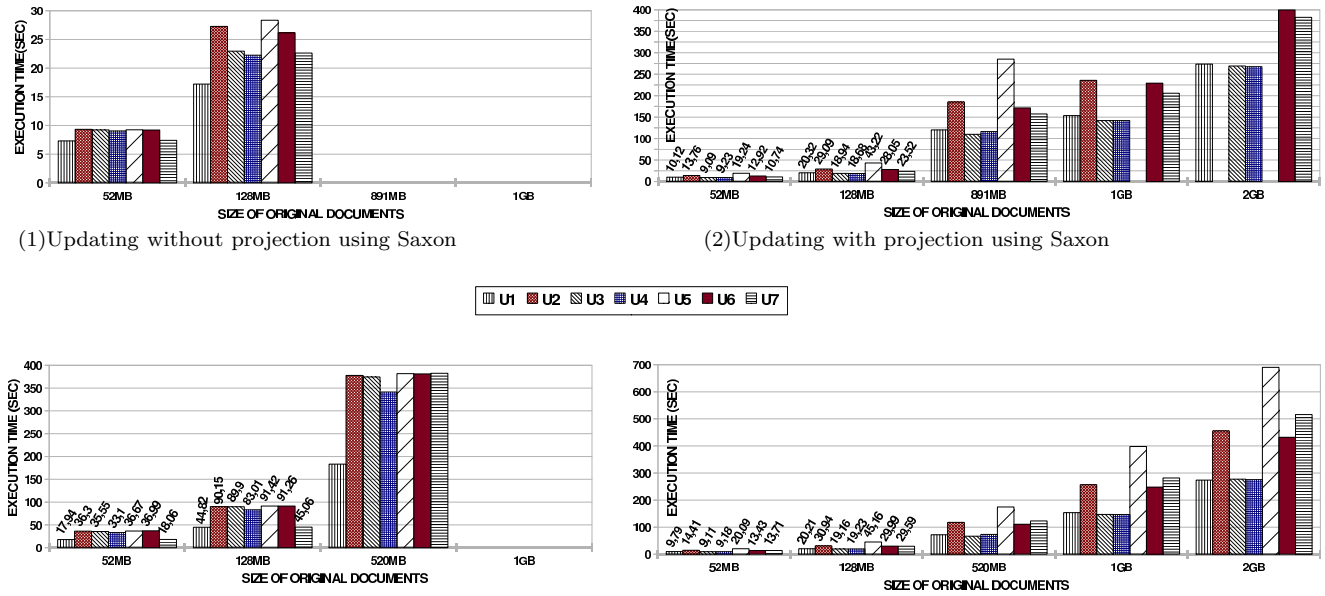
	π_{no}	π_{olb}	π_{eb}
U1	site, closed_auctions, annotation	closed_auct.	$\emptyset$
U2	site, people, address	person, country, street, province, zipcode	$\emptyset$
U3	site, regions, africa, asia, australia, europe, namerica, samerica, item	location	$\emptyset$
U4	site, regions, africa, asia, australia, europe, namerica, samerica, item, mailbox, mail	$\emptyset$	$\emptyset$
U5	site, regions, africa, asia, australia, europe, namerica, samerica, listitem, bold, mailbox, mail, item, description, text, open_auctions, open_auction closed_auctions, closed_auction, annotation, parlist	$\emptyset$	$\emptyset$
U6	site, people, homepage	person, name	$\emptyset$
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
    rename  node $x as "emph"

U6.for $x in $doc/site/people/person
    where not($x/homepage)
    return insert node
    <homepage>www.{$x/name/text()}Page.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. CONLCUSIONS 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



(1)Updating without projection using Saxon

(2)Updating with projection using Saxon

(3)Updating without projection using Qizx

(4)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

[1] eXist. `http://exist.sourceforge.net/`.
[2] Galax. `http://www.galaxquery.org`.
[3] MXquery. http://mxquery.org/.
[4] QizX Free-Engine-3.0. `http://www.xmlmind.com/qizx/free_engine.html`.
[5] QizX/open. `http://www.xmlmind.com/qizx/qizxopen.shtml`.
[6] SAX. http://www.saxproject.org/.
[7] Saxon-ee. `http://www.saxonica.com/`.
[8] M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Verification of tree updates for optimization. In *CAV*, 2005.
[9] M. Benedikt and J. Cheney. Schema-based independence analysis for XML updates. *VLDB*, 2009.
[10] M. Benedikt and J. Cheney. Semantics, types and effects for XML updates. In *DBPL*. Springer, 2009.
[11] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. Type-based XML projection. In *VLDB*, 2006.
[12] S. Bressan, B. Catania, Z. Lacroix, Y.-G. Li, and A. Maddalena. Accelerating queries by pruning XML documents. *Data Knowl. Eng.*, 54(2), 2005.
[13] W. Fan, G. Cong, and P. Bohannon. Querying XML with update syntax. In *SIGMOD Conference*, 2007.
[14] L. Fegaras. A schema-based translation of XQuery updates. In *XSym*, 2010.
[15] W. Gelade, W. Martens, and F. Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. In *ICDT*, 2007.
[16] G. Ghelli, K. H. Rose, and J. Siméon. Commutativity analysis in XML update languages. In *ICDT*, 2007.
[17] G. Ghelli, K. H. Rose, and J. Siméon. Commutativity analysis for XML updates. *ACM Trans. Database Syst.*, 33(4), 2008.
[18] A. Marian and J. Siméon. Projecting XML documents. In *VLDB*, 2003.
[19] M. Schmidt, S. Scherzinger, and C. Koch. Combined static and dynamic analysis for effective buffer minimization in streaming XQuery evaluation. In *ICDE*, 2007.