Reverting the Effects of XQuery Update Expressions

Federico Cavalieri[1], Giovanna Guerrini[1], and Marco Mesiti[2]

[1] DISI – University of Genova
{cavalieri,guerrini}@disi.unige.it
[2] DICo – University of Milano
mesiti@dico.unimi.it

Abstract. The need of reverting the effects of updates on the affected documents arises in many contexts, ranging from undos in transactional applications to versioning systems. In this paper, we investigate this issue for XQuery Update expressions, relying on the Pending Update List (PUL) obtained from the evaluation of an expression on a document. Specifically, we introduce an inversion operator, that, given a PUL to be applied on a document, allows to determine a corresponding inverted PUL that, applied on the modified document, produces the original document. Moreover, an alternative approach for enriching a PUL with additional information, so that it can be inversely applied, is proposed and the two approaches are experimentally compared.

Keywords: XML, Updates, Dynamic reasoning, Update processing.

1 Introduction

The ability of reverting the effects of an update operation is useful in many situations. Consider for instance a distributed transactional context, in which transactions can be aborted and rolled back, and thus the corresponding operations need to be undone. More flexible update processing approaches based on check-in/check-out policies, such as those employed in collaborative document editing, may benefit as well of such ability. It may also become crucial in versioning contexts, if versions are handled by recording the updates that transformed a version into the following one (edit-based approaches according to [4]) instead of the various data snapshots. In this context, update reversion is the basis for moving across different versions.

In this paper, we investigate this problem in the context of updates on XML documents expressed as XQuery Update (XQU) expressions [15]. The evaluation of an XQU expression on a document produces a set of atomic update requests, represented as a Pending Update List (PUL), that is then applied on the document. In [3] we discussed the relevance of contexts (such as collaborative editing, disconnected execution, data clouds) in which updates are not necessarily executed right after and on the same server where the update expression requesting them is evaluated. Thus, the process of expressing and requesting updates is decoupled from that of making them effective on documents. PULs can be produced by a machine, sent over a network, saved to disks, and later applied on the document, possibly by a different machine. Referring to update reverting, this means that the effects of an update expression can be discarded on a server different from the one on which they have been applied.

A.A.A. Fernandes et al. (Eds.): BNCOD 2011, LNCS 7051, pp. 167–181, 2011.
© Springer-Verlag Berlin Heidelberg 2011

According to this processing model, there are two alternative approaches to revert the effects of an update expression. The first one is to invert a PUL through a PUL inversion operator. This operator, given a PUL Δ on a document D, produces another PUL Δ^{-1}, that, when executed on the document updated according to Δ, returns the original document D, thus undoing the updates in Δ. This approach however requires to access document D for producing the inverse. An important feature of the PUL operators in [3], by contrast, is document independence: operators on PULs should not require, whenever possible, to access the document. They rely on structural information on the document that is incorporated in the PUL itself through a labeling scheme. Thus, an alternative approach, to be exploited in contexts where the need of reverting update effects is known to be likely to arise, is to modify the evaluation of XQU expressions so that they produce *completed PULs* rather than PULs. A completed PUL $\Delta^{\leftrightarrow}$ contains the information for being applied either forward (to actually apply updates) or backward (to revert their effects), and in both cases it can be applied in streaming.

The paper investigates both approaches, proposing a set of inversion rules and a PUL inversion algorithm, defining completed PULs, and discussing their forward and backward streaming application. Both approaches have been implemented, by modifying the Qizx [14] library to produce PULs and completed PULs (both represented as XML documents) and to accept them as input.

The paper is organised as follows. Section 2 introduces some preliminary notions on XML documents and PULs. PUL inversion is discussed in Section 3, whereas Section 4 introduces completed PULs and their backward and forward application. Section 5 experimentally compares the two approaches. Section 6 contrasts our approach with related work. Some concluding remarks are finally presented.

2 Preliminaries

In this section we introduce the adopted representations of XML documents, define PULs of operations, their semantics, and discuss their streaming evaluation.

2.1 XML Document Representations

A document can be represented as a labeled tree. A document D is a tuple $(V, \gamma, \lambda, \nu)$ where: V is a set of nodes representing elements, attributes or text nodes (for simplicity, only these types among those in [15] are considered); γ is a function associating with each node its children; λ and ν are labeling functions associating with each element and attribute node a name in a set $\mathcal{N}$ and with each text and attribute node a value in a set $\mathcal{V}$, respectively. Auxiliary functions V and $\mathcal{R}$ denote the nodes and the root(s) of a single tree D or of a collection of trees, respectively, and τ assigns to each node v in V a value in the set $\{\mathtt{e}, \mathtt{a}, \mathtt{t}\}$ denoting its type. Moreover, auxiliary functions LS, P and T assign to each node its adjacent left sibling, parent and subtree, respectively, when they exists ($\perp$, otherwise). Coherently with the XDM model, the attribute value is seen as a property of the attribute node, whereas textual contents of elements are modeled by separate nodes. A unique and immutable identifier is associated with each node in V, and, wherever no confusion arises, we do not distinguish nodes from their identifiers.

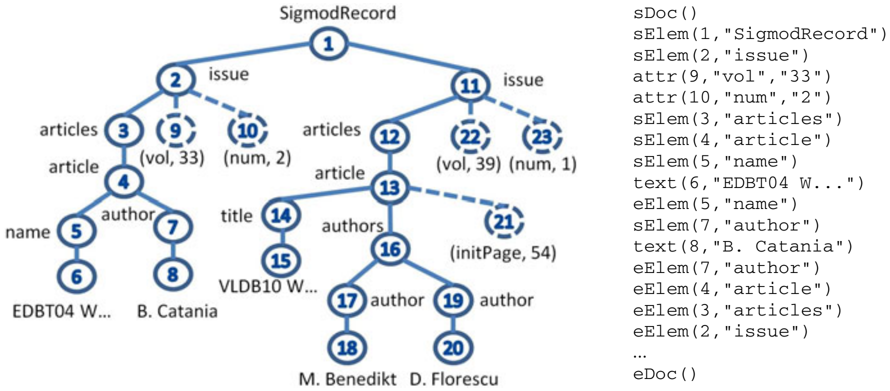

Fig. 1. XML Document representation: tree-based (left), event-based (right)

Alternatively, a document can be represented as a stream of SAX-like events, which describe the nodes encountered in a depth-first traversal of the document. A document D is an ordered sequence of events: `sDoc`() and `eDoc`(), indicating the start/end of the document; `sElem`(v,n) and `eElem`(v,n), indicating the start/end of element v tagged n; `attr`(v,n,s) and `text`(v,s), indicating the attribute/text node $v$with name n and value s. Fig. 1 contrasts the tree/event-based representations of a fragment of the `SigmodRecord` document to be updated. In the former, dotted lines are used to represent edges leading to attribute nodes.

In handling PULs, we need to check whether some relationships (like parent-child, element-attribute, left-right sibling) hold among two nodes in both representations. This information is obtained without directly accessing the document through a labeling scheme [12] associated with nodes through function l. In this scheme, the introduction/deletion of nodes does not require node re-labelling. Table 1 reports the predicates that can be assessed through the adopted labeling scheme.

2.2 Update Operations and PULs

Table 2 reports the primitives defined in [15] that we consider, where $v \in V$ is a node, $P = [T_1, \ldots, T_k]$, $k \geq 0$, is a list (possibly empty in case of the `repN` or `repC` operation) of trees, $n \in \mathcal{N}$ is a name, $s \in \mathcal{V}$ is a value. Given an operation op, $t(op)$ denotes its target node, $o(op)$ denotes its name, and $p(op)$ denotes its second parameter

Table 1. Structural relationships

Predicate	Description
$v_1 \lhd v_2$	v_1 precedes v_2 in document order
$v_1 \lhd_s v_2$	v_1 is the (adjacent) left sibling of v_2
$v_1 \blacktriangleleft_s v_2$	v_1 is a preceding sibling of v_2
$v_1 /_c v_2$	v_1 is a child of v_2
$v_1 /_a v_2$	v_1 is an attribute of v_2
$v_1 /\!/_d v_2$	v_1 is a descendant of v_2
$v_1 /\!/_d^{\neg a} v_2$	v_1 is a descendant of v_2 but not an attribute of v_1

Table 2. Update operations

Operation	Description	Conditions	Completed Operation
$\mathtt{ins}^{\leftarrow}(v, P)$ $\mathtt{ins}^{\rightarrow}(v, P)$	Insert the trees in P before/after node v	$\tau(v) \neq \mathtt{a}, \forall r \in \mathcal{R}(P)\ \tau(r) \neq \mathtt{a}$	idem
$\mathtt{ins}^{\swarrow}(v, P)$ $\mathtt{ins}^{\searrow}(v, P)$	Insert the trees in P as first/last children of node v	$\tau(v) = \mathtt{e}, \forall r \in \mathcal{R}(P)\ \tau(r) \neq \mathtt{a}$	idem
$\mathtt{ins}^{\downarrow}(v, P)$	Inserts the trees in P as children of node v, in an implementation defined position	$\tau(v) = \mathtt{e}, \forall r \in \mathcal{R}(P)\ \tau(r) \neq \mathtt{a}$	idem
$\mathtt{insA}(v, P)$	Inserts the trees in P as attributes of node v	$\tau(v) = \mathtt{e}, \forall r \in \mathcal{R}(P)\ \tau(r) = \mathtt{a}$	idem
$\mathtt{del}(v)$	Deletes node v		$\mathtt{del}(v, T(v))$
$\mathtt{repN}(v, P)$	Replaces node v with the trees in P (possibly none)	$\forall r \in \mathcal{R}(P)\ (\tau(r) = \tau(v) = \mathtt{a}) \vee (\tau(v) \neq \mathtt{a} \wedge \tau(r) \neq \mathtt{a})$	$\mathtt{repN}(v, P, T(v))$
$\mathtt{repV}(v, s)$	Replaces the value of node v with $s \in \mathcal{V}$	$\tau(v) \in \{\mathtt{t}, \mathtt{a}\}$	$\mathtt{repV}(v, s, \nu(v))$
$\mathtt{repC}(v, v')$	Replaces the children of node v with text node v' or with nothing	$\tau(v) = \mathtt{e} \wedge (v' = [] \vee \tau(v') = \mathtt{t})$	$\mathtt{repC}(v, v', T(\gamma(v)))$
$\mathtt{ren}(v, n)$	Renames node v with $n \in \mathcal{N}$	$\tau(v) \in \{\mathtt{e}, \mathtt{a}\}$	$\mathtt{ren}(v, n, \lambda(v))$

(undefined if $o(op) = \mathtt{del}$). An operation *op* is applicable on a document D if its target belongs to D and the applicability conditions (identified in [3]) of *op* hold. The meaning of last column will be discussed in Section 4.

A *pending update list* (*PUL*) [15] is an unordered list of operations among those in Table 2. Since the order of operations is irrelevant, some pairs of operations cannot occur in the same PUL. Specifically, no pairs of replacement operations of the same type with the same target (referred to as *incompatible operations*) can occur. For a PUL to be applicable on a document (cf. function `applyUpdates` in [15]) it must contain no incompatible operations and all its operations must be applicable on the document.

Operation semantics is specified in [1]. The semantics of operation $\mathtt{ins}^{\downarrow}$ is non-deterministic since the actual position of the inserted nodes group in the target node is not univocally specified. Thus, the application of an operation *op* to a document D produces one document in a set of *obtainable documents*, denoted as $\mathcal{O}(op, D)$.

The semantics of a PUL Δ on a document D is obtained by applying the operations in Δ in five stages [1]. At each stage, a subset of the operations are applied to enforce the precedence relation on operation types specified in [15]. The order of application of operations within each stage is not prescribed by [15]. Thus, when multiple insertion operations of the same type with the same target appear in the same PUL, the relative order of the inserted groups of nodes is not fixed as well. Therefore, the cardinality of $\mathcal{O}(\Delta, D)$ is greater than one when $\mathtt{ins}^{\downarrow}$ occurs in Δ or Δ contains more than one insertion operation of the same type on the same target.

Example 1. Let D be the document in Fig. 1. Operation $op_1 = \mathtt{del(14)}$ is deterministic and thus $\mathcal{O}(op_1, D)$ is a singleton. Operation $op_2 =$ ins$^{\downarrow}$(16, <author>G.Guerrini</author>), by contrast, may lead to inserting the element as first, second, or last author of the second paper, thus $\mathcal{O}(op_2, D)$ contains three documents. Finally, $|\mathcal{O}(\Delta, D)| = 6$ for $\Delta =$ {ins$^{\downarrow}$(16, <author>G.Guerrini</author>), ins$^{\searrow}$(4, <initP>132</initP>), ins$^{\searrow}$(4, <lastP>134</lastP>)}.

2.3 PUL Streaming Application

Given a document represented as a sequence of events E, a PUL is modeled as an *event transformer*, which transforms E in a new sequence of events corresponding to one of the obtainable documents. An empty PUL corresponds to an identity transformation. Each operation in a non-empty PUL requires some event transformation. The order in which transformations must be applied on the events of a node v is the same as in the staged execution of an XQU expression.

Example 2. Consider the PUL Δ ={repC(2, []), ren(2, "issues")} and the document in Fig. 1. The first operation requires to remove any non-attribute event which occurs between sElem(2,_) and eElem(2,_). The second one requires to alter these two events replacing the name. The transformed event sequence is: sDoc(), sElem(1, "SigmodRecord"), sElem(2, "issues"), attr(9, volume, "33"), attr(10, number, "2"), eElem(2, "issues"), ..., eDoc().

3 PUL Inverse

In this section we discuss the inversion of operations and PULs.

3.1 Operation Inversion

The inversion of an operation *op* applicable on a document D produces one or more operations that revert the modifications made by *op* on D. To revert the effects of a single operation, multiple operations may be required. Consider the case of a list of subtrees inserted through a single ins: each single subtree must be separatedly deleted.

Definition 1 (Operation Inverse). *Let op be an operation applicable on a document D. An inverse of op on D is a PUL* Δ_{op}^{-1} *s.t:* $\forall D' \in \mathcal{O}(op, D) : \mathcal{O}(\Delta_{op}^{-1}, D') = \{D\}$.

Different PULs can be identified to revert the effects of each operation. The following approach has been devised: ins is reverted by removing all inserted subtrees, repV/ren by restoring the original value/name of the target node, repC by restoring the original node children. For node deletions and replacements the introduced nodes must be deleted, while removed nodes must be placed back in their original position. Attribute nodes are inserted by an insA, while other nodes through $\texttt{ins}^{\rightarrow}$, if an adjacent left sibling exists in D, through an $\texttt{ins}^{\swarrow}$ otherwise. Table 3 defines inverse operations.

Example 3. Consider the document in Fig. 1 and the following operations (the node identifier is reported as superscript of the node). op_1 =ren(5, "title"), op_2 =del(7), op_3 = ins(4, <author>X^{25}</author>^{24}, <author>Y^{27}</author>^{26}). The inverses are: $\Delta_{op_1}^{-1}$={ren(5, "name")}, $\Delta_{op_2}^{-1}$ ={$\texttt{ins}^{\rightarrow}$(5, <author>B.Catania^{8}</author>^{7})}, $\Delta_{op_3}^{-1}$ ={del(24), del(26)}, respectively.

Proposition 1 (Correctness of Operation Inversion). *For any operation op applicable on a document D, let* Δ_{op}^{-1} *be the inverse PUL obtained according to Table 3. Then* Δ_{op}^{-1} *is an inverse of op on D, according to Definition 1.*

Proof Sketch. This can be straightforwardly proved by individually considering the semantics of each operation and of the corresponding inverse as defined in Table 3.

Table 3. Operation inverses

Operation	Inverse	Condition
$\texttt{ins}^{r}(v, [T_1, ..., T_n])$	$\{\texttt{del}(\mathcal{R}(T_1)), ..., \texttt{del}(\mathcal{R}(T_n))\}$	$r \in \{\leftarrow, \rightarrow, \downarrow, \swarrow, \searrow, \texttt{A}\}$
$\texttt{repV}(v, s)$	$\{\texttt{repV}(v, \nu(v))\}$	
$\texttt{ren}(v, l)$	$\{\texttt{ren}(v, \lambda(v))\}$	
$\texttt{repC}(v, v')$	$\{\texttt{repN}(v', \gamma(v))\}$ $\{\texttt{ins}^{\downarrow}(v, \gamma(v))\}$	$v' \neq []$ $v' = []$
$\texttt{del}(v)/\texttt{repN}(v, [])$	$\{\texttt{insA}(P(v), T(v))\}$ $\{\texttt{ins}^{\rightarrow}(LS(v), T(v))\}$ $\{\texttt{ins}^{\swarrow}(P(v), T(v))\}$	$\tau(v) = \texttt{a}$ $\tau(v) \neq \texttt{a} \wedge LS(v) \neq \perp$ $\tau(v) \neq \texttt{a} \wedge LS(v) = \perp$
$\texttt{repN}(v, [T_1, ..., T_n])$	$\{\texttt{repN}(\mathcal{R}(T_1), T(v)), \texttt{del}(\mathcal{R}(T_2)), ..., \texttt{del}(\mathcal{R}(T_n))\}$	

3.2 PUL Inversion

Starting from the operation inverse, we define the PUL inverse.

Definition 2 (PUL Inverse). *Let Δ be a PUL applicable on a document D. An inverse of Δ on D is a PUL Δ^{-1} s.t. $\forall D' \in \mathcal{O}(\Delta, D) : \mathcal{O}(\Delta^{-1}, D') = \{D\}$.*

In inverting a PUL Δ, the following properties must be guaranteed: (*i*) each inverse operation must be applicable in all the obtainable documents; (*ii*) in case an operation in Δ is overridden (that is, it has no effect on the document), its inverse must have no effect; and, finally, (*iii*) the relative order of the nodes removed by Δ must be restored by its inverse. As shown by the following example, simply inverting each single operation in Δ independently does not allow to guarantee these properties.

Example 4. Consider the PUL $\Delta = \{$del(5), repV(6, "VLDB04"), repN(7, <author>X^{25} </author>^{24}$\}$ applicable on the document in Fig. 1. The inverse of Δ, obtained by inverting each single operation independently is $\Delta^{-1} = \{op_1, op_2, op_3, op_4\}$ where op_1 =$\texttt{ins}^{\swarrow}$(4, <name> EDBT04 W...^{6} </name>^{5}), op_2 =repV(6, "EDBT04 W..."), op_3 =del(24), op_4 =$\texttt{ins}^{\swarrow}$(4, <author>B.Catania^{8} </author>^{7})}. Δ^{-1} exhibits two problems: (*i*) the overridden operation repV(6, "VLDB04") has been inverted as op_2, that is both unnecessary (the value of node 6 is restored by op_1) and not applicable, as node 6 does not belong to any document $D' \in \mathcal{O}(\Delta, D)$; (*ii*) the order of the restored nodes 5 and 7 may not be the one in the original document.

To guarantee the first two properties, overridden operations in Δ should be removed. For the last property, special treatment should be devoted to operations `del` and `repN` when their targets are adjacent siblings. In this case, insertion operations of the same kind with the same target might not preserve the insertion order. To obtain the correct order, `repN` and `del` operations on adjacent siblings should be grouped together.

Definition 3 (Removal Group). *Given a PUL Δ, we denote as a removal group of Δ a non-empty ordered sublist $S = [op_1, ..., op_n]$ of Δ s.t.*

- $o(op_i) \in \{\texttt{repN}, \texttt{del}\}, 1 \leq i \leq n$ *and* $t(op_1) \lhd_s t(op_2) \lhd_s ... \lhd_s t(op_n)$,
- $\{op_1, .., op_n\}$ *is maximal, that is,* $\nexists S' \supset \{op_1, .., op_n\}$ *s.t.* S' *is a removal group of* Δ.

Example 5. Consider the PUL Δ ={del(7), ren(5, "title"), repN(5, <a>X), del(11)} on the document in Fig. 1. The removal groups are [repN(5, <a>X), del(7)] and [del(11)].

O1) $$\frac{op_1 = op(v,_)\ \ op_2 = op'(v,_)}{\Delta \cup \{op_1, op_2\} \rightarrow_1 \Delta \cup \{op_2\}} \quad \begin{array}{l} op \ \in \{\texttt{ren}, \texttt{repV}, \texttt{repC}, \texttt{del}, \texttt{ins}^{\swarrow}, \texttt{ins}^{\searrow}, \texttt{ins}^{\downarrow}, \texttt{insA}\} \\ op' \ \in \{\texttt{repN}, \texttt{del}\} \end{array}$$

O2) $$\frac{op_1 = op(v,_)\ op_2 = \texttt{repC}(v,_)}{\Delta \cup \{op_1, op_2\} \rightarrow_1 \Delta \cup \{op_2\}} \quad op \in \{\texttt{ins}^{\swarrow}, \texttt{ins}^{\downarrow}, \texttt{ins}^{\searrow}\}$$

O3) $$\frac{op_1 = op(v,_) \quad op_2 = op'(v',_)}{\Delta \cup \{op_1, op_2\} \rightarrow_1 \Delta \cup \{op_2\}} \quad op' \in \{\texttt{repN}, \texttt{del}\},\ v \mathbin{/\!\!/_d} v'$$

O4) $$\frac{op_1 = op(v,_) \quad op_2 = \texttt{repC}(v',)}{\Delta \cup \{op_1, op_2\} \rightarrow_1 \Delta \cup \{op_2\}} \quad v \mathbin{/\!\!/_d^{\neg a}} v'$$

S5) $$\frac{op = \texttt{ins}^r(v, [T_1, ..., T_n]) \quad \Delta' = \{\texttt{del}(\mathcal{R}(T_1)), ..., \texttt{del}(\mathcal{R}(T_n))\}}{\Delta \cup \{op\}, \Delta^{-1} \rightarrow_2 \Delta, \Delta^{-1} \cup \Delta'} \quad r \in \{\leftarrow, \rightarrow, \downarrow, \swarrow, \searrow, A\}$$

S6) $$\frac{op = \texttt{repC}(v, t) \quad op' = \begin{cases} \texttt{repN}(t, \gamma(v)) \text{ if } t \neq [] \\ \texttt{ins}^{\downarrow}(v, \gamma(v)) \text{ otherwise} \end{cases}}{\Delta \cup \{op\}, \Delta^{-1} \rightarrow_2 \Delta, \Delta^{-1} \cup \{op'\}}$$

S7) $$\frac{op = \texttt{repV}(v, s) \quad op' = \texttt{repV}(v, \nu(v))}{\Delta \cup \{op\}, \Delta^{-1} \rightarrow_2 \Delta, \Delta^{-1} \cup \{op'\}}$$

S8) $$\frac{op = \texttt{ren}(v, n) \quad op' = \texttt{ren}(v, \lambda(v))}{\Delta \cup \{op\}, \Delta^{-1} \rightarrow_2 \Delta, \Delta^{-1} \cup \{op'\}}$$

S9) $$\frac{\Delta' = \{\texttt{insA}(P(v), T(v))\} \cup \{\texttt{del}(\mathcal{R}(u)) |\ u \in p(op)\}}{\Delta \cup \{op\}, \Delta^{-1} \rightarrow_2 \Delta, \Delta^{-1} \cup \Delta'} \quad t(op) = v, \tau(v) = \mathtt{a}, o(op) \in \{\texttt{repN}, \texttt{del}\}$$

G10) $$\frac{\begin{array}{c}\Delta' = \{\texttt{ins}^{\rightarrow}(LS(v_1), [T(v_1), ..., T(v_n)]\} \cup \\ \{\texttt{del}(u) |\ u \in \bigcup_{i=1..n} \mathcal{R}(p(op_i))\}\end{array}}{\Delta \cup \{op_1, ..., op_n\}, \Delta^{-1} \rightarrow_2 \Delta, \Delta^{-1} \cup \Delta'} \quad \begin{array}{l} \{v_1, \ldots, v_n\} \text{ are the operation targets,} \\ [op_1, \ldots, op_n] \text{ is a removal group,} \\ \tau(v_1) \neq \mathtt{a}, LS(v_1) \neq \bot \end{array}$$

G11) $$\frac{\begin{array}{c}\Delta' = \{\texttt{ins}^{\swarrow}(P(v_1), [T(v_1), ..., T(v_n)]\} \cup \\ \{\texttt{del}(u) |\ u \in \bigcup_{i=1..n} \mathcal{R}(p(op_i))\}\end{array}}{\Delta \cup \{op_1, ..., op_n\}, \Delta^{-1} \rightarrow_2 \Delta, \Delta^{-1} \cup \Delta'} \quad \begin{array}{l} \{v_1, ..., v_n\} \text{are the operation targets,} \\ [op_1, ..., op_n] \text{ is a removal group,} \\ \tau(v_1) \neq \mathtt{a}, LS(v_1) = \bot \end{array}$$

Fig. 2. Inversion rules

The inversion of a PUL Δ is computed through the set of inversion rules in Fig. 2. Rules are categorized in the following three classes:

- O Rules that remove overridden operations. Specifically, rule O1 and O3 consider the case in which a `repN` or a `del` operation on a node v overrides other operations in the subtree rooted at v. Rules O2 and O4, are similar in purpose, but consider the case in which a `repC` is the overriding operation. In all cases, rules in this class maintain the `repN`, `repC` , or `del` operation and removes the overridden operation.
- S Rules that compute the inverses of `ins`, `repC`, `repV`, `ren`, as specified in Table 3.
- G Rules that compute the inverse of a removal group. Specifically, rules G10 and G11 produce a deletion for each introduced node and a single insertion for all removed nodes ensuring the restoration of their original order.

A basic algorithm for computing the inversion consists in the application of the rules in two stages. When rules in the first stage cannot be applied any more, rules of the second stage are considered. The rules of class O are considered in the first stage to remove overridden operations from Δ. The rules of classes S and G are considered in the second stage on a pair of PULs, Δ and Δ^{-1}, the PUL to invert and the inverted PUL (initially empty). The application of a rule of classes S and G removes one or more operations from Δ and introduces the corresponding inverse operation(s) in Δ^{-1}.

Example 6. Consider the PUL $\Delta = \{\texttt{del}(5), \texttt{repV}(6, \text{“}\texttt{VLDB04}\text{”}), \texttt{repN}(7, \texttt{<author>X}^{25}\texttt{</author>}^{24}\}$ of Example 4. Its inverse, obtained through the application of rules in Fig. 2 is $\Delta^{-1} =$

{ins$^{\swarrow}$(4, <name>EDBT04 W...^{6}</name>^{5} <author>B.Catania^8</author>^{7}), del(24)}. In this case, differently from what happened in Example 4, the overridden operation repV(6, "VLDB04") has not been inverted and the order of the restored nodes 5 and 7 is preserved.

Proposition 2 (Correctness of the Inversion Rules). *Let Δ be a PUL applicable on a document D. The PUL Δ^{-1} obtained through the application of the inversion rules in Fig. 2 is an inverse of Δ according to Definition 2.* △

Proof Sketch. Consider a PUL Δ applicable on a document D and the inverse Δ^{-1} generated by the inversion rules. The proof of this proposition requires that: (*i*) Δ^{-1} is applicable on each document in $\mathcal{O}(\Delta, D)$, (*ii*) no pairs of incompatible operations are generated, (*iii*) no partially overridden operation either in Δ or Δ^{-1} exists, (*iv*) nodes removed by Δ are placed back in the correct positions by Δ^{-1}, (*v*) Δ^{-1} is deterministic. The proof of (*i*) is a straightforward consequence of the removal of overridden operations and of the operations definition. (*ii*) can be proved considering the algorithm definition and the applicability of Δ on D, while (*iii*) considering the operations definition. The proof of the other points comes directly from the definition of the algorithm, specifically (*iv*) is ensured by the class G rules, while (*v*) follows from the analysis of the generated operations.

3.3 Inversion Algorithm

Algorithm 1 presents an efficient procedure for computing the inverse of a PUL Δ. The following functions are exploited in the algorithm: $applyLocalOverrideRules(\Delta)$ to apply the reduction rules O1 and O2 in Fig. 2 on Δ; $applySInversionRules(\Delta, D)$ and $applyGInversionRules(\Delta, D)$ to apply the inversion rules of class S and G.

The inversion algorithm works as follows. An empty PUL Δ^{-1} is first initialized, then operations in Δ are ordered according to the pre-order traversal of their target nodes and grouped together. Note that, if an overriding operation *op* is present in a group, the groups of the overridden operations immediately follow that of *op*. Then, for each group of operations Δ_{v_i} on a node v_i, the algorithm performs the following steps. (*i*) It checks whether the operations Δ_{v_i} are overridden by operations specified on an ancestor of v_i, in this case they are discarded (this corresponds to the application of rules O3 and O4), otherwise, the $applyLocalOverrideRules$ function is applied on the operations of v_i. (*ii*) Whenever in Δ_{v_i} there is an operation *op* that may override operations in the subtree rooted at v_i, v_i is stored along with the relevant information about *op*. (*iii*) The rules of class S are applied on the remaining operations on v_i through $applySInversionRules$, updating Δ_{v_i} and adding the computed inverses to Δ^{-1}.

When all the partitions have been processed, the remaining operations in Δ are all and only those that belong to a removal group and each removal group is composed of operations that are contiguous in Δ. Function $applyGInversionRules$ can thus be efficiently applied on Δ, adding the inverses to Δ^{-1}.

Proposition 3 (Complexity). *Let Δ be a PUL applicable on a document D, removing r nodes. The complexity of the Algorithm 1 is $\mathcal{O}(n\ \log(n) + sn + r)$ where n is the size of Δ and s the cost of identifying the left sibling/parent of a node in D .* △

Algorithm 1. Inversion

Require: A PUL Δ applicable on a document D

1. $\Delta^{-1} = \emptyset$;
2. $(o, v_o) = (\bot, \bot)$;
3. let $(\Delta_{v_1}, ..., \Delta_{v_n})$ be the partition of Δ according to the preorder of their target node;
4. **for** $i = 1$ **to** n **do**
5. **if not** $((o{=}\texttt{pAttr} \wedge v_i \not/\!/_d^{\neg a} v_o) \vee (o{=}\texttt{rAttr} \wedge v_i \not/\!/_d v_o))$ **then**
6. $\Delta_{v_i} = applyLocalOverrideRules(\Delta_{v_i})$;
7. $delOp = \{o | o \in \Delta_{v_i}, \tau(v_i) = \texttt{e}, o(o) \in \{\texttt{repC}, \texttt{repN}, \texttt{del}\}\}$;
8. $(o, v_o) = \begin{cases} (\texttt{pAttr}, v_i) \text{ if } \texttt{repC} \in delOp \\ (\texttt{rAttr}, v_i) \text{ if } \texttt{repN} \in delOp \vee \texttt{del} \in delOp \end{cases}$
9. $\Delta^{-1} = \Delta^{-1} \cup applySInversionRules(\Delta_{v_i}, D)$
10. **end if**
11. **end for**
12. $\Delta^{-1} = \Delta^{-1} \cup applyGInversionRules(\Delta, D)$

Ensure: Δ^{-1} is an inverse of Δ on D according to Definition 2

Proof Sketch. The algorithm first requires to sort the PUL according to the pre-order traversal of their target nodes, which can be performed in $\mathcal{O}(n \log n)$, employing the labeling information and standard algorithms. Overridden operation removal then requires a single scan of the PUL, thus $\mathcal{O}(n)$. The application of inversion rules of class S requires to consider each remaining operation and might identify up to $n + r$ operations, $(\mathcal{O}(n + r))$. Finally, the application of the inversion rules of class G might require, for each operation, to determine the left sibling/parent of the operation target node in D. Assuming this cost s, the cost is $\mathcal{O}(sn)$, and, thus, the algorithm complexity is $\mathcal{O}(n \log(n) + sn + r)$.

4 Completed PULs

The approach discussed in the previous section, starting from a PUL Δ, identifies another PUL Δ^{-1} which reverts the effects of Δ. An alternative approach is to extend the operations presented in Table 2 with auxiliary information to allow their inverse (i.e., backward) application. These extended operations are reported in the last column of Table 2 and are referred to as *completed operations*. The PUL obtained from Δ by replacing its operations with the corresponding completed operations is named *completed PUL* and denoted by $\Delta^{\leftrightarrow}$. Completed PULs can be applied either forward (obtaining the same effect of the original PUL Δ) or backward (obtaining the same effect of one of its inverses). This approach does not require the explicit identification of an inverse PUL and avoids to access the document. Indeed, all required data is already contained in the completed PUL, which can be applied in streaming in both directions.

The forward application of a completed PUL simply consists in the streaming application of the corresponding PUL, i.e., ignoring additional information included in completed operations. The definition of obtainable documents is trivially extended to completed PULs: $\mathcal{O}(\Delta^{\leftrightarrow}, D) = \mathcal{O}(\Delta, D)$. The backward application of a completed PUL can be performed by applying a different set of transformations on event sequences.

Specifically: *remove* a node and its subtree from the sequence (for removing any inserted node); *restore* a subtree in its original position in the sequence (for restoring any removed subtree); *rename* a node (for inverting `ren` operations); *change value* to a node (for inverting `repV` operations). Consider a completed PUL $\Delta^{\leftrightarrow}$ applicable on a document D, and a document $D' \in \mathcal{O}(\Delta^{\leftrightarrow}, D)$. The algorithm for the backward application of $\Delta^{\leftrightarrow}$ on D' identifies a set of transformations for $\Delta^{\leftrightarrow}$ and applies them on the event sequence corresponding to D', obtaining the sequence of events corresponding to D. Note that no transformation must be applied on a restored subtree, since it is already restored as in the original document.

Example 7. Consider the completed PUL $\Delta^{\leftrightarrow}$ = {del(5, <name>EDBT04 W...^{6}</name>^{5}), repV(6, "VLDB04", "EDBT04 W..."), repN(7, <author>X^{25}</author>^{24}, <author>B.Catania^{8}</author>^{7}} applicable to the document in Fig. 1. To backward apply $\Delta^{\leftrightarrow}$, the inserted subtree (rooted at node 24) must be removed, while the removed subtrees (rooted at node 5 and 7) must be reintroduced. The inverse application of an overridden operation poses no problem, since the corresponding transformation has no effect. For instance, the inversion `repV` requires to change back the value of node 6, but node 6 will not be considered, as it is restored by another transformation.

Algorithm 2 realize the backward application of a completed PUL $\Delta^{\leftrightarrow}$, applicable on a document D, on a document $D' \in \mathcal{O}(\Delta^{\leftrightarrow}, D)$, identifying the sequence of events corresponding to D. Given $\Delta^{\leftrightarrow}$ and the sequence of events E' corresponding to D', the algorithm produces the sequence of events E corresponding to D. The *emit* $e_1, ..., e_n$ directive is employed to indicate that the events $e_1, ..., e_n$ are generated. For the sake of conciseness, we denote the node, name, and value components of an event e, as $e.v$, $e.n$, and $e.s$, respectively. Function $Events$ is used to associate a set of subtrees, considered according to the pre-order traversal of their roots in D, with the corresponding sequence of events. The algorithm works as follows. First, the sets I and R of the nodes inserted (that must be removed) and removed (that must be restored) by $\Delta^{\leftrightarrow}$ are identified. Then, the algorithm processes the document. The `sDoc` event is simply emitted back when encountered, while the other events are distinguished and the transformation applied. When the `eDoc` event is encountered, in case the root of D is present in R (denoted R_{root}), it is restored before emitting back `eDoc`. For each other event $e \in E'$, the original name and value of node $e.v$ are determined (undefined if $e.v$ does not have a name/value property), then one of the following steps is performed: (i) In case e is `sElem` or `text`: if the parent of $e.v$ is not being removed, any removed preceding siblings of $e.v$ in R (denoted $R_{\leftarrow}$) is emitted. Afterwards, if node $e.v$ should not be removed e is emitted with the original name and value of $e.v$, (followed by the attributes of $e.v$ in R, denoted R_{attr}, if e is a `sElem`). (ii) In case e is `eElem`: if $e.v$ should not be removed, the children of node $e.v$ in R (denoted $R_{\searrow}$) are emitted, followed by e with the original name and value of $e.v$. (iii) In case e is `attr`: e is emitted with the original name and value of $e.v$, unless $e.v$ has to be removed. To avoid restoring multiple times the same subtree, subtrees are removed from R when restored.

Proposition 4 (Correctness). *Let $\Delta^{\leftrightarrow}$ be a completed PUL applicable on a document D, and let D' be a document in $\mathcal{O}(\Delta^{\leftrightarrow}, D)$. The application of Algorithm 2 on $\Delta^{\leftrightarrow}$ and D' produces D.*

Algorithm 2. Completed PUL streaming backward application

Require: A completed PUL $\Delta^{\leftrightarrow}$ applicable on a document D, the sequence of events $[e_1, ..., e_n]$ for a document $D' \in \mathcal{O}(\Delta^{\leftrightarrow}, D)$

1. $I = \{V(p(op))|o(op) \in \{\texttt{ins}, \texttt{repN}, \texttt{repC}\} \wedge op \in \Delta^{\leftrightarrow}\}$ be the nodes inserted by $\Delta^{\leftrightarrow}$
2. $R = \{T(t(op))|o(op) \in \{\texttt{del}, \texttt{repN}\} \wedge op \in \Delta^{\leftrightarrow}\} \cup \{T(\gamma(t(op))|o(op) = \texttt{repC}, op \in \Delta^{\leftrightarrow}\}$ be the subtrees removed by $\Delta^{\leftrightarrow}$
3. **for** $i = 1$ **to** n **do**
4.   **if** $e_i = \texttt{sDoc}()$ **then**
5.   emit e_i
6.   **else if** $e_i = \texttt{eDoc}()$ **then**
7. $R_{root} = \begin{cases} T & \text{if } \exists\, T \in R \text{ s.t. } T \text{ is the document root} \\ \emptyset & \text{otherwise} \end{cases}$
8.   emit $Events(R_{root}), e_i$
9.   **else**
10. $\bar{n} = \begin{cases} n' & \text{if } \exists\, \texttt{ren}(e_i.v, n, n') \in \Delta^{\leftrightarrow} \\ e_i.n & \text{otherwise} \end{cases}$ $\bar{s} = \begin{cases} s' & \text{if } \exists\, \texttt{repV}(e_i.v, s, s') \in \Delta^{\leftrightarrow} \\ e_i.s & \text{otherwise} \end{cases}$
11.   **if** $e_i = \texttt{sElem}(v, n)$ **then**
12. $R_{\leftarrow} = \begin{cases} \{T_1, ..., T_n\} & \text{if } \exists\, [T_1, ..., T_n] \text{ removal group in } R \text{ s.t. } \mathcal{R}(T_n) \blacktriangleleft_s v \\ \emptyset & \text{otherwise} \end{cases}$
13.   **if** $\nexists v' \in I$ s.t. $v /_c v'$ **then** emit $Events(R_{\leftarrow})$ **end-if**
14. $R_{attr} = \{T_1, ..., T_n | \forall i, 1 \leq i \leq n, T_i \in R, \mathcal{R}(T_i) /_a v\}$
15.   **if** $v \notin I$ **then** emit $\texttt{sElem}(v, l, \bar{n}), Events(R_{attr})$ **end-if**
16. $R = R \setminus (R_{\leftarrow} \cup R_{attr})$
17.   **else if** $e_i = \texttt{eElem}(v, n)$ **then**
18. $R_{\searrow} = \begin{cases} \{T_1, ..., T_n\} & \text{if } \exists\, [T_1, ..., T_n] \text{ removal group in } R \text{ s.t. } \mathcal{R}(T_n) /_c v \\ \emptyset & \text{otherwise} \end{cases}$
19.   **if** $v \notin I$ **then** emit $Events(R_{\searrow}), \texttt{eElem}(v, \bar{n})$ **end-if**
20. $R = R \setminus R_{\searrow}$
21.   **else if** $e_i = \texttt{attr}(v, n, s)$ **then**
22.   **if** $v \notin I$ **then** emit $\texttt{attr}(v, \bar{n}, \bar{s})$ **end if**
23.   **else if** $e_i = \texttt{text}(v, s)$ **then**
24. $R_{\leftarrow} = \begin{cases} \{T_1, ..., T_n\} & \text{if } \exists\, [T_1, ..., T_n] \text{ removal group in } R \text{ s.t. } \mathcal{R}(T_n) \blacktriangleleft_s v \\ \emptyset & \text{otherwise} \end{cases}$
25.   **if** $\nexists v' \in I$ s.t. $v /_c v'$ **then** emit $Events(R_{\leftarrow})$ **end-if**
26. $R = R \setminus R_{\leftarrow}$
27.   **if** $v \notin I$ **then** emit $\texttt{text}(v, \bar{s})$ **end if**
28.   **end if**
29.   **end if**
30. **end for**

Ensure: The sequence of events emitted models the document D.

Proof Sketch. The inverse application of a completed PUL $\Delta^{\leftrightarrow}$ must (*i*) remove all inserted nodes; (*ii*) restore the original value and name of nodes, and (*iii*) insert back in the document any removed node in its original position. When the updated document is processed any node that has been inserted by $\Delta^{\leftrightarrow}$ (the nodes in I) is not emitted (removed). Otherwise the node is emitted back with its original value and name, which is retrieved considering the `ren` and `repV` operations in $\Delta^{\leftrightarrow}$. In this process, the node labels and the set of removed subtrees R are used to determine if any node must be

restored between two encountered nodes, ensuring that the original document nodes are restored in their original position. Therefore, since the three properties are met by Algorithm 2, we are guaranteed that the original document is produced.

Proposition 5 (Complexity). *Let $\Delta^{\leftrightarrow}$ be a completed PUL of size n applicable on a document D, let D' be a document in $\mathcal{O}(\Delta^{\leftrightarrow}, D)$ composed of d nodes, and let i and r be the number of nodes inserted and removed by $\Delta^{\leftrightarrow}$, respectively. The complexity of Algorithm 2 is $\mathcal{O}(n \log n + d + i + r)$.*

Proof Sketch. The identification of the nodes to be removed and the original name and value of updated nodes can rely on a hash-table, which requires $\mathcal{O}(n)$ for its initialization and $\mathcal{O}(1)$ for retrievals. For what concerns the restoration of nodes, a more efficient representation of R is a list ordered according to the pre-order traversal of the subtrees roots, with cost $\mathcal{O}(n \log n)$. Moreover, since events are generated according to a pre-order visit of the tree, identifying whether there is a subtree to restore, requires to check only the first element in the list, with cost $\mathcal{O}(1)$, assuming that subtrees removed multiple times are pruned during the inverse application. Assessing whether the parent of a node has been removed requires constant time, provided that this information is stored in a state variable. Thus, since all nodes in the updated document need to be processed and up to r nodes need to be restored, the complexity of the algorithm is $\mathcal{O}(n \log n + d + i + r)$.

5 Evaluation

In this section we discuss some experiments we have conducted by means of the extended Qizx library that is able to generate and process both PULs and completed PULs represented as XML documents containing the serialization of each operation along with the identifiers and labels of the target nodes. Our test machine uses an Intel I5 760 processor, 16GB of RAM, and runs the Sun JDK v.1.6.20.

To assess the computational costs of the inversion operator and contrast the peculiar advantages of PULs and completed PULs, we exploit documents of various sizes, ranging from 16MB to 256MB, produced by means of the XMark data generator. Node identifiers and labeling have been stored within the corresponding documents through an encoding of their XDM structure. On such documents, synthetic XQU expressions and their corresponding PULs/completed PULs have been generated with a varying number of operations that are equally distributed among the operation types.

Starting from an XQU expression the generation of a PUL Δ/completed PUL $\Delta^{\leftrightarrow}$ requires to load the whole document in memory before the actual evaluation of the XQuery Update expression. When the expression is evaluated, generating a completed PUL has the additional cost of retrieving and storing within the PUL itself all the modified values and removed nodes. Analogously, the generation of PUL inversion requires to load the whole document in memory. The application of the so generated PULs, by contrast, requires to load the whole PUL in memory and then update the corresponding document through a single scan identifying a new document. While Δ and Δ^{-1} serializations contain only the strictly required information for their application, the serialization of $\Delta^{\leftrightarrow}$ contains extra information that is discarded depending on the direction (forward or backward) of application.

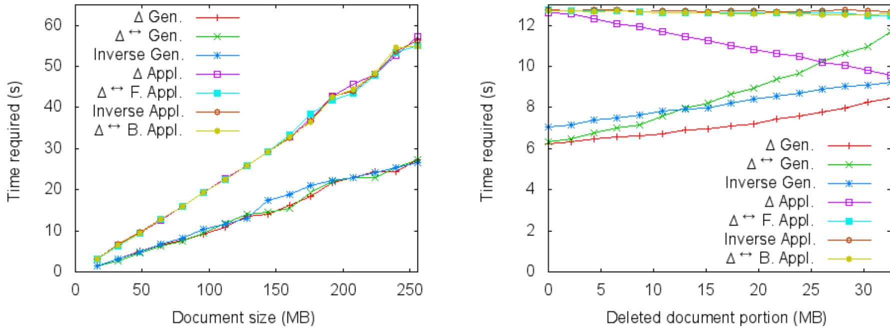

Fig. 3. Experiments

Given a synthetic XQU expression, we first investigated the correlation between the size of the document and the time required for: (i) generating the corresponding PUL Δ/completed PUL $\Delta^{\leftrightarrow}$; (ii) computing the inverse Δ^{-1} of Δ; (iii) applying Δ, $\Delta^{\leftrightarrow}$, Δ^{-1} and backward applying $\Delta^{\leftrightarrow}$. In all cases, the portion of the document being inserted or removed is approximately 1MB. Results for this experiment are reported on the left side of Fig. 3. Moreover, we consider a 64MB document and we vary the amount of information removed or replaced (ranging from 0% to 50%). Results of the second experiment are reported in the right side of Fig. 3. The experiments indicate that the cost of generating or applying a PUL Δ^* (either Δ, $\Delta^{\leftrightarrow}$, or Δ^{-1}) is dominated by the number of nodes analysed. Specifically, when a PUL Δ or $\Delta^{\leftrightarrow}$ is applied, the number of nodes in the original document, those in the updated document and those in the PUL itself contribute to the running time of PUL application. Analogously, for the application of Δ^{-1}, the number of nodes in the original document and those in Δ and Δ^{-1} affect the running time. Thus, when the portion of the document being inserted or deleted is not relevant, PULs generation and application time are not influenced by Δ^*, while applications require almost twice the time of generations. By contrast, when the size of the deleted portions of the document increases, completed PULs become less efficient, as the forward application time is unchanged and its generation time increases. Similarly, when the number of nodes inserted by a PUL is high, PUL backward application time remains unchanged, and its generation time increases, even if to a lesser extent.

6 Related Work

The paper relies on the approach to update execution proposed in [3] which proposes the decoupling of PUL production from PUL evaluation as an execution model for XML updates expressed through XQU expressions in distributed contexts. The need of more flexible mechanisms for update handling entails the development of suitable mechanisms for reasoning on updates before actually apply them, and specifically for composing as well as reverting them. In [3], PUL operators for composing updates (integration and aggregation) as well as for devising a compact representation (reduction) are investigated, but inversion is not addressed.

The notion of completed PULs is inspired by completed deltas proposed by [11] in the context of XML versioning. The goal of the two notions is the same, in that they both aim at having a compact representation of changes that can be applied either forward (to actually apply them) or backward (to revert their effects). However, completed deltas are obtained by comparing two document versions (through diff algorithms) and do not represent the effects of XQU expressions. As a consequence, both the set of primitive operations and the associated semantics are different. A peculiar aspect in inverting updates expressed as PUL is indeed related to the XQU snapshot semantics by which a PUL is a unordered list of operations, that have to be applied on documents according to some precedence among operators prescribed by [15] and formalized in a five stage semantics in [1]. The inversion mechanism proposed in the paper is designed according to that semantics. By contrast, completed deltas refer to sequences of non-conflicting operations.

Though our approach is not specifically targeted to a transactional context, the ability of reverting the effects of XML updates could also be useful in that setting. The notion of compensating transaction as a transaction that semantically undoes the partial effects of a transaction without performing cascading aborts of dependent transactions, restoring the system to a consistent state, has been proposed in the context of long-lived transaction [6,7,10] and is particularly relevant in the context of workflows and web services [13]. These types of compensation range from traditional undo, at one extreme, to application-dependent, special-purpose compensating transactions, at the other extreme. XML transactions have been investigated in [5,8,9] but the focus was on isolation levels and lock mechanisms. An approach to atomicity for XML transactions relying on statement undos is proposed in [2]. They consider the update operations in a PUL as separate transactions and discuss how individual operations can be undone. However, interactions among different operations in a PUL, e.g., overriding, neither on the same node nor on nodes bound by hierarchical relationships in the tree, are considered.

7 Concluding Remarks

In this paper we considered updates on XML documents expressed through XQU expressions and the corresponding dynamic model of updates based on PULs. We investigated the issue of reverting the effects of an update expression, referring to the case in which PUL production is decoupled from their application. Two alternative approaches have been considered: the first one is the inversion of a PUL through an inversion operator. The second one is the extension of the PUL model (completed PULs) so that the required information for both a forward and a backward (inverse) application are included. We presented, discussed, and implemented in the Qizx library the algorithms for both inverting a PUL and for the streaming backward application of a completed PUL. Finally, we contrasted the two proposed approaches through an experimental evaluation. As future work we plan to investigate the correlation between the inversion operators presented in this paper with the operations proposed in [3] in order to identify an algebra on PULs. Moreover, we wish to tailor the developed operator in specific contexts like versioning, transaction, and cloud. Finally, in the current setting, labeling information is stored within the document. This has the effect of increasing

the document size of three times. This issue could be solved by considering a shredded representation of the document and, as future work, we wish to explore this possibility.

References

1. Benedikt, M., Cheney, J.: Semantics, Types and Effects for XML Updates. In: Gardner, P., Geerts, F. (eds.) DBPL 2009. LNCS, vol. 5708, pp. 1–17. Springer, Heidelberg (2009)
2. Biswas, D., Jiwane, A., Genest, B.: Atomicity for XML Databases. In: Bellahsène, Z., Hunt, E., Rys, M., Unland, R. (eds.) XSym 2009. LNCS, vol. 5679, pp. 180–187. Springer, Heidelberg (2009)
3. Cavalieri, F., Guerrini, G., Mesiti, M.: Dynamic Reasoning on XML Updates. In: EDBT, pp. 165–176. ACM Digital Library (2011)
4. Chien, S.-Y., Tsotras, V.J., Zaniolo, C.: Efficient Schemes for Managing Multiversion XML Documents. VLDB J. 11(4), 332–353 (2002)
5. Dekeyser, S., Hidders, J., Paredaens, J.: A Transaction Model for XML Databases. World Wide Web 7(1), 29–57 (2004)
6. Elmagarmid, A.K. (ed.): Database Transactional Models for Advanced Applications. Morgan Kaufmann, San Francisco (1992)
7. Garcia-Molina, H., Salem, K.: Sagas. In: SIGMOD Conference, pp. 249–259. ACM Press, New York (1987)
8. Grabs, T., Böhm, K., Schek, H.-J.: XMLTM: Efficient Transaction Management for XML Documents. In: CIKM, pp. 142–152 (2002)
9. Helmer, S., Kanne, C.-C., Moerkotte, G.: Evaluating Lock-based Protocols for Cooperation on XML Documents. SIGMOD Record 33(1), 58–63 (2004)
10. Korth, H.F., Levy, E., Silberschatz, A.: A Formal Approach to Recovery by Compensating Transactions. In: VLDB, pp. 95–106 (1990)
11. Marian, A., Abiteboul, S., Cobena, G., Mignet, L.: Change-Centric Management of Versions in an XML Warehouse. In: VLDB, pp. 581–590 (2001)
12. O'Connor, M.F., Roantree, M.: Desirable Properties for XML Update Mechanisms. In: Updates in XML EDBT/ICDT Workshop (2010)
13. Peltz, C.: Web Services Orchestration and Choreography. Computer 3(10), 46–52 (2003)
14. PIXwere Ltd. QIZX. An Open-source XQuery Processor (2010)
15. W3C. XQuery Update Facility 1.0 (June 2009)