Dynamic Reasoning on XML Updates

Federico Cavalieri
University of Genova, Italy
cavalieri@disi.unige.it

Giovanna Guerrini
University of Genova, Italy
guerrini@disi.unige.it

Marco Mesiti
University of Milano, Italy
mesiti@dico.unimi.it

ABSTRACT

In many emerging XML application contexts and distributed execution environments (like disconnected and cloud computing, collaborative editing and document versioning) the server that determines the updates to be performed on a document, by evaluating an XQuery Update expression, is not always the same that actually makes such updates -represented as Pending Update Lists (PULs)- effective. The process of generating the PUL is thus decoupled from that of making its effect persistent on the document. The PUL executor needs to manage several PULs, that, depending on the application context, are to be executed as sequential or parallel update requests, possibly relying on application-specific policies. This requires some capabilities of dynamic reasoning on updates. In the paper, we state the most relevant properties to reason on, develop the corresponding algorithms and present a PUL handling system, providing an experimental evaluation of this system.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages; H.2.4 [**Database Management**]: Systems — *Query Processing*

General Terms

Algorithms, Languages, Measurement, Performance

1. INTRODUCTION

Most of the work on XML updates and current implementations of the XQuery Update Facility [22] rely on the assumption that updates are executed right after and on the same server where the expression requesting them is evaluated. In many distributed environments, however, several application contexts require the process of expressing and requesting updates to be decoupled from that of making them effective on documents. The ability to handle update requests, expressed as Pending Update Lists (PULs) resulting from the evaluation of updating expressions, and to reason about them is thus needed. PULs can be exchanged among nodes and algorithms to handle them as parallel (i.e., to be integrated as a single transformation from the current document to a new one) or sequential (i.e., to be applied one after the other) update requests are required.

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

In the context of collaborative editing [1, 17], for instance, a node, handling the authoritative version of a document, may share it with the collaborating nodes, and then recollect back from them the PULs expressing updates they would like to perform on such a document. After integrating and reconciling such update requests, that are to be interpreted as parallel, a new authoritative version is generated. This relies on an appropriate definition of conflicts and of check-in/check-out approaches *à la SVN* to integrate updates from different nodes.

In disconnected execution, a node may work for a while on a document, thus producing different PULs resulting in different sequentially updated documents. When it later reconnects to the server holding the document, it sends the sequence of PULs to be applied to align the local and remote versions of the document. To avoid going through all the intermediate documents, the ability to aggregate the sequential PULs in a single one is needed. In case of big documents, it is more convenient to exchange deltas (i.e., the updates -expressed as PULs- that the documents have undergone) rather than the documents themselves. A similar operation is needed in document versioning and temporal XML [8, 10, 11], where we may want to get rid of some intermediate document versions (expressed as deltas over an original version) and only keep the most relevant ones.

In the context of the cloud [20], finally, updates travel on the network, and arrive at the node holding the document they refer to, where they are collected, and only subsequently actually applied. This requires the ability both to integrate and reconcile updates specified on the same document state and to aggregate updates intended to be applied sequentially. In most of the devised contexts, the node in charge of executing the updates cannot rely on interactions with the nodes requesting them in case conflicts among requests originated by different nodes arise.

In all the above mentioned contexts, the ability of manipulating PULs and to reason on their effects is required. A PUL is an unordered list of primitive tree update operations, each one targeted at a single node. Even if the list is unordered, in the language semantics a precedence among operations is specified (e.g., deletion follows all the other operations). The insertion position of some `insert` operations is left open to implementation leading to a degree of non-determinism in update execution. Moreover, some operations (such as two renames) on the same node would result in an unclear semantics, thus they are defined as incompatible operations. A PUL containing incompatible operations is not applicable, in that its execution produces a dynamic error. The current W3C specification defines a *merge* operation that merges two different PULs in a single one, provided that the set of operations resulting from their union contains no incompatible operations.

Decoupling PUL production (through the evaluation of XQuery

Update expressions on a document) from PUL execution introduces additional costs in serializing and exchanging PULs on the network if compared to a scenario where PULs are executed locally and right after being produced. However, the PUL executor may take advantage of the knowledge of the exact nodes targets of the updates to execute PULs in streaming, without the need to access and load in main memory the whole document.

Reasoning on XML updates has mostly been intended as static analysis of update expressions [6]. Dynamic update analysis is certainly easier, since more concrete information about data is available in PULs than in update expressions, but it also raises new interesting issues. A first one is node identification: since PULs have to be exchanged we cannot rely on the local main-memory representation of a document anymore. Another important issue is document independence: our operations on PULs do not require accessing the document, rather they benefit from structural information on the document that is incorporated in the PULs themselves through a labeling scheme. This labeling allows us to assess the structural relationships among target nodes that are needed by the reasoning. Since it is intended to be applied in a document update framework, the labeling scheme used to represent these relationships must accommodate updates. The only approach to dynamic reasoning on XML updates we are aware of is [11], and it does not cope with these issues, since the proposed PUL composition operator composes the effects of a sequence of locally executed PULs.

The specific PUL operations we consider are: PUL reduction, integration and aggregation. The *reduction* operation transforms a PUL with the aim of collapsing similar operations and removing useless operations (i.e., operations whose effects are overridden by others). In addition, deterministic reduction produces a PUL whose semantics is deterministic, and a canonical form is proposed so that each PUL has a single canonical reduction. The *integration* of two PULs produces a single PUL containing their operations that do not conflict and a set of *conflicting operations* representing clashes among operations in the PULs, i.e., points in which their intentions in updating the document seem different. Integration is useful in parallel execution of PULs, and corresponds to merge in case no conflicts arise (it is however stronger than merge since incompatibilities are only one among the five types of conflicts detected by integration). For solving conflicts, we propose the use of application-specific policies. Each producer can specify its own desiderata on the PULs it sends for execution (e.g., order matters, deletion means "I do not need it any more" rather than "It must be deleted") and the executor may rely on these policies, on its own policies (e.g., producers reputation) to reconcile conflicting updates. The *aggregation* of two PULs, by contrast, refers to their sequential execution, and produces a single PUL cumulating their effects.

The paper thus brings several contributions: (*i*) definition of a mechanism for exchanging PULs in a distributed environment, and modification of Qizx [18] to produce PULs and accept PULs (represented as XML documents) as input; (*ii*) characterization of PUL equivalence, reduction, integration, and aggregation in terms of PUL semantics as prescribed by [22] and formalized by [5]; (*iii*) development of efficient algorithms for reduction, integration, and aggregation possibly relying on application-specific policies for solving conflicts; (*iv*) implementation of such algorithms in the PUL handling system and their experimental evaluation.

The remainder of the paper is organized as follows. Section 2 introduces the notions our framework relies on, Section 3 presents the PUL operations and related algorithms, while Section 4 describes how they are implemented in a PUL handling system and evaluates its performance. Finally, Section 5 discusses related work and Section 6 concludes the paper.

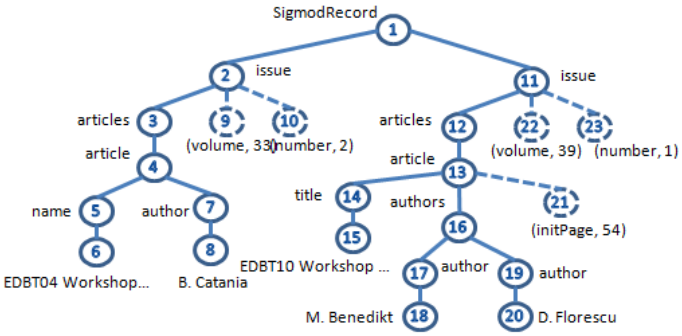


Figure 1: Tree representation of an XML document

2. PRELIMINARIES

In this section we introduce the adopted tree representation of XML documents, then define PULs of operations according to [22], present their semantics and introduce the notions of PUL equivalence and substitutability.

2.1 XML Document Representation

An XML document is represented relying on the notion of labeled tree. A document D is described by $(V, \gamma, \lambda, \nu)$ where: V is a set of nodes representing elements, attributes or the value of elements (text nodes)[1]; 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 $\mathcal{R}$[2] denote the nodes and the root of a document D, respectively, and τ assigns to each node v in V a value in the set $\{\mathtt{e}, \mathtt{a}, \mathtt{t}\}$ denoting its type. Coherently with the XDM model, the attribute value is seen as a property of the attribute node, whereas the textual contents of an element are modeled by separate nodes. A unique identifier, preserved upon modification, is associated with each node in V, so that identifiers of nodes removed from the document are not reused. In what follows, wherever no confusion arises, we do not distinguish nodes from their identifiers. Figure 1 reports a fragment of the `SigmodRecord` document that needs to be updated. Dotted lines are used to represent edges leading to attribute nodes and to point out that their relative order is not relevant.

In handling PULs, we avoid to directly access and navigate the document. Rather, we simply need to consider some structural relationships among nodes targets of updates. The required structural information includes the ability to check: whether the parent-child (P-C), element-attribute (E-A), left sibling, first/last child, ancestor-descendant (A-D) and preceder-follower (P-F) relationships hold among two nodes. This information can be obtained through a labeling scheme associated with document nodes. In Section 4 we will detail different labeling schemes that we have adopted in our prototype. Table 1 reports the predicates that can be assessed through the labeling scheme.

2.2 Update Operations and PULs

We consider the update primitives defined in [22] and reported in Table 2, where $v \in V$ is a node, $P = [T_1, \ldots, T_n]$, $n \geq 0$, is a list (possibly empty in case of the `repN` or `repC` operation) of trees, l is a name in $\mathcal{N}$, s is a value in $\mathcal{V}$.

[1]For simplicity, we consider only these types among those in [22].
[2]Function $\mathcal{R}$ generalizes to a list of trees returning their roots.

Predicate	Description
$v_1 \lhd v_2$	v_1 precedes v_2 in document order
$v_1 \lhd_s v_2$	v_1 is left 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
$v_1 /_c^{\leftarrow} v_2$	v_1 is the first child of v_2
$v_1 /_c^{\rightarrow} v_2$	v_1 is the last child 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 1: Structural relationships

The first parameter of each primitive specifies the affected node, that is, the operation *target*. Given an operation op, $t(op) \in V$ denotes its target, $o(op)$ denotes its name, and $p(op)$ denotes its second parameter (undefined if $o(op) = \texttt{del}$). The update operations can be categorized in three main classes: insertions (all the variants of `ins`), deletions (operation `del`), replacements (all the variants of `rep` and `ren`). $c(op) \in \{\texttt{i}, \texttt{d}, \texttt{r}\}$ denotes the class of operation op. For each operation, some applicability conditions are given, stating the conditions that the parameters must satisfy for the operation to be applicable.

DEFINITION 1 (APPLICABLE OPERATION). *An update operation op is applicable on a document D if $t(op) \in V(D)$ and op matches the applicability conditions in D.* □

The semantics of each operation can be expressed through the judgement $D \models op \rightsquigarrow D'$. The semantics we refer to has been obtained by adapting the one given in [5] to our tree representation of XML documents and is not presented here for space constraints (it can be found in [7]). The semantics of operation $\texttt{ins}^{\downarrow}$ is nondeterministic because the actual position of the inserted nodes in a document is not univocally specified. Therefore, because of this operation, the application of an operation to a document produces a document in a set of possible outcomes, we refer to as *set of obtainable documents*. Specifically, this set is always a singleton, except for the $\texttt{ins}^{\downarrow}$ operation, for which it is a set of documents that differ only for the position of the inserted children among sibling nodes. The semantics of an operation on a document produces one of the documents in the obtainable set.

DEFINITION 2 (OPERATION SEMANTICS). *Let op be an update operation applicable on a document D. The set $\mathcal{O}(op, D) = \{D' | D \models op \rightsquigarrow D'\}$ denotes the documents obtainable by the application of op on D. The application of op on D produces any of the documents in the set $\mathcal{O}(op, D)$.* □

EXAMPLE 1. *Let D be the document in Figure 1. $op_1 = \texttt{del}(14)$ involves no non-determinism and thus $\mathcal{O}(op_1, D)$ is a singleton. $op_2 = \{\texttt{ins}^{\downarrow}(16, \texttt{<author>G.Guerrini</author>})$, 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 the three corresponding documents.*

A *pending update list* (*PUL*) [22] 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.

DEFINITION 3 (OPERATION COMPATIBILITY). *Let op_1 and op_2 be two operations. They are compatible unless $t(op_1) = t(op_2)$, $o(op_1) = o(op_2)$ and $c(op_1) = \texttt{r}$.* □

EXAMPLE 2. *Consider the operations: $op_1 = \texttt{ren}(1, \texttt{dblp})$, $op_2 = \texttt{ren}(1, \texttt{myDblp})$, $op_3 = \texttt{repC}(1, \texttt{'nopapers'})$. Operations op_1 and op_3 are compatible, and so are op_2 and op_3, whereas op_1 and op_2 are incompatible.*

Operation	Description	Conditions
$\texttt{ins}^{\leftarrow}(v, P)$ $\texttt{ins}^{\rightarrow}(v, P)$	Insert the trees in P before/after node v	$\tau(v) \neq \texttt{a}$, $\forall r \in \mathcal{R}(P)$ $\tau(r) \neq \texttt{a}$
$\texttt{ins}^{\swarrow}(v, P)$ $\texttt{ins}^{\searrow}(v, P)$	Insert the trees in P as first/last children of node v	$\tau(v) = \texttt{e}$, $\forall r \in \mathcal{R}(P)$ $\tau(r) \neq \texttt{a}$
$\texttt{ins}^{\downarrow}(v, P)$	Inserts the trees in P as children of node v, in an implementation defined position	$\tau(v) = \texttt{e}$, $\forall r \in \mathcal{R}(P)$ $\tau(r) \neq \texttt{a}$
$\texttt{insA}(v, P)$	Inserts the trees in P as attributes of node v	$\tau(v) = \texttt{e}$, $\forall r \in \mathcal{R}(P)$ $\tau(r) = \texttt{a}$
$\texttt{del}(v)$	Deletes node v	
$\texttt{repN}(v, P)$	Replaces node v with the trees in P (possibly none)	$\forall r \in \mathcal{R}(P)$ $(\tau(r) = \tau(v) = \texttt{a}) \vee (\tau(v) \neq \texttt{a} \wedge \tau(r) \neq \texttt{a})$
$\texttt{repV}(v, s)$	Replaces the value of node v with $s \in \mathcal{V}$	$\tau(v) \in \{\texttt{t}, \texttt{a}\}$
$\texttt{repC}(v, t)$	Replaces the children of node v with text node t or with nothing	$\tau(v) = \texttt{e}, t = [] \vee \tau(t) = \texttt{t}$
$\texttt{ren}(v, l)$	Renames the label of node v with $l \in \mathcal{N}$	$\tau(v) \in \{\texttt{e}, \texttt{a}\}$

Table 2: Update operations

For a PUL to be applicable on a document (cf. function `applyUpdates` in [22]) it must contain no incompatible operations and all its operations must be applicable on the document.

DEFINITION 4 (APPLICABLE PUL). *Let D be a document and $\Delta = \{op_1, \ldots, op_n\}$, with $n \geq 0$, be a PUL. Δ is applicable on D if $\forall op \in \Delta$ op is applicable on D and $\forall op, op' \in \Delta$ op and op′ are compatible.* □

The merge operation (cf. function `mergeUpdates`) is the only operation considered in [22] for handling PULs .

DEFINITION 5 (PUL MERGE). *Let Δ_1 and Δ_2 be PULs applicable on a document D. Their merge, denoted by $\Delta_1 \circ \Delta_2$, is defined as $\Delta_1 \cup \Delta_2$ provided that it is applicable on D.* □

The semantics of a PUL Δ on a document D can be easily specified by extending the judgement $\models$ and the obtainable set $\mathcal{O}$ from a single operation to a set of operations. The specification of $D \models \Delta \rightsquigarrow D'$ is obtained by applying the operations in Δ in five stages (the full specification can be found in [7]). At each stage a subset of the operations are applied in order to guarantee the precedence on the type of operations specified in [22]. The operations in each stage are the following: (1) $\texttt{ins}^{\downarrow}$, `insA`, `repV`, `ren`; (2) $\texttt{ins}^{\leftarrow}$, $\texttt{ins}^{\rightarrow}$, $\texttt{ins}^{\swarrow}$, $\texttt{ins}^{\searrow}$; (3) `repN`; (4) `repC`; (5) `del`.

The order of application of operations within each stage is not prescribed by [22]. Thus, when multiple insertion operations of the same type with the same target appear in the same PUL, the relative order of their inserted groups of children is not fixed as well. Therefore, the cardinality of $\mathcal{O}(\Delta, D)$ is greater than one when $\texttt{ins}^{\downarrow}$ occurs in Δ or Δ contains more than one insertion operation of the same type and on the same target.

EXAMPLE 3. *Let $\Delta = \{\texttt{ins}^{\downarrow}(16, \texttt{<author>G.Guerrini</author>}),$ $\texttt{ins}^{\searrow}(4, \texttt{<initP>132</initP>}), \texttt{ins}^{\searrow}(4, \texttt{<lastP>134</lastP>})\}$ be a PUL on the document D in Figure 1. $|\mathcal{O}(\Delta, D)| = 6$.*

The notion of obtainable documents set can be extended to a set of documents S in the intuitive way as $\mathcal{O}(\Delta, S) = \bigcup_{D \in S} \mathcal{O}(\Delta, D)$, and to a sequence of consecutive PULs as follows: $\mathcal{O}(\Delta_1; \Delta_2, D) = \mathcal{O}(\Delta_2, \mathcal{O}(\Delta_1, D))$.

2.3 PULs Equivalence and Substitutability

To reason about PULs, it is important to determine whether two PULs have the same effects, that is, they are equivalent, or the set of obtainable documents by the application of a PUL Δ_1 is contained in that of another PUL Δ_2 on the same document, that is, the first PUL is substitutable to the second one.

O1) $\dfrac{op_1 = op(v, \_)\ op_2 = op'(v, \_)}{op_1, op_2 \triangledown_1 op_2}$ $op \in \{\texttt{ren}, \texttt{repV}, \texttt{repC}, \texttt{del}\} \cup \{\texttt{ins}^{\swarrow}, \texttt{ins}^{\searrow}, \texttt{ins}^{\downarrow}, \texttt{insA}\}$ $op' \in \{\texttt{repN}, \texttt{del}\}$

O2) $\dfrac{op_1 = op(v, \_)\ op_2 = \texttt{repC}(v, \_)}{op_1, op_2 \triangledown_1 op_2}$ $op \in \{\texttt{ins}^{\swarrow}, \texttt{ins}^{\downarrow}, \texttt{ins}^{\searrow}\}$

O3) $\dfrac{op_1 = op(v, \_)\ \ op_2 = op'(v', \_)}{op_1, op_2 \triangledown_1 op_2}$ $op' \in \{\texttt{repN}, \texttt{del}\},\ v \mathbin{/\!\!/_d} v'$

O4) $\dfrac{op_1 = op(v, \_)\ \ op_2 = \texttt{repC}(v', L)}{op_1, op_2 \triangledown_1 op_2}$ $v \mathbin{/\!\!/_d^a} v'$

I5) $\dfrac{op_1 = op(v, L_1)\ \ op_2 = op(v, L_2)}{op_1, op_2 \triangledown_1 op(v, [L_1, L_2])}$ $c(op) = \texttt{i}$

I6) $\dfrac{op_1 = \texttt{ins}^{\downarrow}(v, L_1)\ \ op_2 = \texttt{ins}^{\swarrow}(v, L_2)}{op_1, op_2 \triangledown_2 \texttt{ins}^{\swarrow}(v, [L_2, L_1])}$

I7) $\dfrac{op_1 = \texttt{ins}^{\downarrow}(v, L_1)\ \ op_2 = \texttt{ins}^{\searrow}(v, L_2)}{op_1, op_2 \triangledown_3 \texttt{ins}^{\searrow}(v, [L_1, L_2])}$

IR8) $\dfrac{op_1 = \texttt{repN}(v, L_1)\ \ op_2 = \texttt{ins}^{\leftarrow}(v, L_2)}{op_1, op_2 \triangledown_4 \texttt{repN}(v, [L_2, L_1])}$

IR9) $\dfrac{op_1 = \texttt{repN}(v, L_1)\ \ op_2 = \texttt{ins}^{\rightarrow}(v, L_2)}{op_1, op_2 \triangledown_4 \texttt{repN}(v, [L_1, L_2])}$

I10) $\dfrac{op_1 = \texttt{ins}^{\downarrow}(v, L_1)\ \ op_2 = \texttt{ins}^{\leftarrow}(v', L_2)}{op_1, op_2 \triangledown_5 \texttt{ins}^{\leftarrow}(v, [L_1, L_2])}$ $v' \mathbin{/_c} v$

I11) $\dfrac{op_1 = \texttt{ins}^{\downarrow}(v, L_1)\ \ op_2 = \texttt{ins}^{\rightarrow}(v', L_2)}{op_1, op_2 \triangledown_6 \texttt{ins}^{\rightarrow}(v, [L_2, L_1])}$ $v' \mathbin{/_c} v$

IR12) $\dfrac{op_1 = \texttt{repN}(v, L_1)\ \ op_2 = \texttt{ins}^{\downarrow}(v', L_2)}{op_1, op_2 \triangledown_7 \texttt{repN}(v, [L_1, L_2])}$ $v \mathbin{/_c} v'$

IR13) $\dfrac{op_1 = \texttt{repN}(v, L_1)\ \ op_2 = \texttt{insA}(v', L_2)}{op_1, op_2 \triangledown_8 \texttt{repN}(v, [L_1, L_2])}$ $v \mathbin{/_a} v'$

I14) $\dfrac{op_1 = \texttt{ins}^{\leftarrow}(v, L_1)\ \ op_2 = \texttt{ins}^{\swarrow}(v', L_2)}{op_1, op_2 \triangledown_8 \texttt{ins}^{\leftarrow}(v, [L_2, L_1])}$ $v \mathbin{/_c^{\leftarrow}} v'$

I15) $\dfrac{op_1 = \texttt{ins}^{\rightarrow}(v, L_1)\ \ op_2 = \texttt{ins}^{\searrow}(v', L_2)}{op_1, op_2 \triangledown_8 \texttt{ins}^{\rightarrow}(v, [L_1, L_2]),}$ $v \mathbin{/_c^{\rightarrow}} v'$

IR16) $\dfrac{op_1 = \texttt{repN}(v, L_1)\ \ op_2 = \texttt{ins}^{\swarrow}(v', L_2)}{op_1, op_2 \triangledown_8 \texttt{repN}(v, [L_2, L_1])}$ $v \mathbin{/_c^{\leftarrow}} v'$

IR17) $\dfrac{op_1 = \texttt{repN}(v, L_1)\ \ op_2 = \texttt{ins}^{\searrow}(v', L_2)}{op_1, op_2 \triangledown_8 \texttt{repN}(v, [L_1, L_2])}$ $v \mathbin{/_c^{\rightarrow}} v'$

I18) $\dfrac{op_1 = \texttt{ins}^{\leftarrow}(v, L_1)\ \ op_2 = \texttt{ins}^{\rightarrow}(v', L_2)}{op_1, op_2 \triangledown_9 \texttt{ins}^{\leftarrow}(v, [L_2, L_1])}$ $v' \lhd_s v$

IR19) $\dfrac{op_1 = \texttt{repN}(v, L_1)\ \ op_2 = \texttt{ins}^{\rightarrow}(v', L_2)}{op_1, op_2 \triangledown_9 \texttt{repN}(v, [L_1, L_2])}$ $v' \lhd_s v$

IR20) $\dfrac{op_1 = \texttt{repN}(v, L_1)\ \ op_2 = \texttt{ins}^{\leftarrow}(v', L_2)}{op_1, op_2 \triangledown_9 \texttt{repN}(v, [L_2, L_1])}$ $v \lhd_s v'$

Figure 2: Reduction rules

DEFINITION 6 (EQUIVALENCE AND SUBSTITUTABILITY). *Let Δ_1 and Δ_2 two PULs applicable on a document D. Δ_1 is equivalent to Δ_2 on D, denoted $\Delta_1 \simeq_D \Delta_2$, if and only if $\mathcal{O}(\Delta_1, D) = \mathcal{O}(\Delta_2, D)$. Δ_1 is substitutable to Δ_2 on D, denoted $\Delta_1 \preceq_D \Delta_2$, if and only if $\mathcal{O}(\Delta_1, D) \subseteq \mathcal{O}(\Delta_2, D)$.* □

EXAMPLE 4. *Consider PULs* $\Delta_1 = \{\texttt{ins}^{\rightarrow}(19, \texttt{<author>M.Mesiti</author>}), \texttt{repV}(15, \texttt{'Report on ...'})\}$ *and* $\Delta_2 = \{\texttt{ins}^{\searrow}(16, \texttt{<author>M.Mesiti</author>}), \texttt{repC}(14, \texttt{'Report on ...'})\}$. *$\Delta_1$ is equivalent to Δ_2. Consider now PULs* $\Delta_1 = \{\texttt{ins}^{\searrow}(4, \texttt{<initP>132</initP>}), \texttt{ins}^{\searrow}(4, \texttt{<lastP>134</lastP>})\}$, *and* $\Delta_2 = \{\texttt{ins}^{\searrow}(4, \texttt{<initP>132</initP>}, \texttt{<lastP>134</lastP>})\}$. *$\Delta_2$ is substitutable to Δ_1.*

3. HANDLING PULS

In this section we present three operators for reasoning on PULs. For each operator we provide the definition, state some properties and present the algorithms for its evaluation, discussing their complexity. The algorithms rely on the use of rules on pairs of operations that are organized in stages. When rules in a stage cannot be applied any more, rules of the next stage are considered.

3.1 PUL Reduction

According to its definition [22], a PUL may contain interacting operations: they may have the same target and one may even destroy (i.e., override) the effects of others. Reducing the operations in a PUL, by collapsing similar operations and removing operations whose effects are overridden aims at obtaining a more compact PUL with the same effect of the original one. For this purpose, a set of *reduction* rules reported in Figure 2 has been devised. Each rule considers a pair of operations that can be reduced obtaining a single operation. Rules are organized in the following three categories, and are evaluated through the $\triangledown$ operator in 9 stages.

- O Rules for eliminating overridden operations when a `repC`, `repN`, or `del` operation is targeted at the same node or at an ancestor node.
- I Rules for collapsing insertion operations targeted at the same node, or at sibling or parent-child nodes.
- IR Rules for collapsing insertion and replacement operations (specified by means of `repN` operations) targeted at the same node, or at sibling, parent-child, parent-attribute nodes.

DEFINITION 7 (PUL REDUCTION). *Let Δ be a PUL, $\Delta^{\triangledown}$ denotes the PUL obtained by applying the reduction rules in Figure 2 in stages 1 to 9.* □

A reduced PUL may have a non-deterministic semantics, since it may contain some $\texttt{ins}^{\downarrow}$ that have not been reduced with other operations. Since a deterministic semantics may be desiderable, a stronger form of reduction, named *deterministic reduction*, resulting in a PUL with deterministic semantics, is introduced. This is realized through the introduction of a new stage 10, in which the $\texttt{ins}^{\downarrow}$ operations, that still occur in the PUL (because not reduced by the previous stages), are transformed in $\texttt{ins}^{\swarrow}$ operations.

DEFINITION 8 (DETERMINISTIC PUL REDUCTION). *Let Δ be a PUL, $\Delta^{\blacktriangledown}$ denotes the PUL obtained by applying the reduction rules in stages 1 to 10 described above.* □

Given a PUL, however, it may not have a unique determininistic reduction. The same rule may be applied more than once in the same stage and, in general, different application orders may lead to different reduced PULs. However, a *canonical form*, that is, a "standardized" reduced representation, may be useful for reasoning on PULs. To determine such a unique reduction, all the allowed rule application sequences must be guaranteed to produce the same result. An approach to enforce this property is to impose a specific order on reduction rules application. Any arbitrary order can be selected, we choose the order dictated by document order of operation targets and lexicographic order of parameters, mainly because it can be efficiently computed. An order relation $<_o$ between operations is thus defined, s.t. $op_1 <_o op_2 \Leftrightarrow t(op_1) \lhd t(op_2) \vee (t(op_1) = t(op_2) \wedge p(op_1) <_{lex} p(op_2))$, where $<_{lex}$ denotes the lexicographic ordering of the serialization of parameters. This order relation can then be extended to pairs of operations as follows: $\langle op_1, op_2 \rangle <_p \langle op_3, op_4 \rangle \Leftrightarrow (op_1 <_o op_3 \vee (op_1 = op_3 \wedge op_2 <_o op_4))$.

stage	rule	op_1	op_2	reduced
1	O1	ren(5, title)	repN(5, <a>M M)	repN(5, <a>M M)
1	I5	ins$^{\rightarrow}$(7, <a>A C)	ins$^{\rightarrow}$(7, <a>G G)	ins$^{\rightarrow}$(7, <a>A C, <a>G G)
1	I5	ins$^{\rightarrow}$(7, <a>A C, <a>G G)	ins$^{\rightarrow}$(7, <a>F C)	ins$^{\rightarrow}$(7, <a>A C, <a>G G, <a>F C)
4	IR8	repN(5, <a>M M)	ins$^{\leftarrow}$(5, <t>R</t>)	repN(5, <t>R</t>, <a>M M))
8	I15	ins$^{\rightarrow}$(7, <a>A C, <a>G G, <a>F C)	ins$^{\searrow}$(7, <m>M</m>)	ins$^{\rightarrow}$(7, <a>A C, <a>G G, <a>F C, <m>M</m>)
8	IR16	repN(5, <t>R</t>, <a>M M))	ins$^{\swarrow}$(4, <y>2004</y>)	repN(5, <y>2004</y>, <t>R</t>, <a>M M)

Table 3: Reduction steps of Example 5

DEFINITION 9 (CANONICAL FORM). *Let Δ be a PUL, $\Delta^{\blacktriangledown}$ denotes its canonical form obtained by applying rules in stages 1 to 10 and so that a rule r can be applied on a pair $\langle op_3, op_4\rangle \in \Delta$ only if no pair of operations $\langle op_1, op_2\rangle$ exists such that r applies on $\langle op_1, op_2\rangle$ and $\langle op_1, op_2\rangle <_p \langle op_3, op_4\rangle$.* □

PROPOSITION 1 (PUL REDUCTIONS). *Let Δ be a PUL applicable on a document D, and $\Delta^{\triangledown}$, $\Delta^{\blacktriangledown}$, and $\Delta^{\bar{\blacktriangledown}}$ be a reduction, a deterministic reduction and the canonical form of Δ, respectively.*

- *$\Delta^{\triangledown}$,$\Delta^{\blacktriangledown}$,$\Delta^{\bar{\blacktriangledown}}$ are substitutable to Δ on D.*
- *$|\mathcal{O}(\Delta, D)| \geq |\mathcal{O}(\Delta^{\triangledown}, D)| \geq |\mathcal{O}(\Delta^{\blacktriangledown}, D)| = |\mathcal{O}(\Delta^{\bar{\blacktriangledown}}, D)| = 1$.*
- *$\Delta^{\bar{\blacktriangledown}}$ is unique for Δ.*
- *$(\Delta^r)^r = \Delta^r$, $r \in \{\triangledown, \blacktriangledown, \bar{\blacktriangledown}\}$.* △

The proposed definitions of reduction, deterministic reduction, and canonical form suggest a straight-forward algorithm for their computations. An optimized algorithm can be obtained relying on the following considerations: (*i*) reduction rules O3 and O4 only apply to pairs of operations whose targets are bound by the A-D relationship; (*ii*) stage 10 (useful only for the deterministic and canonical reduction) requires a simple translation of operations; (*iii*) all other reduction rules only apply on pairs of operations whose targets are the same, sibling or bound by the P-C/E-A relationship. The optimized algorithm, whose complexity is $\mathcal{O}(k \cdot log(k))$, where k is the size of the PUL, can be found in [7].

EXAMPLE 5. *Let Δ be the PUL specified on the document in Figure 1 consisting of the following operations:*
ins$^{\swarrow}$(4, <year>2004</year>),
ins$^{\searrow}$(4, <month>March</month>),
ren(5, title),
ins$^{\rightarrow}$(7, <author>A.Chaudhri</author>),
ins$^{\leftarrow}$(5, <title>Report on EDBT04 ...</title>),
ins$^{\rightarrow}$(7, <author>G.Guerrini</author>),
ins$^{\rightarrow}$(7, <author>F.Cavalieri</author>),
repN(5, <author>M.Mesiti</author>),
ins$^{\downarrow}$(16, <author>P.Gardner</author>).

In Table 3 each row reports a stage of the reduction process along with the applied rule, the reduced operations, and the obtained result (only initials of tags and values are reported for the sake of conciseness). The reduced PUL is $\Delta^{\triangledown}$ = {repN(5, <y>2004</y>, <t>R</t>, <a>M M), ins$^{\rightarrow}$(7, <a>A C, <a>G G, <a>F C, <m>M</m>), ins$^{\downarrow}$(16, <t>P G</t>)} *. This reduction is not deterministic because it contains a ins$^{\downarrow}$ operation. The deterministic reduction is obtained by transforming this operation in a ins$^{\swarrow}$. The obtained deterministic reduction is not a canonical form. Indeed, rule I5 in stage 1 has been applied without taking into account the lexicographic order of operation parameters. The canonical form is $\Delta^{\blacktriangledown}$ =* {repN(5, <y>2004 </y>, <t>R</t>, <a>M M), ins$^{\rightarrow}$ (7, <a>A C, <a>F C, <a>G G, <m>M</m>), ins$^{\swarrow}$ (16, <t>P G</t>)}.

3.2 Handling Parallel PULs: PUL Integration

Given two PULs referring to the same document we aim at integrating them, that is, at obtaining a single PUL that combines their effects. Integrating two PULs in a single PUL, however, is not always possible, due to different kinds of *conflicts* among the operations in the PULs, that do not allow to simply "put them together" through a union. Intuitively, conflicts characterize the situations in which both PULs modify the same document nodes and cause a clash. Specifically, the following situations can be distinguished: (*i*) *incompatibility* between operations in the PULs, according to Definition 3, that would prevent the set of operations resulting from the union to be a PUL; (*ii*) *repetitions*: two operations of the same type, like, e.g., two insertions of the same attribute into an element, that would result in an error when the PUL is executed; (*iii*) *order-dependence*: operations whose effects are different depending on the order in which they are executed, e.g., two insertions of a first child into the same element; *(iv) overriding* of an operation by another, so that the first operation has no effect on the document due to the presence of the second one. For instance, the deletion of a node overrides a rename of that node.

We aim at identifying conflicting operations by inspecting the PULs, without computing their effects on the document. This leads us to identify the following types of conflicts, that can be detected between an operation of the first PUL and one of the second PUL:

1. *repeated modification:* repeated modifications with the same target, that results in incompatible operations;
2. *repeated attribute insertion:* repeated attribute insertions with the same target, that results in a repetition error;
3. *element insertion order:* insertion operations of the same kind (except ins$^{\downarrow}$) with the same target;
4. *local override:* overriding between operations with the same target, specifically, any operation is overridden by a deletion or a node replacement, while insertions of new children by a children replacement;
5. *non-local override:* overriding between operations with different targets, that is, operations overridden by a deletion or a replacement targeted at an ancestor of their targets.

Conflict types 1–3 are symmetric, whereas 4 and 5 are not (there is an overriding operation and an overridden one). To capture this difference, we represent conflicts as symmetric relations $\overset{ct}{\longleftrightarrow}$ ($ct \in [1..3]$) and asymmetric relations $\overset{ct}{\rightarrowtail}$ ($ct \in [4, 5]$) as defined by rules in Figure 3[3]. These conflicts need to be handled appropriately when solving conflicts and reconciling PULs. For handling them, we model a conflict as a triple, as stated by the following definition.

[3]For the sake of conciseness, in the rules we have not explicitly considered operation repN(v, []) because it is equivalent to del(v).

Repeated modification:

$$\frac{t(op_1) = t(op_2) \quad o(op_1) = o(op_2) \quad o(op_1) \in \{\texttt{ren}, \texttt{repN}, \texttt{repC}, \texttt{repV}\}}{op_1 \overset{1}{\longleftrightarrow} op_2}$$

Repeated insertion:

$$\frac{t(op_1) = t(op_2) \quad o(op_1) = o(op_2) = \texttt{insA} \quad \exists v_1 \in \mathcal{R}(p(op_1)), v_2 \in \mathcal{R}(p(op_2)) \; s.t. \; \lambda(v_1) = \lambda(v_2)}{op_1 \overset{2}{\longleftrightarrow} op_2}$$

Insertion order:

$$\frac{t(op_1) = t(op_2) \quad o(op_1) = o(op_2) \in \{\texttt{ins}^{\leftarrow}, \texttt{ins}^{\rightarrow}, \texttt{ins}^{\swarrow}, \texttt{ins}^{\searrow}\}}{op_1 \overset{3}{\longleftrightarrow} op_2}$$

Local overriding:

$$\frac{t(op_1) = t(op_2) \quad o(op_1) \in \{\texttt{repN}, \texttt{del}\} \quad o(op_2) \in \{\texttt{ren}, \texttt{repV}, \texttt{repC}, \texttt{ins}^{\swarrow}, \texttt{ins}^{\searrow}, \texttt{insA}, \texttt{ins}^{\downarrow}, \texttt{del}\} \quad \neg(o(op_1) = o(op_2) = \texttt{del})}{op_1 \overset{4}{\rightarrowtail} op_2}$$

$$\frac{t(op_1) = t(op_2) \quad o(op_1) = \texttt{repC} \quad o(op_2) \in \{\texttt{ins}^{\swarrow}, \texttt{ins}^{\downarrow}, \texttt{ins}^{\searrow}\}}{op_1 \overset{4}{\rightarrowtail} op_2}$$

Non-local overriding:

$$\frac{t(op_2) \mathbin{/\!\!/_d} t(op_1) \quad o(op_1) \in \{\texttt{repN}, \texttt{del}\} \quad o(op_2) \neq \texttt{del}}{op_1 \overset{5}{\rightarrowtail} op_2}$$

$$\frac{t(op_2) \mathbin{/\!\!/_d^a} t(op_1) \quad o(op_1) \in \{\texttt{repC}\} \quad o(op_2) \neq \texttt{del}}{op_1 \overset{5}{\rightarrowtail} op_2}$$

Figure 3: Conflicts

DEFINITION 10 (CONFLICT). *A conflict is a triple $\langle op, OS, ct\rangle$ where op is either unspecified (denoted by Λ) or an operation, OS is a set of operations, $ct \in [1..5]$ is the conflict type, and:*

- *if $ct \in [1..3]$, $op = \Lambda$ and OS is a (maximal) set of operations among which relation $\overset{ct}{\longleftrightarrow}$ holds;*
- *if $ct \in [4, 5]$, $op \neq \Lambda$ and OS is the (maximal) set of operations op' such that $op \overset{ct}{\rightarrowtail} op'$.* □

DEFINITION 11 (INTEGRATION). *Let Δ_1, Δ_2 be two PULs. Their integration $\Delta_1 \odot \Delta_2$ is defined as a pair $\langle \Delta, \Gamma\rangle$ where Γ is the set of conflicts among operations in Δ_1 and Δ_2 and $\Delta = \{op | op \in \Delta_1 \cup \Delta_2 \wedge op \notin \bigcup_{c\in\Gamma}\{\Pi_1(c)\} \cup \Pi_2(c)\}$[4] is a PUL.* □

Note that the PUL obtained as integration of two PULs Δ_1 and Δ_2 coincides to their merge (according to Definition 5) $\Delta_1 \circ \Delta_2$ when no conflict arises.

PROPOSITION 2 (INTEGRATION WITHOUT CONFLICTS). *Let Δ_1, Δ_2 be the deterministic reductions of two PULs on a document D, if the Γ component of their integration $\Delta_1 \odot \Delta_2$ is empty, then the Δ component is $\Delta_1 \circ \Delta_2$ and it is equivalent to $\Delta_1; \Delta_2$ and to $\Delta_2; \Delta_1$ on D.* △

The proof the proposition relies on the following lemma.

LEMMA 1. *Given PULs Δ_1 and Δ_2 applicable on the same document D, if two operations $op_1 \in \Delta_1$ and $op_2 \in \Delta_2$ are non-conflicting then:*

- *they are compatible according to Definition 3, thus $\{op_1, op_2\}$ is an applicable PUL;*
- *op_1 is applicable on any document in $\mathcal{O}(op_2, D)$ and op_2 is applicable on any document in $\mathcal{O}(op_1, D)$.* △

[4]Given a tuple t, $\Pi_i(t)$ denotes the i-th component of the tuple.

Algorithm 1 Conflict detection

Require: Δ_1, ... Δ_n
1: C = ∅;
2: $(\Delta_{v_1}, ..., \Delta_{v_k})$ is the partition of Δ_1 ... Δ_n according to the target node, sorted in preorder traversal;
3: **for** $k = 1$ **to** k **do**
4: $\quad C = C \cup Conflicts^{1..4}(\Delta_{v_i})$;
5: **end for**
6: Create a tree $T = (V, E)$ s.t.:
$\quad V = \{v_1, \ldots, v_k\}$;
$\quad E = \{(v_1, v_2) | (v_2 /_c v_1 \vee v_2 \mathbin{/\!\!/_d} v_1) \wedge \nexists v'.(v' \mathbin{/\!\!/_d} v_1 \wedge v_2 \mathbin{/\!\!/_d} v')\}$;
7: $C = C \cup Conflict^5(T)$;
8: $\Delta = \{op | op \in \Delta_1 \cup ... \cup \Delta_n, op \notin \{\Pi_1(C)\} \cup \Pi_2(C)\}$;
Ensure: **return** $\langle \Delta, C\rangle$;

EXAMPLE 6. *Let $\Delta_1 = \{$*insA(4, initPage = "132"), repV(8, 'MM'), repN(7, <authors/>)*$\}$ and $\Delta_2 = \{$*insA(4, lastPage = "134"), ren(5, title)*$\}$ be two PULs specified by two producers. No conflicts arise in their integration. Therefore, it is possible to merge the two PULs. Its deterministic reduction $\Delta^{\blacktriangledown}$ is $\{$*insA(4, initPage = "132", lastPage = "134"), ren(5, title), repN(7, <authors/>)*$\}$.*

An efficient algorithm for detecting conflicts among a list of PULs $\Delta_1, \ldots, \Delta_n$ can be obtained by looking at the operations in groups according to their target node. Specifically, conflicts of types 1–3 arise between operations of the same type with the same target, conflicts of type 4 between operations of different types but with the same target. Only conflicts of type 5 require to compare operations targeted at different nodes, bound by the $\mathbin{/\!\!/_d}$ relationship, and are thus more costly to determine.

Algorithm 1 first sorts the operations in the PULs according to their target nodes preorder traversal and tags each operation with the PUL it belongs to (because conflicts should be checked among operations belonging to different PULs), thus grouping operations with the same target. For each group, local conflicts (types 1-4) are detected in four stages. Each operation involved in at least a conflict is marked as conflicted. Then, non-local conflicts (type 5) are detected exploiting a tree structure whose nodes are the targets of the operations in the PULs and edges represent the P-C, or A-D relationships existing among the nodes in the original document (if a forest is obtained a dummy root node is introduced). Nodes are associated with the corresponding group of operations. Finally, through a postorder visit of the tree, the non-local conflicts are determined. The operations in each node v are compared with those recursively collected from the children of v in order to identify conflicts of type 5. When all groups have been processed, the PUL of non-conflicting operations and the identified conflicts are returned.

EXAMPLE 7. *Let $\Delta_1 = \{op_1^1 =$* insA(7, email = "catania@disi"), $op_1^2 =$ ins$^{\rightarrow}$(5, <author>G G</author>), $op_1^3 =$ repV(9, '34') *, $\Delta_2 = \{op_2^1 =$* insA(7, email ="catania@gmail"), $op_2^2 =$ ins$^{\rightarrow}$(5, <author>A C </author>), $op_2^3 =$ repV(9, '35'), $op_2^4 =$ repV(8, 'F C'), $op_2^5 =$ ins$^{\leftarrow}$(7, <author>F C </author>)*\}$ and $\Delta_3 = \{op_3^1 =$* repC(7, 'G G')*\}$ be three PULs to be integrated. Operations are first ordered and partitioned according to their target node and the following partitioning is obtained: $[\{op_1^2, op_2^2\}_5, \{op_1^1, op_2^1, op_3^1, op_2^5\}_7, \{op_2^4\}_8, \{op_1^3, op_2^3\}_9]$. Conflicts of type 1-4 are detected on operations with the same target node. By applying the rules in Figure 3 the following conflicts are identified: $cf_1 = \langle \Lambda, \{op_1^2, op_2^2\}, 3\rangle$, $cf_2 = \langle \Lambda, \{op_1^1, op_2^1\}, 2\rangle$, $cf_3 = \langle \Lambda, \{op_1^3, op_2^3\}, 1\rangle$. Then, operations are organized in the tree structure in Figure 4 for detecting conflicts among operations whose targets are bound by the A-D relationship. Through a visit of this tree, the non-local conflict $cf_4 = \langle op_3^1, \{op_2^4\}, 5\rangle$ is identified.*

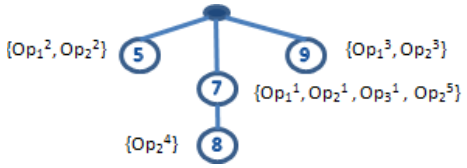


Figure 4: Tree structure for detecting non-local conflicts

PROPOSITION 3 (COMPLEXITY OF ALGORITHM 1). *Let* $\Delta_1 \ldots \Delta_n$ *be a list of PULs to be integrated. The complexity of Algorithm 1 is* $O(k^2 + a)$ *where* $k = \sum_{i=1}^{n} |\Delta_i|$ *and* a *is the number of attributes inserted through* `insA` *operations in* $\Delta_1 \ldots \Delta_n$. △

The presence of conflicts may prevent PUL integration to maintain a semantics substitutable to that of a sequential application of the PULs. Therefore, this can lead to discard the PULs and asking the PUL producers to devise different modifications of the document. However, in some of the application contexts we have considered, it could be better to solve the conflicts even if this causes some operations to be discarded (think for example the data cloud context in which the PUL producers might not be available anymore). A simple solution would be to force, whenever possible, a sequential application of the PULs (i.e., PUL aggregation, discussed in next section), but this blurs the semantics of parallel execution of PULs (by forcing a specific, arbitrary, application order) and may not be desiderable in many contexts.

For this reason, we devise an architecture in which PUL producers can specify policies that give the possibility to the PUL executor to relax (or strictly enforce) some constraints specified in its PULs to guarantee that the different PULs on the same document can be reconciled. Moreover, PUL executors are equipped with a reconciliation algorithm, named *conflict resolution algorithm*, that, given a set of conflicts, relies on the policies of the involved PUL producers to find a solution to the identified conflicts that meets such policies (even if this means that some operations contained in the PULs may be discarded), if any.

DEFINITION 12 (PUL RECONCILIATION). *Let* Δ_1, Δ_2 *be PULs,* Ξ_1, Ξ_2 *their policies,* Υ *a conflict resolution algorithm,* $\Delta_1 \odot \Delta_2 = \langle \Delta, \Gamma \rangle$ *be their integration (with* $\Gamma \neq \emptyset$*). The reconciliation of* Δ_1 *and* Δ_2 *according to* Ξ_1, Ξ_2, Υ*, denoted as* $\Delta_1 \circledcirc_{\Pi} \Delta_2$ *is* $\Delta \cup \Upsilon(\Xi_1, \Xi_2, \Delta, \Gamma)$*, if* $\Upsilon(\Xi_1, \Xi_2, \Delta, \Gamma)$ *returns a set of operations, it is undefined in case it fails.* □

The algorithm for reconciling PULs depends on the specific conflict resolution policies adopted by the executor, and on the producer policies associated with the PULs, and both of them are application-dependent. Thus, we provide an instantiation of a conflict resolution algorithm, producer policies and the corresponding reduction algorithm in Section 4, when describing our prototype PUL handling system.

3.3 Handling Sequential PULs: PUL Aggregation

Given two PULs Δ_1 and Δ_2 we aim at aggregating them, that is, at obtaining a single PUL Δ which cumulates the effects of their sequential executions $\Delta_1; \Delta_2$ in the specified order. Differently from integration where the two PULs refer to the original document, in aggregation Δ_2 can refer to the document updated through Δ_1. Moreover, there is no need of recurring to policies to reconcile conflicts, since the net result of the sequential application of

$$\text{A1)}\ \frac{op_1 = op(v, L_1)\quad op_2 = op(v, L_2)}{\Delta_1' \cup \{op_1, op_2\}, \Delta_2' \overset{A}{\multimap}_1 \Delta_1' \cup \{op(v, [L_1, L_2])\}, \Delta_2'} \quad c(op) = \texttt{i}$$

$$\text{A2)}\ \frac{op_1 = op(v, L_1)\quad op_2 = op(v, L_2)}{\Delta_1', \Delta_2' \cup \{op_1, op_2\} \overset{A}{\multimap}_1 \Delta_1', \Delta_2' \cup \{op(v, [L_1, L_2])\}} \quad c(op) = \texttt{i}$$

$$\text{B3)}\ \frac{op_1 = op(v, \_)\quad op_2 = op(v, \_)}{\Delta_1' \cup \{op_1\}, \Delta_2' \cup \{op_2\} \overset{A}{\multimap}_2 \Delta_1', \Delta_2' \cup \{op_2\}} \quad op \in \{\texttt{ren}, \texttt{repV}, \texttt{repC}\}$$

$$\text{C4)}\ \frac{op_1 = \texttt{ins}^r(v, L_1)\quad op_2 = \texttt{ins}^r(v, L_2)}{\Delta_1' \cup \{op_1\}, \Delta_2' \cup \{op_2\} \overset{A}{\multimap}_3 \Delta_1', \Delta_2' \cup \{op(v, [L_1, L_2])\}} \quad r \in \{\leftarrow, \searrow\}$$

$$\text{C5)}\ \frac{op_1 = \texttt{ins}^r(v, L_1)\quad op_2 = \texttt{ins}^r(v, L_2)}{\Delta_1' \cup \{op_1\}, \Delta_2' \cup \{op_2\} \overset{A}{\multimap}_3 \Delta_1', \Delta_2' \cup \{op(v, [L_2, L_1])\}} \quad r \in \{\rightarrow, \swarrow\}$$

$$\text{D6)}\ \frac{op_1 = op(v, T_1 \ldots T_i \ldots T_n)\ \Delta_{v'} = \{o \in \Delta_2 \mid t(o) = v'\}}{\Delta_1' \cup \{op_1\}, \Delta_2 \overset{A}{\multimap}_4 \Delta_1' \cup \{op(v, P)\}, \Delta_2 \setminus \Delta_{v'}} \quad \begin{array}{l} v' \in V(T_i), \\ T_i \models \Delta_{v'} \rightsquigarrow T_i' \\ P = T_1 \ldots T_i' \ldots T_n \end{array}$$

Figure 5: Aggregation rules

two PULs is always well defined. The aggregation of two PULs consists of their merge, provided that the operations in the second PULs do not depend on the operations specified in the first one. Otherwise, dependencies must be removed before merging the two PULs. Specifically, we identify the following dependencies and define the rules in Figure 5 for their removal. The position of nodes inserted by operations in Δ_2 depends on the position of a node inserted by an insert operation of the same type on the same node in Δ_1. Rules A1, A2, C4, and C5 cumulate the effect of insert operations of the same type on the same node, so that the order of inserted nodes in the aggregated PUL is one of those obtainable by the sequential execution of Δ_1 and Δ_2. Rule B3 deals with overriding operations in Δ_2 by eliminating the overridden operation specified in Δ_1. Rule C6, finally, handles the situation in which a set of operations in Δ_2 refers to nodes of a tree T_i which is a parameter of an operation *op* in Δ_1. In this case, the modifications specified for T_i in Δ_2 are applied and removed from the operations of Δ_2. Note, however, that in case the first PUL specifies a `repC` operation on a node v and the second PUL an $\texttt{ins}^{\swarrow}$, $\texttt{ins}^{\downarrow}$ or $\texttt{ins}^{\searrow}$ operation on v a more complex treatment is required. For the sake of conciseness, we omit the discussion of this situation. The interested reader may refer to the extended version [7] of this paper.

DEFINITION 13 (AGGREGATION). *Let* Δ_1, Δ_2 *be two PULs. Their aggregation* $\Delta_1 \multimap \Delta_2$ *denotes the PUL obtained through the application of operator* $\overset{A}{\multimap}$ *in Figure 5 according to stages* 1 *to* 4*. Specifically, let* Δ_1^f,Δ_2^f *be the result of* $\langle \Delta_1, \Delta_2 \rangle \overset{A}{\multimap} \langle \Delta_1^f, \Delta_2^f \rangle$*. Then* $\Delta_1 \multimap \Delta_2 = \Delta_1^f \cup \Delta_2^f$. □

PROPOSITION 4 (AGGREGATION). *Let* Δ_1 *be a PUL applicable on a document* D *and* Δ_2 *be a PUL applicable on any document in* $\mathcal{O}(\Delta_1, D)$*. Their aggregation* $\Delta_1 \multimap \Delta_2$ *is substitutable to* $\Delta_1; \Delta_2$ *on* D. △

Algorithm 2 for efficiently aggregating a list $\Delta_1, \ldots, \Delta_n$ of PULs, with the property that Δ_k $(1 \leq k \leq n)$ is applied on the original document D modified according to $\Delta_1, \ldots, \Delta_{k-1}$, has been defined. The algorithm relies on the use of an hash table H indexed on target nodes of the operations in the PULs and on nodes of the trees inserted by means of `ins` or `rep` operations. The value of $H(v)$ is a pair $\langle state, ops \rangle$ representing whether v belongs to the original document (`old`) or is being inserted by the considered PULs (`new`) and *ops* is the list of operations whose target is v.

The hash table is populated as follows. If the target v of an operation $op(v, P)$ in a PUL Δ_k $(1 \leq k \leq n)$ has been already

included in H, $op(v, P)$ is appended to the operations associated with v, otherwise, it means that v belongs to the original document and no operations have been already specified for it. Therefore, $H(v)$ is associated with $\langle \texttt{old}, \{op(v, P)\}\rangle$. Then, each node v_n, contained in the parameter of *op*, is new with respect to the original document. Therefore, $H(v_n)$ can be defined and the value $\langle \texttt{new}, \{\}\rangle$ associated with it. Once the hash table has been populated with the operations in a PUL Δ_k, it contains an entry for each node v belonging to the original document (and target of an operation) or being inserted through an operation of a PUL $\Delta_i, i \leq k$, and the update operations specified on it. The aggregation rules in Figure 5 can be applied[5] on the operations in $H(v).ops$ for each v in the original document D, where $op_1 \in H(v).ops \setminus \Delta_k$ and $op_2 \in H(v).ops \cap \Delta_k$. Instead, the operations whose target v is not a node of D can be directly applied on the tree containing v (note that this corresponds to the application of rule $D6$ in Figure 5). Finally, the algorithm collects and returns all the operations in the hash table.

EXAMPLE 8. *Consider the following PULs to be aggregated. For the sake of conciseness, the node identifier is reported as superscript of the node.* $\Delta_1 = \{\texttt{ins}^{\searrow}(3, \texttt{<article}^{24}\texttt{><title}^{25}\texttt{>XML}^{26}\texttt{</title></article>}), \texttt{repV}(10, \texttt{'13'})\}$, $\Delta_2 = \{\texttt{ins}^{\searrow}(24, \texttt{<author}^{27}\texttt{>G G}^{28}\texttt{</author>}, \texttt{<author}^{29}\texttt{>M M}^{30}\texttt{</author>}), \texttt{ren}(5, \texttt{title})\}$, $\Delta_3 = \{\texttt{repN}(29, \texttt{<author}^{31}\texttt{>F C}^{32}\texttt{</author>}), \texttt{ren}(5, \texttt{name}), \texttt{repV}(26, \texttt{'On XML'})\}$. *Aggregation* $\Delta_1 \multimap \Delta_2$ *is* $\{\texttt{ins}^{\searrow}(3, \texttt{<article}^{24}\texttt{><title}^{25}\texttt{>XML}^{26}\texttt{</title><author}^{27}\texttt{>G G}^{28}\texttt{</author><author}^{29}\texttt{>M M}^{30}\texttt{</author></article>}), \texttt{repV}(10, \texttt{'13'}), \texttt{ren}(5, \texttt{title})\}$ *in which* $\texttt{ins}^{\searrow}$ *of* Δ_2 *has been executed on the parameter of the* $\texttt{ins}^{\searrow}$ *of* Δ_1. *Moreover, aggregation* $\Delta_1 \multimap \Delta_2 \multimap \Delta_3$ *is* $\{\texttt{ins}^{\searrow}(3, \texttt{<article}^{24}\texttt{><title}^{25}\texttt{>On XML}^{26}\texttt{</title><author}^{27}\texttt{>G G}^{28}\texttt{</author><author}^{31}\texttt{>F C}^{32}\texttt{</author></article>}), \texttt{repN}(10, \texttt{special} = \texttt{"2bis"}^{33}), \texttt{ren}(5, \texttt{name})\}$. *The last author of* $\texttt{article}^{24}$ *and text node* 26 *have been changed by applying aggregation rule D6, and* ren *in* $\Delta_1 \multimap \Delta_2$ *has been removed because it is overridden by the same operation on the same node in* Δ_3 *(aggregation rule B3).*

PROPOSITION 5 (COMPLEXITY OF ALGORITHM 2). *Let* $\Delta_1 \ldots \Delta_n$ *be a list of PULs to be aggregated. The complexity of Algorithm 2 is* $\mathcal{O}(k + p)$ *where* $k = \sum_{i=1}^{n} |\Delta_i|$ *and p is the number of nodes inserted through insert operations in* $\Delta_1 \ldots \Delta_n$. △

Algorithm 2 Aggregation

Require: Δ_1, ... Δ_n
1: H is a new hash table;
2: **for** $k = 1$ **to** n **do**
3: **for each** $op(v, P) \in \Delta_k$ **do**
4: **if** $H(v)$ is not defined **then**
5: $H(v) = \langle \texttt{old}, \{op(v, P)\}\rangle$;
6: **else**
7: $H(v).ops = H(v).ops \cup \{op(v, P)\}$;
8: **end if**
9: **for each** $v_n \in V(P)$ **do**
10: $H(v_n) = \langle \texttt{new}, \emptyset\rangle$;
11: **end for**
12: **end for**
13: **for each** $v \in Dom(H)$ **do**
14: **if** $H(v).state = \texttt{old}$ **then**
15: $applyRules(H(v).ops \setminus \Delta_k, H(v).ops \cap \Delta_k)$;
16: **else**
17: $applyRules(\cup_{v' \in Dom(H)} H(v').ops, H(v).ops)$;
18: **end if**
19: **end for**
20: **end for**
Ensure: **return** $\cup_{v \in Dom(H)} H(v).ops$;

4. THE PUL HANDLER SYSTEM

In our reference architecture, PUL execution is decoupled from PUL production. PUL production is performed by evaluating an XQuery Update expression on a document, whereas PUL execution makes the updates in a PUL effective on a document. We refer to the node collecting PULs and making them effective on the document as *PUL executor*, while all the other nodes are referred to as *PUL producers*. We assume a single PUL executor per document, representing the node handling the *master* or authoritative version of a document, to avoid dealing with synchronization problems between different replicas of the same document, but our approach can be generalized to multiple executors as well. To decouple PUL production from their execution, we have modified the Qizx library to produce PULs and to accept PULs as input. PULs are represented as XML documents containing the serialization of each PUL operation along with the identifiers and labels of the target nodes. Decoupling PUL production from PUL execution introduces additional costs in serializing and exchanging PULs on the network if compared to a scenario where PULs are executed locally and right after being produced. However, the PUL executor may take advantage of the knowledge of the exact nodes subjected to modification to execute PULs in streaming, without the need to access and load the whole document in main memory.

Our reasoning algorithms are made available both to the executor and to the producers. Thus, for instance, a producer may decide to aggregate its PULs before sending them to the executor or it can send disaggregated PULs that may be then aggregated by the executor before executing them.

In the remainder of this section, we discuss the most relevant problems we have to cope with in developing the system, that is, node identification and structural information, then present conflict resolution policies and, finally, provide an experimental evaluation of the developed system.

4.1 Node Identification and Structural Information

In the previous sections we have denoted document nodes through the node itself, since no need for explicit node identifiers arises. If PULs are produced and executed locally in the same process, as in current XQuery Update implementations, nodes are identified by means of the in-memory document representation. When PULs have to be serialized and exchanged across the network, explicit identifiers need to be assigned to document nodes. Such identifiers need to be unique in the document, immutable, not reusable (that is, identifiers of deleted nodes are not re-assigned to other nodes). Given a document, identifiers are assigned to nodes by means of an appropriate algorithm which is agreed upon and employed by all the PUL producers that manipulate that document. In this way, identifiers of document nodes belonging to the master document stored at the executor can be uniformly assigned by the producers and PULs containing operations targeted at them exchanged and later executed. We need however to decide when identifiers are assigned to new nodes inserted by the execution of the PUL itself. Specifically, in the aggregation context, a producer may execute sequential PULs on its local copy of the document, and these PULs

[5]Given O and O' sets of operations, $applyRules(O, O')$ denotes the application of the aggregation rules for $op_1 \in O$ and $op_2 \in O'$.

may need to refer to nodes inserted during previous PULs. Identifiers are thus assigned to nodes when PULs are applied (either locally by a producer or on the master document by the executor). The producer relies on these identifiers for its local reasoning (if any). To ensure that the identifiers simultaneously assigned by different producers do not clash two alternative approaches can be adopted: either each producer has an assigned identification space and assigns identifiers in this space, or local identifiers assigned by producers are replaced by "global" ones assigned by the executor when updating the authoritative copy of the document.

For what concerns the structural information, as we discussed in Section 2, our reasoning does not require accessing the document. Rather, to check whether one of the structural relationships of Table 1 holds among two nodes (targets of operations in the PUL) a labeling scheme is used. Labels are associated with nodes in the document tree and attached to the target nodes of the operations specified in a PUL. During reasoning, structural information is only needed for the portion of the document already present before the start of the first PUL, thus the labeling is only modified when updates are applied by the executor on the authoritative copy of the document. Different labeling schemes can be adopted in our context with the property to be tolerant to updates, that it, document updates should not lead to relabeling of nodes. The one we have adopted is the Zhang containment, encoded by means of the CDQS[15], or alternatively the CDBS[14], encoder. This labeling scheme allows us to evaluate all the relationships in Table 1 with the exception of left sibling ($\lhd_s$) and the possibility to distinguish between attributes/children of a given element ($/_c$ and $/_a$). For this purpose, we extended the labeling scheme to include the node type and the identifier of the left sibling node (if any). The obtained labeling scheme allows to determine all the relationships in Table 1 in constant time.

4.2 Conflict Resolution Policies

Different conflict resolution policies can be specified by the PUL producers in order to characterize the approach to solve conflicts during PUL integration. Consequently, the conflict resolution algorithm relies on the specified policies. Its behaviour may be very diverse depending on the context in which our system is used, but it should at least guarantee the strict observation of the policies specified by the PUL producers. Specifically, the algorithm must fail whenever it cannot identify a *valid reconciliation*: a conflict-free PUL that satisfies all the policies of the involved PUL producers.

In our system we have adopted the following policies that may be eventually specified by the PUL producers.

- *Preservation of insertion order.* The specified order for inserted nodes must not be altered by operations of other PULs.
- *Preservation of inserted data.* Inserted data (through `repN`, `repC`, `repV` or `ins`) must occur in the final document.
- *Preservation of removed data.* Removed data (through `repN`, `repC`, `repV` or `del`) must not occur in the final document.

Given a set of conflicts, dependencies among conflicts can be identified. Thus, by solving a specific conflict other dependent conflicts may be solved as well or become unsolvable according to the specified policies. Therefore, different resulting PULs may be obtained depending on the order in which conflicts are processed and solved. Moreover, the order of conflict processing may influence the number of discarded operations and thus the gap between the semantics of the original PUL and that of the reconciliated PUL.

Algorithm 3 Best-effort conflict resolution

Require: $conflicts, \Delta, policies$
1: $E = \emptyset; \Delta_g = \emptyset;$
2: let $(\langle op_1, OS_1, ct_1\rangle, ..., \langle op_n, OS_n, ct_n\rangle)$ be the list $conflicts$ ordered as defined;
3: **for** $i = 1$ **to** n **do**
4: $op_i = op_i$ if $op_i \notin E, \Lambda$ otherwise;
5: $OS_i = OS_i \setminus E$;
6: $\langle solved, gen, excl\rangle = solve(\langle op_i, OS_i, ct_i\rangle, policies)$;
7: **if** ($solved$ = **true**) **then**
8: $\Delta_g = \Delta_g \cup gen$;
9: $E = E \cup excl$;
10: **else**
11: **abort**; – *conflicts not solvable*
12: **end if**
13: **end for**
Ensure: **return** $(\Delta_g \cup \bigcup_{i=1..n}(OS_i \cup \{op_i\})) \setminus E$;

Determining the best ordering of conflicts, for a given sets of policies, easily become computationally prohibitive, especially when a large number of conflicts must be processed. Thus, the algorithm needs to balance different requirements: maximizing the probability of finding a valid reconciliation, minimizing execution time, and the number of discarded operations.

Algorithm 3 for conflict resolution aims to solve as many conflicts as possible, with a very low computational cost, by excluding operations involved in a conflict unless forbidden by the specified policies. The algorithm works as follows: it first orders the conflicts according to a given criterion and then processes a conflict at a time aiming at solving it. The resolution of a conflict may exclude one or more operations from the PUL making unnecessary to further consider them in the other conflicts they eventually belong to. Some of the conflicts that still need to be processed may thus be automatically solved. Specifically, a conflict is automatically solved when it is symmetric and it involves at most one operation, or it is asymmetric and either the overrider operation does not belong to the PUL anymore or the overridden operation set is empty.

Conflicts are ordered by identifying the *focus node* of a conflict. The focus node is the common target for symmetric conflicts while it is the overrider operation target for asymmetric ones. A conflict c_1 precedes another conflict c_2 if the focus node of c_1 precedes the focus node of c_2 according to document order ($\lhd$). When the focuses of the conflicts coincide, their ordering is determined by the conflict type and the operations contained in the conflicts exploiting the following precedence: (*i*) conflicts of type 1 among `repN` operations, (*ii*) conflicts of type 4 with a `repN` overriding operation, (*iii*) conflicts of type 1 among `del`s, (*iv*) conflicts of type 4 with a `del` overriding operation, (*v*) conflicts of type 1 among `repC`s, (*vi*) conflicts of type 4 with a `repC` overriding operation, (*vii*) remaining conflicts of type 1 and type 2 , (*viii*) conflicts of type 3 and, finally, (*ix*) conflicts of type 5. Ordering the conflicts relying on the focus node allows us to consider a conflict on a focus node v only when we are sure that v will be present in the final document, that is, when all the operations that remove v have been excluded. Moreover, this ordering criterion avoids inconsistencies in resolution, as a conflict is processed only when any involved operation cannot be later excluded by a subsequent conflict resolution.

A conflict is processed through the $solve$ function. Two are the possible outcomes: (*i*) the conflict cannot be solved according to the specified policies and the entire reconciliation is aborted; (*ii*) the conflict is solved and the sets of operations $excl$ and gen are generated. $excl$ is the set of operations excluded from the final PUL

(and thus also from the conflicts still to be processed); and gen is (possibly empty) set of operations generated when solving an order (type 3) conflict. The specific behaviour of this function depends on the conflict type, as follows:

- *Asymmetric conflicts.* Either the overriding or overridden operations are excluded. To increase the number of automatically solved conflicts, and thus the probability of finding a valid reconciliation, the overridden operations are excluded.
- *Order conflicts.* All the involved insertion operations are excluded and a new insertion operation of the same type having as parameter the concatenation of the operation parameters in a certain order is generated.
- *Non-order symmetric conflicts.* All but one of the involved operations are excluded.

When all conflicts have been processed a valid reconciliation is identified, which is composed of (*i*) the conflicted operations which have not been excluded, (*ii*) the original PUL operations not involved in any conflict and (*iii*) the set of operations generated when solving order conflicts.

EXAMPLE 9. *Consider the conflicts detected in Example 7. The algorithm identifies the focuses of these conflicts and orders them according to the specified criteria. The resulting list, where the focus node is reported as a superscript, is* $[cf_1^{[5]}, cf_2^{[7]}, cf_4^{[7]}, cf_3^{[9]}]$.

Suppose that the first producer specifies that insertion order and inserted data must be preserved, the second producer does not pose any constraint, and the last producer specifies a constraint on inserted data only. The algorithm processes the conflicts in the specified order according to these policies. The first conflict is solved by excluding both the involved operations and introducing the new operation $\texttt{ins}^{\rightarrow}(5, \texttt{<author>G G</author>}, \texttt{<author>A C</author>})$ *that adheres to the "preservation of insertion order" policy of the first producer. The second conflict is solved by excluding operation* op_2^1 *to adhere to the "preservation of inserted data" policy of the first producer. Other conflicts are solved analogously by excluding operations* op_2^4 *and* op_2^3*, respectively. The resulting PUL is* $\{\texttt{ins}^{\rightarrow}(5, \texttt{<author>G G</author>}, \texttt{<author>A C</author>}), op_1^1, op_1^3, op_3^1, op_2^5\}$.

If, by contrast, all the three producers required the preservation of insertion order, when processing the first conflict no ordering of the inserted data would satisfy the producers policies. Thus, the reconciliation would fail.

PROPOSITION 6 (COMPLEXITY OF ALGORITHM 3). *Let* $c_1, \ldots, c_n$ *be a list of conflicts to be solved. The complexity of Algorithm 3 is* $\mathcal{O}(n \log n + c_{op})$ *where* c_{op} *is the total number of operations in* $c_1, \ldots, c_n$. △

4.3 Experimental Results

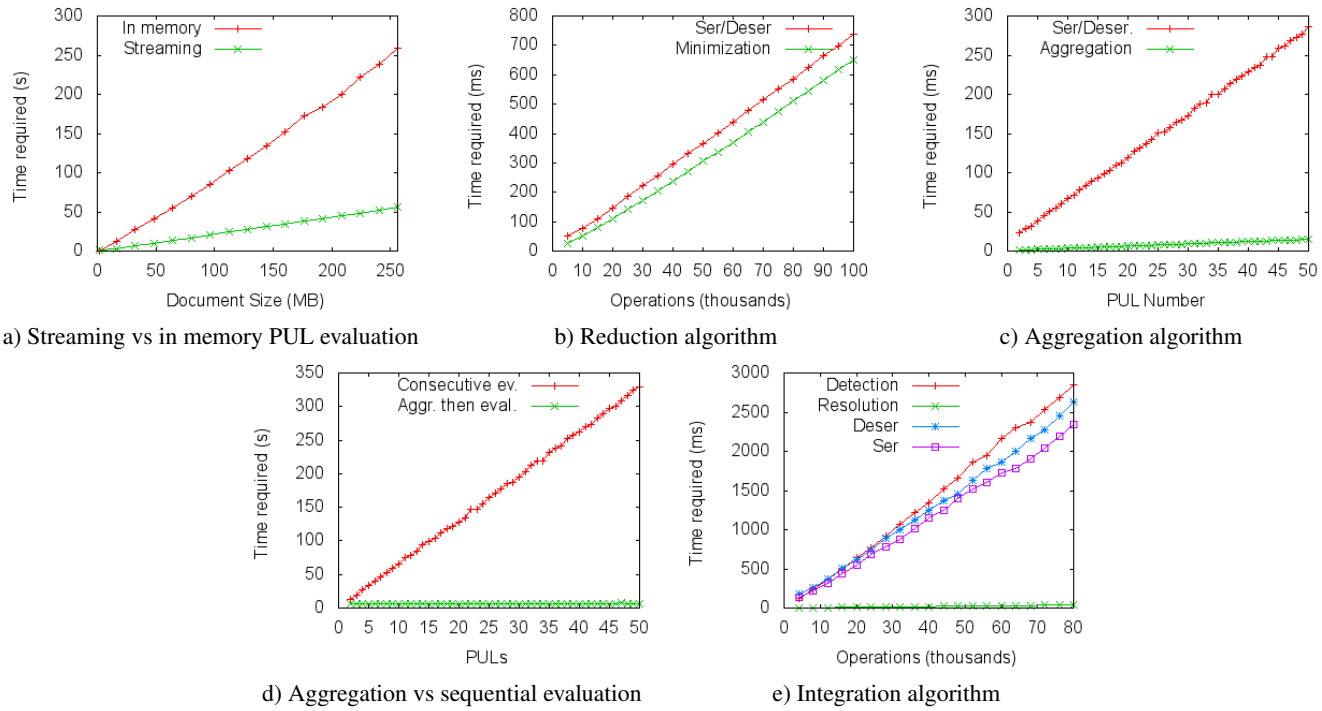
To assess the computational costs and advantages of our algorithms we exploit synthetic XQuery Update expressions and their corresponding PULs generated by means of the modified Qizx library. Documents of various sizes, ranging from 1MB to 256MB, have been produced by means of the XMark data generator. Synthetic PULs have been generated as well, with a varying number operations, equally distributed among the operation types, specified on these documents. In these experiments, the node identifiers and labeling, fundamental for the application of our algorithms, have been stored within the related documents. Our test machine employs an Intel I5 760 processor, coupled with 16GB of RAM, and runs the 64bit version of the Sun Java JDK v.1.6.20.

Two approaches have been implemented for the evaluation of a PUL on a given document: an adaptation of the Qizx library for directly handling PULs and a novel streaming evaluation algorithm. The former loads the entire document in memory, applies the PUL and finally serializes the document back to disk. The latter, instead, is based on a specialized SAX parser and writer. The original document is parsed generating a sequence of SAX events, that are transformed on-the-fly applying the operations specified in the PUL and immediately serialized to disk. In both algorithms, the entire source document is processed and, while operations are applied, new labels and identifiers are generated. A noteworthy advantage of the streamed evaluation is that no in-memory representation of the document is needed, effectively decoupling the main memory requirements from the document size. Note that the amount of information required to compute new labels depends on the number of operations rather than on the document size.

We first investigated the correlation between the number of operations in a PUL and its evaluation time on a document of the considered sizes. The experiment pointed out that the number of operations in the PUL has a negligible effect on the evaluation time, which is instead largely affected by the number of nodes to be read/written and thus by the input and output document sizes. Then, the actual performance benefit of using streaming evaluation over the extended Qizx in-memory algorithm has been analyzed by comparing the execution time for evaluating a PUL with a thousand operations. Figure 6.a) shows that streaming evaluation is significantly more efficient and its advantage with respect to the in-memory evaluation increases with document size. Specifically, on the considered documents, streaming evaluation is about 3 times faster and this speed up is directly proportional to document size.

PUL reduction. PUL reduction complexity has been experimentally evaluated by analyzing the execution time to deserialize, reduce and then reserialize PULs whose sizes ranges from 5000 to 100,000 operations, with approximatively a successful rule application every 10 operations. The trend of execution times reported in Figure 6.b) of the algorithm is compliant with the presented complexity and the time spent for the actual reduction is smaller than PUL serialization and deserialization. In the current settings, the number of operations in a PUL does not significantly affect its execution time. Therefore, this operation does not produce benefits in terms of performance. However, it is still important for reasoning on equivalence/substitutability of PULs.

PUL aggregation. Applying the aggregation of a list of PULs, rather than a sequential application of each PUL in the list, may lead to a noteworthy advantage in terms of execution time as the document has to be processed only once. Thus, we analyzed under varying circumstances whether the aggregation cost may outbalance any gained advantage. We first analyzed the execution time for the deserialization, aggregation and reserialization of a varying number of PULs, with a different ratio of operations on nodes not present in the original document and different number of operations. We observed that the ratio of operations on nodes in the original document has a negligible effect on execution time, while there is an almost linear dependency between the aggregation time and both the number of operations in each PUL and the number of PULs. We report in Figure 6.c) the cost of aggregating an increasing number of PULs each containing 1000 operations, half of them specified on original document nodes. As the graph shows, the deserialization/serialization cost dominates the aggregation cost which is under 5 msec. even for the aggregation of 15000 operations equally divided in 15 PULs. Figure 6.d) compares the execu-



a) Streaming vs in memory PUL evaluation

b) Reduction algorithm

c) Aggregation algorithm

d) Aggregation vs sequential evaluation

e) Integration algorithm

Figure 6: Experiments

tion time for the aggregation and (streaming) evaluation of a list of PULs contrasting it to the cost of their sequential (streaming) evaluation. Results show that in contexts where direct access to a given node in a document is not possible, the advantage of aggregating a list of PULs and then applying the resulting PULs respect to the sequential application of the PULs is significant and increases rapidly with the number of PULs in the list. The cost of aggregating the PULs is not even noticeable on the graph.

PUL integration and conflict resolution. The cost of the integration algorithm depends on the number of PULs, operations, conflicts, operations involved in a conflict, conflict type and on how a conflict resolution affects other conflicts. As for aggregation, we first analyzed how the integration/evaluation time varies in function of these variables, discovering that given a certain number of operations involved in a conflict, the number of PULs and the conflict type have a marginal impact on execution time. Figure 6.e) reports the results of integrating 10 PULs, containing a varying number of operations, from 4K to 80K each. Half of them are involved in conflicts, which contain an average of 5 operations. We also ensured that only 1/5 of the conflicts are solved thanks to the removal of operations in other conflicts and that the remaining conflicts are unaffected and equally distributed with respect to their type. Results show that integration is a cost effective operation.

5. RELATED WORK

The notions of PUL, function `mergeUpdates` for merging two PULs, and PUL semantics in terms of function `applyUpdates` are defined in W3C XQuery Update Facility [22]. XQuery Scripting Extension [21] is a further W3C XQuery extension that supports control constructs and allows to apply update operations not only with XQuery Update *snapshot* semantics (i.e., update application is delayed to the end of the query), but also with an *iterative* one: immediate update application and explicit left-to-right evaluation order. A semantics for XQuery Updates has been defined by Benedikt and Cheney in [5].

Most of the approaches to update reasoning are static. An early approach in the relational and object setting is [13] where the notion of potential conflict is employed to determine whether set-oriented update sequences have deterministic semantics (which may not be the case because of sharing). In the XML context, static analysis has been proposed for various properties and various languages [6]. Most of the proposals focus on non-interference (also referred to as independence) between updates and queries, or between two updates (finalized to update optimization). Exact approaches for static analysis of independence require to consider restricted languages (e.g., navigational subsets) [2, 3, 19] while the alternative is developing approximate approaches [4]. An approximate approach is taken for analyzing update commutativity in [12] for updates with iterative as well as snapshot semantics.

For what concerns dynamic handling of updates, relevant related work are in the contexts of versioning, of distributed execution and of collaborative editing. In the context of versioning, some approaches focus on *delta* management, where a delta represents the transformation between two versions of the same document [9]. Completed deltas [16] are deltas that contain additional information about operations and can be inverted and composed. PULs do not include such additional information. Moreover, deltas can be deduced by comparing different versions, and most research effort has been devoted to develop efficient algorithms for obtaining compact deltas. Our approach is instead closer to [11], addressing PUL composition for modeling the changes made by an XQuery Scripting Extension [21] program as a single PUL rather than as a sequence of PULs. Composition allows to summarize the effect of several, possibly interdependent, PULs as a single (local) PUL. Thus, it has the same goal as our aggregation. However, it is performed when applying a PUL and relies on backpointers

from nodes in the document to the corresponding nodes in the local PUL. By contrast, in our framework we work on PULs with no accesses to relative documents. Composition work on normalized PULs (i.e., PULs containing at most one update primitive of each type for each target, and replacing $\texttt{ins}^{\downarrow}$ with an another insert primitive). Reduced deterministic PULs of our approach are normalized, while the converse does not hold. Local PULs are kept normalized during the composition process in [11].

In the distributed context, a union operation is considered in XRPC [23] for cumulating the PULs a peer has handled in different XRPC calls for a given update query guaranteeing isolation of transactions. The definition of this operation is given in [24] as concatenation, but it is bound to the deterministic update semantics used in MonetDB/XQuery. In the same context, [25] deals with the problems of respecting node identities and preserving structural properties in decomposing update expressions for distributed execution through a dynamic XML projection technique.

Collaborative editing, finally, denotes the process by which separate users are allowed to work concurrently on the same data. *Operation-based* approaches [1, 17] have been proposed for maintaining consistency of the copies of the shared document. They allow local operations to be executed immediately after their generation while remote operations need to be transformed against the other operations, guaranteeing their eventual consistency. The transformations are performed so that the intentions of the users are preserved and, at the end, the copies of the documents converge. This shows some similarity with our integration operation.

6. CONCLUSIONS

In the paper we have investigated three relevant operations on PULs, namely PUL reduction, integration and aggregation, and we have developed and experimentally evaluated algorithms implementing them. The operations can be combined, that is, it would be useful to apply reduction after integration/aggregation, to get a more compact (and deterministic, if desired) PUL. Note that applying reduction before integrating/aggregating would lead to a different result and is less preferable because it may introduce additional conflicts involving operations not appearing in the original PULs. An experimental evaluation of combined use of PULs operations on real PULs would be interesting, but not many data on real PULs are available. Our experimental evaluation could be extended in other directions as well, explicitly taking distribution issues into account, as well as considering documents that are stored, and possibly shredded, inside a DBMS.

An important property of our approach is that the reasoning on PULs does not require to access the document they refer to, provided that some structural relationships among nodes of the document can be determined through labeling. Another interesting topic we will consider as future work is the study of PUL inversion, but this requires either the extension of the PUL production algorithm or the access to the document the PUL refers to.

In the current version of our system, node identifiers and labeling have been stored within the document. This solution significantly affects the size of the original documents (which becomes approximatively 3 times bigger) and PUL application time. As future work we plan to consider the possibility to use external data structures to store this information and to check the advantages in terms of performances.

Acknowledgments. We wish to thank Dana Florescu for a useful discussion allowing us to present our work in a more general setting.

7. REFERENCES

[1] A. H. Davis, C. Sun, and J. Lu. Collaborative Editing of XML Documents. An Operational Transformation Approach. In *ACM GROUP*, 2001.

[2] M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Adding Updates to XQuery: Semantics, Optimization, and Static Analysis. In *XIME-P*, 2005.

[3] M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Verification of Tree Updates for Optimization. In *CAV*, LNCS(3576), pp 379–393. 2005.

[4] M. Benedikt and J. Cheney. Schema-Based Independence Analysis for XML Updates. *PVLDB*, 2(1):61–72, 2009.

[5] M. Benedikt and J. Cheney. Semantics, Types and Effects for XML Updates. In *DBPL*, LNCS(5708), pp 1–17. 2009.

[6] M. Benedikt and J. Cheney. Static Analysis of Declarative Updates. In *EDBT/ICDT Workshops*. ACM, 2010.

[7] F. Cavalieri, G. Guerrini, and M. Mesiti. Dynamic Reasoning on XML Updates. TR, U. of Genova, September 2010 `ftp.disi.unige.it/person/GuerriniG/reports/pulprocTR.pdf`.

[8] S.-Y. Chien, V. J. Tsotras, and C. Zaniolo. XML Document Versioning. *SIGMOD Record*, 30(3):46–53, 2001.

[9] G. Cobena, S. Abiteboul, and A. Marian. Detecting Changes in XML Documents. In *ICDE*, pp 41–52, 2002.

[10] C. E. Dyreson and F. Grandi. Temporal XML. In *Encyclopedia of Database Systems*, pp 3032–3035. 2009.

[11] G. Fourny, D. Florescu, D. Kossmann, and M. Zacharioudakis. A Time Machine for XML: PUL Composition. In *XML Prague*, 2010.

[12] G. Ghelli, K. H. Rose, and J. Siméon. Commutativity Analysis for XML Updates. *ACM TODS*, 33(4), 2008.

[13] C. Laasch and M. H. Scholl. Deterministic Semantics of Set-Oriented Update Sequences. In *ICDE*, pp 4–13, 1993.

[14] C. Li, T. Ling, and M. Hu. Efficient Processing of Updates in Dynamic XML Sata. In *ICDE*, pp 13, 2006.

[15] C. Li, T. Ling, and M. Hu. Efficient Updates in Dynamic XML Data: from Binary String to Quaternary String. *VLDB Journal*, 17(3):573–601, 2008.

[16] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-Centric Management of Versions in an XML Warehouse. In *VLDB*, pp 581–590, 2001.

[17] G. Oster, P. Urso, P. Molli, and A. Imine. Data Consistency for P2P Collaborative Editing. In *CSCW*, pp 259–268, 2006.

[18] PIXwere. QIZX. An Open-source XQuery Processor, 2010.

[19] M. Raghavachari and O. Shmueli. Conflicting XML Updates. In *EDBT*, LNCS(3896), pp 552–569. 2006.

[20] W. Vogel. Eventually Consistent. *Communications of the ACM*, 52(1), 2009.

[21] W3C. XQuery Scripting Extension 1.0, April 2010.

[22] W3C. XQuery Update Facility 1.0, June 2009.

[23] Y. Zhang and P. A. Boncz. XRPC: Interoperable and Efficient Distributed XQuery. In *VLDB*, pp 99–110, 2007.

[24] Y. Zhang and P. A. Boncz. Loop-Lifted XQuery RPC with Deterministic Updates. Technical Report INS-E0607, CWI, Amsterdam, The Netherlands, November 2006.

[25] Y. Zhang, N. Tang, P. A. Boncz. Projective Distribution of XQuery with Updates. *IEEE TKDE*, 22(8):1059–1076, 2010.