Algebraic incremental maintenance of XML views*

Angela Bonifati
CNR, Italy
bonifati@icar.cnr.it

Martin Goodfellow
University of Strathclyde, UK
Martin.Goodfellow@cis.strath.ac.uk

Ioana Manolescu
INRIA Saclay, France
Ioana.Manolescu@inria.fr

Domenica Sileo
University of Basilicata, Italy
Domenica.Sileo@gmail.com

ABSTRACT

Materialized views can bring important performance benefits when querying XML documents. In the presence of XML document changes, materialized views need to be updated to faithfully reflect the changed document. In this work, we present an algebraic approach for propagating source updates to XML materialized views expressed in a powerful XML tree pattern formalism. Our approach differs from the state of the art in the area in two important ways. First, it relies on set-oriented, algebraic operations, to be contrasted with node-based previous approaches. Second, it exploits state-of-the-art features of XML stores and XML query evaluation engines, notably XML structural identifiers and associated structural join algorithms. We present algorithms for determining how updates should be propagated to views, and highlight the benefits of our approach over existing algorithms through a series of experiments.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

Keywords

XML view maintenance, XML updates, XML query processing

1. INTRODUCTION

XML data management has reached by now a certain level of maturity, with many commercial and open-source systems supporting the W3C's XPath and XQuery [35] standards for querying XML documents. The complexity of XPath and XQuery and of the XML data itself raised many performance challenges. One direction of work towards improving the performance of XML query evaluation consists of relying on materialized views (or caches) storing precomputed query results, based on which queries can be answered more speedily than by using the original documents only [6, 18, 25, 29, 38]. Such techniques have been shown to improve query evaluation performance by up to several orders of magnitude.

*This work was performed while M. Goodfellow and D. Sileo were visiting INRIA Saclay. The work has been partially funded by Agence Nationale de la Recherche, decision ANR-08-DEFIS-004.

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

More recently, the W3C has also proposed an update extension to the XQuery language, namely XQuery Update [36]. XQuery Update is gradually being implemented in XML data management platforms. When materialized views are used as a performance-enhancing tool, updates to the XML database raise two new problems. First, one has to determine whether the result of a view should change due to the update (or, as often said, whether the update *affects* the view). This problem has been studied recently in [11, 12], and in the particular case when XML schemas are available to describe the documents in [9]. Second, when a view is indeed affected, a related issue is how to efficiently update the view to reflect the update. This second problem is the main focus of this paper.

Figure 1 illustrates the view maintenance problem in this context. Evaluating the view v over the XML document d leads to materializing $v(d)$. An XML update transforms d into d', and correspondingly the affected view v should be transformed into $v(d')$. One possibility is to evaluate v from scratch on the modified document d'. Instead, our focus is on incrementally modifying v by adding, removing, or modifying data as needed, to transform it into $v(d')$, without recomputing it.

The incremental maintenance of XML materialized views has been considered in previous works [13, 16, 17, 18, 19, 28, 31]. Maintaining XML views over relational databases is studied in [16]. The maintenance of boolean XPath queries is studied in [13]. Views expressed in a richer XPath dialect are considered in [30, 31], which focus on *node-level updates*, that is, they consider updates which add or remove exactly one node to/from the document. Node-level updates are propagated to XQuery views in [17]. While node-level updates are conceptually simple, updates in real scenarios often involve more than one node. One reason is that by XQuery Update semantics, when node n is inserted in document d, all descendants of n become d nodes, thus adding n naturally leads to adding all its subtree. The same holds for deletions, i.e., removing n' from d automatically removes all the descendants of n' from d. Another reason is that updates can be performed within for-where XQuery expressions, again applying many node-level updates through a single statement. Repeatedly applying node-level update propagation procedures may become inefficient. Thus, we focus on *statement-level updates*, and study how to propagate in one step all the changes entailed by a given XQuery update statement to the affected view. This problem was studied in [19] which proposes an XQuery algebra-based approach for maintaining XQuery views. However, that approach is defined in the Galax algebra and is thus quite tied to the internals of that system. More information on related works is provided in Section 6.

In our work, we address the incremental maintenance of XML views in the presence of statement-level XML updates. Our view

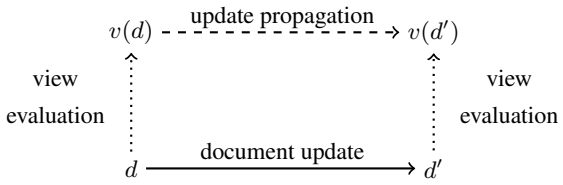


Figure 1: View update propagation.

language corresponds to a core useful conjunctive XQuery subset. This language supports the child and descendant axis, value and branch predicates, and moreover allows returning data from more than one node, unlike the XPath dialects studied in [13, 30, 31]. Our approach is designed to take advantage of advanced artifacts of current XML query processors, such as structural joins and smart identifiers [37]. Employing such efficient tools allows our algorithms to outperform node-level update propagation techniques in the frequent case where more than one node is added/removed at the same time. Moreover, our approach integrates smoothly in the process of updating the source document itself, by re-using some partial results of the update process. These features make it a good candidate to be integrated within a persistent XML database.

This paper is organized as follows. Section 2 presents our model for documents, views, and updates. Section 3 provides algorithms for propagating insertions, whereas Section 4 studies deletions. We study the performance of our algorithms in Section 5, compare our work in more detail with the state of the art in Section 6 and then conclude.

2. PRELIMINARIES

In this Section, we present our model for documents, node identifiers, views, and updates.

2.1 XML documents and node identifiers

We view XML documents as ordered label trees, consisting of element, attribute and text nodes. Element nodes and attribute nodes have a label, text nodes have an associated string representing the value. Each node has a unique identifier (or ID, in short), which is given by a compact unique string in the corresponding encoding scheme. Among the many node ID schemes from the literature, we use the recently proposed compact dynamic Dewey IDs [37] since they have many properties useful in our context:

- they are structural, i.e., by comparing two nodes, it is possible to know whether one is a parent (or ancestor) of the other;
- from the ID of a node, one may extract the IDs and labels of its ancestors at all levels;
- they do not require node relabeling in the presence of updates to the document;
- they can be encoded in a very compact fashion.

2.2 Views

Let $\mathcal{L}$ be a finite set of XML node names, and $\mathcal{XP}$ be the XPath$^{\{/,//,[]\}}$ language. We consider views expressed in the XQuery dialect described in Figure 2. In the for clause, $absVar$ corresponds to an absolute variable declaration, which binds a variable named x_i to a path expression $p \in \mathcal{XP}$ to be evaluated starting from the root of some document available at the URI uri. The non-terminal $relVar$ allows binding a variable named x_i to a path expression

1	q :=	for $absVar$ (, $(relVar)$)* (where $pred$ (and $pred$)*)? return ret
2	$absVar$:=	x_i in doc(uri) p
3	$relVar$:=	x_i in x_j p // x_j *introduced before* x_i
4	$pred$:=	string(x_i) = c
5	ret :=	$\langle l \rangle$ $elem$* $\langle /l \rangle$
6	$elem$:=	$\langle l_i \rangle \{$ (x_k | id(x_k) | string(x_k)) $\} \langle /l_i \rangle$

```
for     $p in doc("confs")//confs//paper, $a in $p/affiliation
return ⟨result⟩ ⟨pid⟩{id($p)}⟨/pid⟩ ⟨aid⟩{id($a)}⟨/aid⟩
               ⟨acont⟩{$a}⟨/acont⟩ ⟨/result⟩
```

Figure 2: Grammar for XML materialized views (top) and sample view (bottom).

$p \in \mathcal{XP}$ to be evaluated starting from the bindings of a previously-introduced variable x_j. The optional where clause is a conjunction over a number of predicates, each of which compares the string value of a variable x_i with a constant c.

The return clause builds, for each tuple of bindings of the for variables, a new element labeled l, having some children labeled l_i ($l, l_i \in \mathcal{L}$). Within each such child, we allow one out of three possible information items related to the current binding of a variable x_k, declared in the for clause: (1) x_k denotes the full subtree rooted at the binding of x_k; (2) string(x_k) is the string value of the binding; (3) id(x_k) denotes the ID of the node to which x_k is bound.

There are important differences between the *subtree* rooted at an element (or, equivalently, its *content*), its *string value* and its *ID*. The content of x_i includes all (element, attribute, or text) descendants of x_i, whereas the string value is only a concatenation of n's text descendants [34]. Therefore, string(x_i) is very likely smaller than x_i's content, but it holds less information. Second, an XML ID does not encapsulate the content of the corresponding node. However, XML IDs enable joins which may stitch together tree patterns into larger ones. Our view dialect distinguishes IDs, value and contents, and allows any subset of the three to be returned for any of the variables, resulting in significant flexibility.

Tree pattern representation for views For ease of explanation, we represent views using the following tree pattern dialect, denoted $\mathcal{P}$.

1. Pattern nodes can carry the label of an XML element or attribute, or some word, respectively. A word is defined as a sequence of characters appearing either in a PCDATA node or in an attribute value, delimited by the usual separators (whitespace, tab, end of line). Each internal pattern node carries a label from a tag alphabet $A_l = \{a, b, c, \ldots\}$. Each leaf node carries a label from a word alphabet $A_w = \{\underline{a}, \underline{b}, \underline{c} \ldots\}$.
2. Pattern edges correspond to parent-child or ancestor-descendant relationships between nodes.
3. Each pattern node may be annotated with *stored attributes*, describing additional information items that the pattern stores out of each XML document node, that matches the pattern node. The $cont$ annotation indicates that the full (serialized) image matching the XML tree node is stored. The ID annotation indicates the Compact Dynamic Dewey ID [37] of the corresponding nodes. Storing IDs in views enables combining several views in order to answer a query [27, 33]. Finally, the val annotation stands for the node's text value, obtained by concatenating all its text descendants in document order.
4. Each node may be annotated with a predicate of the form $[val = \underline{c}]$ where $\underline{c} \in A_w$, restricting the XML nodes which match the pattern node, to those satisfying the predicate.

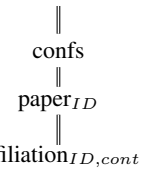


Figure 3: Sample tree pattern.

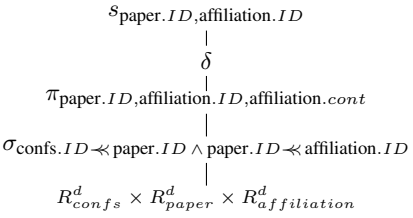


Figure 4: Algebraic semantics of the tree pattern in Figure 3.

The translation of an XQuery view into an equivalent tree pattern is described (for a superset of the language considered here) in [5].

Algebraic tree pattern semantics View semantics can be defined in the customary way based on tree embeddings [3]. For our purposes, we will rely on an equivalent semantics, introduced by means of an algebra. We present it here briefly, and point the reader to [4] for the detailed tree pattern semantics.

Given a document d and label $a \in \mathcal{L}$, we denote by R^d_a and call it *virtual canonical relation of a in d*, the list of tuples of the form $(ID, val, cont)$ obtained from all the a-labeled nodes in d. The tuples in R^d_a are sorted in the order of appearance of the corresponding nodes in d. We denote by $\prec$ the *parent comparison operator*, which returns true if its left-hand argument is the ID of a parent of the node whose ID is the right-hand argument. Similarly, $\prec\!\!\prec$ is the *ancestor comparison operator*. Observe that we only discuss a logical algebra here, and make no assumptions on how $\prec$ and $\prec\!\!\prec$ are actually implemented in a physical store.

Let $\mathcal{A}$ be the algebra consisting of the following operators: (1) the n-ary cartesian product $\times$; (2) selection, denoted σ_{pred}, where $pred$ is a conjunction of predicates of the form $a \odot \underline{c}$ or $a \odot b$, a and b are attribute names, $\underline{c}$ is some constant, and $\odot$ is a binary operator among $\{=, \prec, \prec\!\!\prec\}$; (3) projection denoted π_{cols}; (4) duplicate elimination (denoted δ); (5) sort, denoted s_{cols}. For convenience, we also use joins, defined, as usual, as selections over $\times$.

The algebraic semantics of the tree pattern in Figure 3 is the algebraic expression in Figure 4. Reading from the bottom up, there is an R_a atom per query node labeled a, and they are connected through $\times$ operators. The selection σ enforces (i) all value constraints on the nodes, and (ii) all structural $\prec$ or $\prec\!\!\prec$ relationships between query nodes. The projection retains the attributes projected by query nodes, e.g. paper.ID, affiliation.ID and affiliation.$cont$, but also *the identifiers of all nodes annotated with some attribute*. After duplicate elimination (δ), we sort the tuples in the order dictated by the IDs of the bindings of all nodes.

Derivation count A final important note is needed on the view semantics. In keeping with the standard literature on view maintenance for relational and XML data [22, 31], we associate to each tuple in a view a *derivation count*, which intuitively corresponds to the number of reasons why the tuple belongs to the view. In the semantics based on embeddings, the number of derivations of a tuple t corresponds to the number of distinct embeddings of the view in a document, that lead to the same view tuple t. In our algebraic semantics, the derivation count of t is the number of tuples in the output of the δ operator which lead to obtaining t.

We end by noting that the semantics (content) of a view v (corresponding to a tree pattern) on a document d, together with the derivation numbers, can be computed in $O(|v| \times |d|)$, e.g. by an extension to the algorithm presented in [15].

2.3 Updates

We consider the following kinds of updates:

- delete q, where q is an XPath query from $\mathcal{XP}$;
- for \$x in q insert xml into \$x, where $q \in \mathcal{XP}$, and xml is a forest of XML trees. This generalizes to updates of the form for \$x in q_1 insert q_2 into \$x, where q_1 is any XQuery, and q_2 is an XPath query from $\mathcal{XP}$ as follows. First, we evaluate q_1 on the original document, and then we proceed as if we were inserting q_2 results as children. Of course q_2 may depend on \$x, in which case different forests may be inserted under different nodes returned by q_1.
- the simpler form insert xml into q with $q \in \mathcal{XP}$ as above is also supported, together with its more general variant insert q_1 into q_2 where $q_1, q_2 \in \mathcal{XP}$.

3. PROPAGATING INSERTIONS

Let v be a view and u be an insert update directive on document d. In this Section, we discuss algorithms for transforming v into v' so as to reflect the effect of u on d. Section 3.1 outlines our general approach, then Section 3.2 describes a set of techniques which reduce the view maintenance effort, and Section 3.3 outlines how schema information can be used for further pruning. Section 3.4 introduces a set of useful ingredients for our algorithms, based on state-of-the-art XML query and update processing. Finally, Section 3.5 shows how to propagate updates which lead to *adding* new tuples to a view, whereas Section 3.6 considers the case when an XML insertion only leads to *modifying* view tuples.

3.1 Approach

Our approach relies on the algebraic view semantics, as follows. Assume that the nodes in the view v are labeled with the tags $a_1, a_2, \ldots, a_k \in A_l$. Then, v can be written as:

$$v = e_v(\sigma_{a_1}(R_{a_1}) \bowtie \sigma_{a_2}(R_{a_2}) \bowtie \ldots \bowtie \sigma_{a_{k-1}}(R_{a_{k-1}}) \bowtie \sigma_{a_k}(R_{a_k}))$$

where for each a_i, σ_{a_i} is the possible selection on the value of the nodes matching the respective view node, and e_v is an algebraic expression including the remaining projections, sorting steps, and duplicate eliminations in the full algebraic semantics of v. The joins $\bowtie$ correspond to the specific structural relationship predicates connecting the a_i nodes in the view v.

We designate the nodes added by u to d as *new* nodes. For any node label l, we term Δ^+_l the ordered collection of tuples of the form $(n.ID, n.val, n.cont)$ obtained from all the nodes n added to the document by the update. The IDs of the new nodes are computed as a side-effect of the document update, whereas their values and contents can be extracted directly from the subtrees rooted at the nodes (recall that according to the XQuery update semantics, when a node n is added to d, all the subtree of n is added to d). Based on the Δ^+ relations, the impact of u on d can be expressed as follows: for each node label l occurring in v, replace R^d_l by $R^{d'}_l = R_l \cup \Delta^+_l$.

After the update, the content of the view v should thus become:

$$v' = v(d') = e_v(\sigma_{a_1}(R_{a_1} \cup \Delta^+_{a_1}) \bowtie \sigma_{a_2}(R_{a_2} \cup \Delta^+_{a_2}) \bowtie \ldots \bowtie \sigma_{a_{k-1}}(R_{a_{k-1}} \cup \Delta^+_{a_{k-1}}) \bowtie \sigma_{a_k}(R_{a_k} \cup \Delta^+_{a_k}))$$

To solve our incremental view maintenance problem, we will ignore e_v, which is the part of v's algebraic semantics that applies projections, sorts, and eliminates duplicate tuples, because maintaining e_v's output is straightforward once updates have been propagated to e_v's input. Instead, we will focus on efficiently computing the join-over-union expression which builds the input to e_v. To ease presentation, and without loss of generality, we may simply ignore e_v in the following examples. However, we underline that e_v is necessary to ensure correctness of the semantics.

Distributing the joins over unions in the expression above leads to a single union of 2^k *terms*, each of which is a join expression. One of them involves no Δ^+ relation, and corresponds to the original view v. Propagating u thus requires computing the remaining $2^k - 1$ union terms.

EXAMPLE 3.1. *Let d be a document and u_1 be an update that inserts in d the following XML snippet:*

$$xml_1 = \langle a\rangle\langle b/\rangle\langle b\rangle\langle c/\rangle\langle/b\rangle\langle/a\rangle$$

Let a_1, b_1, b_2 and c_1 be the XML elements inserted in d by u_1. The Δ^+ relations corresponding to u_1 are:

Δ^+_a	Δ^+_b	Δ^+_c
$(a_1.id, a_1.val, a_1.cont)$	$(b_1.id, b_1.val, b_1.cont)$ $(b_2.id, b_2.val, b_2.cont)$	$(c_1.id, c_1.val, c_1.cont)$

Consider the view $v_1 = //a//b_{id}//c_{id}$. After u_1 is applied, v_1 should become:

$$\begin{array}{r} v_1 \cup (R_a \bowtie_{a \prec\!\!\prec b} R_b \bowtie_{b \prec\!\!\prec c} \Delta^+_c) \cup \\ (R_a \bowtie_{a \prec\!\!\prec b} \Delta^+_b \bowtie_{b \prec\!\!\prec c} R_c) \cup (R_a \bowtie_{a \prec\!\!\prec b} \Delta^+_b \bowtie_{b \prec\!\!\prec c} \Delta^+_c) \cup \\ (\Delta^+_a \bowtie_{a \prec\!\!\prec b} R_b \bowtie_{b \prec\!\!\prec c} R_c) \cup (\Delta^+_a \bowtie_{a \prec\!\!\prec b} R_b \bowtie_{b \prec\!\!\prec c} \Delta^+_c) \cup \\ (\Delta^+_a \bowtie_{a \prec\!\!\prec b} \Delta^+_b \bowtie_{b \prec\!\!\prec c} R_c) \cup (\Delta^+_a \bowtie_{a \prec\!\!\prec b} \Delta^+_b \bowtie_{b \prec\!\!\prec c} \Delta^+_c) \end{array}$$

For brevity, in the sequel, we will omit the join predicates (which are always those of the view) from the union terms. Thus, the expected content of v_1 after the insertion in Example 3.1 can be written as:

$$v_1 \cup R_a R_b \Delta^+_c \cup R_a \Delta^+_b R_c \cup R_a \Delta^+_b \Delta^+_c \cup \Delta^+_a R_b R_c \cup \Delta^+_a R_b \Delta^+_c \cup \Delta^+_a \Delta^+_b R_c \cup \Delta^+_a \Delta^+_b \Delta^+_c$$

In the remainder of this Section, we study practical algorithms for computing the $2^k - 1$ terms whose results need to be added to v in order to make it reflect the insertion.

3.2 Term pruning

Several observations lead us to infer when some of the union terms are guaranteed to have empty results. Such union terms are *pruned*, that is, their evaluation is not necessary in order to propagate the insertion to the view. Pruning significantly reduces the update propagation effort.

Pruning by the update semantics. The semantics of XQuery Update allows deciding that some terms will always have empty results. Intuitively, this is because XQuery updates allow adding new children to existing nodes, but not new parents, as the following example illustrates.

EXAMPLE 3.2. *Consider the insertion u_1 from Example 3.1. A newly added a node cannot have as child a b node which belonged to d before u_1 was applied. Thus, $\Delta^+_a R_b$ is empty, therefore the terms $\Delta^+_a R_b R_c$ and $\Delta^+_a R_b \Delta^+_c$ to be added to v_1 in order to maintain it are guaranteed to produce an empty result. Similarly, no b element in Δ^+_b can have descendants in R_c, therefore $\Delta^+_b c$ is also empty, and so are $R_a R_b \Delta^+_c$ and $\Delta^+_a R_b \Delta^+_c$. Thus, to compute v'_1, it suffices to add to v the results of evaluating the terms:*

(*) $R_a R_b \Delta^+_c \cup R_a \Delta^+_b \Delta^+_c \cup \Delta^+_a \Delta^+_b \Delta^+_c$

This observation is generalized by the following proposition:

PROPOSITION 3.3. *Let v be a view of k nodes, and n_1, n_2 be v nodes such that n_2 is a (/ or //) child of n_1. Let R_{n_1}, respectively, R_{n_2} be the atoms corresponding to n_1, respectively, n_2 in the algebraic semantics of v, i.e., $R_{n_1} \bowtie R_{n_2}$ is a sub-expression of v.*

Let u be an arbitrary insertion, and t be one of the $2^k - 1$ terms to be added to v in order to propagate the effect of u. If t contains as a sub-expression $\Delta^+_{n_1} R_{n_2}$, then t produces an empty result.

Observe that Proposition 3.3 *does not depend on the insertion u*. Therefore, in the sequel, when propagating updates to a view, we only focus on the terms which have survived this pruning.

Inserted data-driven pruning. Inspecting the XML fragments to be inserted in the document may allow further pruning, as illustrated by the following example:

EXAMPLE 3.4. *Consider the view v_1 from Example 3.1 and the insertion u_2 which adds the following XML snippet:*

$$xml_2 = \langle a\rangle\langle b/\rangle\langle b/\rangle\langle/a\rangle$$

The difference with respect to Example 3.1 is that xml_2 does not include a c element, i.e., $\Delta^+_c = \emptyset$. This entails that all the terms of the expression (*) in Example 3.2 are empty and thus, v_1 is not affected by u_2.

Value predicates may also impact update propagation, as the following example shows:

EXAMPLE 3.5. *Consider the view $v_2 = //a_{[val=5]}//b_{id}$ and the insertion u_3 adding the following XML snippet:*

$$xml_3 = \langle a\rangle 3\langle b/\rangle\langle b/\rangle\langle/a\rangle$$

In this case, $\Delta^+_a \neq \emptyset$ and $\Delta^+_b \neq \emptyset$, however $\sigma_b(\Delta^+_b) = \emptyset$ because the new a element does not satisfy the view predicate $[val = 5]$. Thus, $R_a \Delta^+_b$ and $\Delta^+_a \Delta^+_b$, which both involve $\sigma_b(\Delta^+_b)$, are empty. Since Proposition 3.3 the term $\Delta^+_a R_b$ is also empty, v_2 is unaffected by u_3.

This generalizes to the following simple observation:

PROPOSITION 3.6. *Let u be an insertion adding the trees $t_1, t_2, \ldots, t_k$ to d, and v be a view. If a node n of v is not matched in any of the trees $t_1, t_2, \ldots, t_k$, all union terms involving Δ^+_n are empty.*

Inserted ID-driven pruning. A third pruning criteria takes advantage of an interesting property of Compact Dynamic Dewey IDs [37]:

EXAMPLE 3.7. *Consider the view v_1 from Example 3.1 and the insertion u_4, adding the following XML snippet:*

$$xml_4 = \langle b\rangle\langle c/\rangle\langle/b\rangle$$

as a child of a node a, whose ID is $a.id$. This ID encodes the labels of all the nodes on the path from a to the root [37]. Assume that we inspect $a.id$ and find that no ancestor labeled b appears above the a node. Then, the new (inserted) c node has only one b ancestor, namely the inserted (new) b node. Thus, the term $R_a R_b \Delta_c^+$ in (*) is empty.

In this example, moreover, since $\Delta_a^+ = \emptyset$, the term $\Delta_a^+ \Delta_b^+ \Delta_c^+$ in (*) is also empty. Thus, the only term we need to compute to update v_1 after inserting xml_4 is: $R_a \Delta_b^+ \Delta_c^+$.

This generalizes as follows:

PROPOSITION 3.8. *Let u be an insertion adding children to the nodes $p_1, p_2, \ldots, p_k$ in d. Let v be a view, and n_1, n_2 be v nodes such that n_1 is an ancestor of n_2 in v.*

If for each $i = 1, 2, \ldots, k$, p_i is not labeled n_1 and has no ancestor labeled n_1, then all union terms containing $R_{n_1} \Delta_{n_2}^+$ are empty.

3.3 Exploiting schema information

A schema for XML documents might not always be available, nor is the programmer expected to write update statements that are valid with respect to a schema. When document type information is available, we can use it in our approach to either reject some schema-violating insertions, or optimize their propagation. The first issue (deciding view-update independence) is not our main focus and has been thoroughly considered elsewhere [7, 10]. We present in this section the kind of *run-time* independence decisions that our framework captures. Our techniques rely on the inserted data, thus they may require partially evaluating the update (as opposed to purely static techniques using only the update statement and the view). Being inspired by real update scenarios, we thought that it could have been useful to make these decisions at run-time and actually let the user choose to proceed or not with update propagation, in the presence of schema violations. In practice, static and run-time checks are complementary and can be combined to reduce view maintenance effort whenever possible. Dealing with such a combination is beyond the scope of our work.

Deciding view-update independence We consider that documents are characterized by DTDs expressed as extended context-free grammars (CFGs), where the right-hand side of each rule is a regular expression over an alphabet of terminal and non-terminal symbols. For instance, Figure 5 depicts two DTDs. In this Figure, a, b, c, d1, d2 and x are terminal symbols and AS and BS are non-terminal ones. DTD d1 (a) has mandatory edges, while DTD d2 (b) features concatenation, disjunction and recursion.

EXAMPLE 3.9. *Consider the view v_1 from Example 3.1 and an insertion u_5, adding the following XML snippet:*

$$xml_5 = \langle a\rangle\langle b\rangle\langle/b\rangle\langle/a\rangle$$

Applying the update would make the document invalid with respect to the DTD in Figure 5(a), since a c element is missing under b. More generally, from the DTD d1, one can derive that the following statement must hold for any newly inserted XML tree:

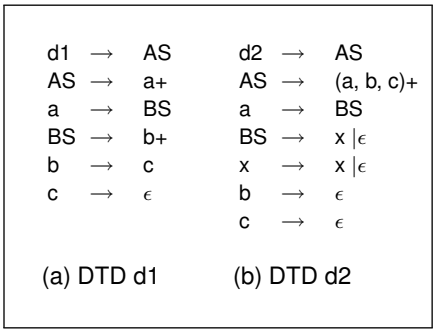


Figure 5: Sample DTDs, expressed as CFGs.

$$\Delta_c^+ = \emptyset \Rightarrow \Delta_b^+ = \emptyset$$

Since this does not hold on the update u_5, we reject it due to its attempted schema violation.

The same consideration applies to Figure 5(b), in which a $d2$ element must have as children the concatenation of a, b and c. Therefore, any insertion of an a element under the root $d2$ must occur with b and c elements.

EXAMPLE 3.10. *The DTD in Figure 5(b) implies that the following statement must hold on any XML forest inserted under a given node:*

$$\Delta_a^+ \neq \emptyset \Rightarrow (\Delta_b^+ \neq \emptyset \wedge \Delta_c^+ \neq \emptyset)$$

More generally, from the DTD rules, one can infer a set of constraints on the Δ^+ tables, and check them before applying the update. If any constraint is violated, the update is rejected.

3.4 Helper functions and operators

Let u be an update (insertion or deletion), and $pul(u)$ be the **pending update list** [36] resulting from u. Thus:

- if u is an insertion, $pul(u) = \{(n_1, t_1), (n_2, t_2), \ldots, (n_k, t_k)\}$, a list of pairs consisting of an XML element n_i *target of the update*, and a subtree t_i to be copied as a child of n_i.
- if u is a deletion, $pul(u) = \{n_1, n_2, \ldots, n_k\}$ is the list of the nodes to be removed.

We assume available:

compute-pul(u) A procedure which from an update u, computes its pending update list $pul(u)$. Any XQuery Update-compliant store has (some version of) this procedure.

apply-insert(n,t) is a function which, given a node n and a tree t to be copied as a child of n copies t as a new child of n and returns the tree t' created by the copy operation. Importantly for us, *the tree t' also includes the IDs assigned to the copied t nodes in their new context (in d).*

extr-pattern(p, t) is a function which, given a tree pattern p and an XML tree t, evaluates p on t and returns the corresponding set of tuples. An $O(|p| \times |t|)$ implementation of **extr-pattern** is obtained by extending the algorithm presented in [15].

operators We also assume available the following logical (and physical) operators:

Structural Joins comprise the classical Stack Tree Ancestor and Stack Tree Descendant algorithms [2];

Algorithm 1: Propagate Insert by New Tuples **(PINT)**

Input : view v, insert update u
Output: updated view v' to reflect u

1 Develop the $2^k - 1$ union terms to be added to v in case of insertions, and prune them based on XQuery update semantics (Proposition 3.3)
2 Compute the Δ^+ tables corresponding to u
3 Further prune terms based on the Δ^+ tables (Propositions 3.6 and 3.8)
4 Evaluate the remaining terms and add their results to v
5 If needed, update auxiliary structures

Path Filter allows checking whether a node having a specific ID is on a path satisfying a specific condition.

Path Navigate allows to obtain, from the ID of some nodes, the IDs of their parents.

The PathFilter and PathNavigate are implemented by exploiting the information encapsulated by the expressive node IDs we use [37].

3.5 Propagating insertions by adding tuples to the view

Our first update propagation algorithm considers the cases when an insertion to the XML document either leads to new tuples being added to the view, or does not affect the view. All the examples considered so far fall into this situation. The cases left out are those when the insert leads to *modifying* a view tuple. This case will be addressed in Section 3.6.

Algorithm 1 outlines the propagation procedure. Line 1 is performed when v is created, since it is independent of the update. The computation of the Δ^+ tables will be detailed shortly. The core of the complexity in Algorithm 1 lies in line 4: the computation of the remaining union terms.

Term evaluation based on auxiliary lattice. Let us now see how to compute the terms that survive pruning. Each such term is sure to involve at least a Δ^+ table, and all but (possibly) one also involve an R_l table, for some view node label l. To maintain the view v, we rely on a collection of auxiliary data structures which are materialized and also maintained when the data changes. These data structures are organized in a *view lattice*, which is best introduced by means of examples.

Figures 6 and 7 depict the lattices corresponding to the tree patterns shown at the top left of each Figure. For conciseness, in both Figures, the pattern of each node is shown simply by the set of its node labels. Thus, the pattern of the topmost node in Figure 7 is $//a_{ID}//b_{ID}//c_{ID}//d_{ID}$, its rightmost child corresponds to the pattern $//b_{ID}//c_{ID}//d_{ID}$ etc. The lattice top node is always the complete pattern, and each pattern node corresponds to one lattice leaf. The view lattice recalls the AND-OR graphs well-known in relational data warehousing [23], adapted to XML tree patterns with multiple return nodes.

Formally, let v be a tree pattern. The lattice of v is a DAG with three categories of nodes: (i) A set of nodes, each labeled by some $\mathcal{P}$ pattern. One node is labeled by the pattern v_{ID}, obtained from v by annotating all nodes exactly with ID. Every other node is labeled by a distinct sub-pattern (itself a tree pattern) of v_{ID}. (ii) A set of or-nodes labeled $\vee$; (iii) A set of join nodes labeled $\bowtie$.

Lattice edges trace possible ways of computing some lattice nodes based on nodes below it. For instance, a join ($\bowtie$) allows computing the node labeled ab, corresponding to $//a_{ID}//b_{ID}$, out of the nodes labeled a and b respectively. When the sub-pattern corresponding to a lattice node can be computed in several ways out of the lower nodes, this is modeled by the or ($\vee$) node which alone points to the target lattice node. In Figure 6, the sub-pattern abc can be computed in three distinct ways, whereas in Figure 7 there are only two possibilities [1]

A view lattice captures all view sub-patterns, and possible ways of computing them from one another. Materializing (and maintaining) all these sub-patterns suffices to update v to reflect any insertion. For each union term t to be added to v after an insertion:

- Let t_R be the $\bowtie$ sub-expression(s) of t containing only R_l occurrences. Then, t_R corresponds exactly to some materialized lattice node(s).
- Let t_{Δ^+} be the $\bowtie$ sub-expression(s) of t containing only Δ^+ occurrences. Then, t_{Δ^+} is easily computed based on the new inserted data.

The lattice allows solving the problem of maintaining v based on the "smaller" problems of maintaining sub-patterns of v. Given that the maintenance of the lattice leaves is trivial (e.g., replace R_l by $R_l \cup \Delta_l^+$), in theory, this is a solution. However, materializing and maintaining all lattice nodes is likely to be very expensive in terms of space and time. Fortunately, we can focus only on a subset of these nodes:

DEFINITION 3.11 (SNOW CAP). *Let v be a tree pattern. We term* snow cap of v*, any non-empty subtree u of v such that: for each node $n \in v$ that also appears in u, the parent of n in v also appears in u.*

For instance, in Figure 6 and Figure 7, the boxed nodes depict snow caps. Intuitively, a snow cap copies the root of the pattern and then goes down to some length on all paths, *only including a node n if it includes its parent*. This mimics the way mountains are covered by snow, thus the name. We can now state:

PROPOSITION 3.12. *Let v be a view, u an insertion and t a resulting term which survives our first pruning (Proposition 3.3). Then, the algebraic expression t_R is exactly the algebraic semantics of a pattern v_{t_R} which is a snow cap in v's lattice.*

The proof follows directly from Proposition 3.3. For instance, consider the view v_1 in Figure 6. For an insertion u to add tuples to v_1, one or several of the following cases must hold: (i) u adds a d child to an element on the path $//a//b//c$. The impact of this addition on v is obtained by joining the snowcap abc with Δ_d^+; (ii) u adds a c child to an element matching $//a//b$, and this c child has at least one d descendant (join the snowcap ab with $\Delta_c^+\Delta_d^+$); (iii) u adds a b child to an element matching $//a$, and this b child has at least one $//c//d$ descendant (join the snowcap a with $\Delta_b^+\Delta_c^+\Delta_d^+$); or (iv) u adds matches to the full $//a//b//c//d$ view path, and to reflect such additions, no auxiliary structure (lattice node) is needed. A similar analysis of the possible ways in which an insertion could add tuples to v_2 in Figure 7 leads to the conclusion that the snow cap nodes in that lattice are necessary and sufficient to maintain v_2.

PROPOSITION 3.13. *Materializing all and only the snow caps of a view v suffices to maintain v. Moreover, each snow cap can be maintained based only on other snow caps and the Δ^+ relations extracted during each update.*

[1] We have depicted only the possibilities of computing a pattern by joining mutually disjoint sub-patterns. Clearly, there are more possibilities, e.g., one can compute abc by joining ab with bc. We omit this to keep the Figures readable.

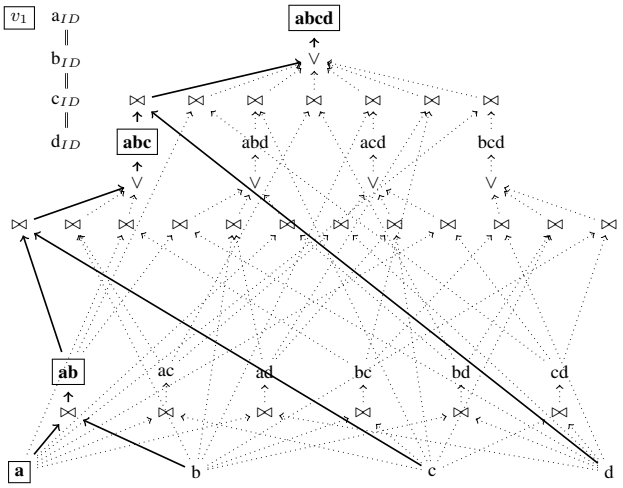


Figure 6: Sub-pattern lattice and snow caps for the view v_1.

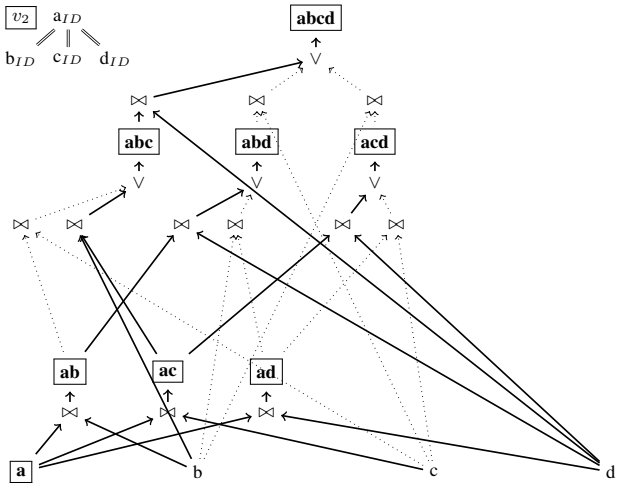


Figure 7: Sub-pattern lattice and snow caps for the view v_2.

For example, consider the view v_2 in Figure 7 and an insertion adding some new c elements to the document. The relation Δ_c^+ holds the IDs assigned to the newly added cs in the respective places where they have been inserted. Let $c_1.ID$ be such an ID. From $c_1.ID$, we can extract the IDs of all its a ancestors (if any); assume for our example there are two such ancestor IDs, $a_1^1.ID$ and $a_1^2.ID$. The tuples ($a_1^1.ID$, $c_1.ID$) and ($a_1^2.ID$, $c_1.ID$) must be added to the snow cap ac, then joined with the existing snow caps ab and ad on $a.ID$. This, in turn, may lead to new tuples being added to the snow caps abc and acd. If this is the case, a final join of these two determines which tuples, if any, should be added to v_2 to reflect the update.

Alternative to snow caps: lattice leaves An alternative to materializing and maintaining the snow caps consists of materializing just the leaf nodes, i.e., the collections of IDs for all the labels appearing in the view. (In fact, many combinations are possible, e.g., materializing a combination of snow caps and leaves, or other lattice node sets etc.) Clearly, the space occupancy and view update costs heavily depend on the data, view and update mix. We have experimented with both the set of snow caps and the leaf set, and report on this later on.

Algorithm 2: Evaluate terms resulting from insert (**ET-INS**)

Input : update u, view v, materialized snow caps
Output: updated view v to reflect u; updated materialized snow caps

1 Evaluate Δ^+ tables (call Algorithm **CD+**(u, v))
2 $\Delta_v^+ \leftarrow \emptyset$ (tuples to be possibly added to v)
3 **foreach** *term t surviving pruning* **do**
4 Evaluate t_{Δ^+} by structural joins over the Δ^+ tables
5 Add to Δ_v^+ the result of joining t_R (snow cap materialized in the lattice) and t_{Δ^+}
6 **foreach** *tuple* $t_\Delta \in \Delta_v^+$ **do**
7 **if** $t_\Delta \in v$ **then**
8 increase the derivation count of t_Δ
9 **else**
10 add t_Δ to v with a derivation count of 1
11 Update the snow caps from the bottom up in the lattice

Algorithm 3: Compute Δ^+ tables (**CD+**)

Input : update u, view v
Output: Δ^+ tables

1 $(n_1, t_1), \ldots, (n_k, t_k) \leftarrow$ **compute-pul**(u)
2 **for** $n \in v$ *labeled* l **do**
3 let p_l be the pattern $//l_{ID,val,cont}$
4 $\Delta_l^+ \leftarrow \bigcup_{1 \le i \le k}$(**extr-pattern**($p_l$, t_i))

Putting it all together. Algorithm 2 (**ET-INS**) outlines the evaluation of non-pruned terms resulting from insertions. At lines 4 and 5, Algorithm **ET-INS** relies on structural joins, in order to take advantage of efficient optimization and evaluation techniques provided by the persistent XML store and query engine.

Computing Δ^+ relations. Given an insertion u on the document d and the view v, Algorithm 3 computes the Δ^+ relations. It relies on the functions **compute-pul** and **extr-pattern** presented in Section 3.4 to compute the update list, and then to extract Δ^+ relations out of the pending updates.

ID-based optimization We now exploit the special properties of Compact Dynamic Dewey IDs to make update propagation even more efficient. Let n_1 be a view node labeled a and n_2 be a descendant of n_1 in v, labeled b. Consider an update u such that $\Delta_b^+ \neq \emptyset$. Then, in the union terms involving R_a and Δ_b^+, we can replace R_a by the set of a-labeled identifiers of the newly added b nodes. We can do this because the XML ID scheme we use [37] encapsulates in the ID of a node, the IDs and labels of all its ancestors. The IDs of all the a ancestors of the new b nodes appear in the new b nodes' IDs.

3.6 View tuple modification

In some cases, an insertion on a document d may lead to *modifying* existing tuples of a view v. This occurs when the insertion changes the value or the content of an XML node n, whose value (respectively, content) is stored in v. A view tuple storing the value and/or content of n may also need an update if a descendant of n is added or modified by the update.

EXAMPLE 3.14. *Consider the view $/a_{ID}/b_{ID}//c_{ID,Cont}$ and an insertion u adding the XML snippet:*

$\langle extra \rangle$ *some value* $\langle /extra \rangle$

into $//d//c$*. In this case, no* Δ^+ *relation affects the view, thus no new tuples need to be added. However, the insertion* u *may lead to modifying some of the* $c.cont$ *values stored by the view, if the intersection* $/a/b//c$ *and* $//d//c$ *is not empty.*

In the following, we present an algorithm that addresses this case. The algorithm considers all XML nodes for which the view stores content or value, verifies whether that node is affected by the update, and if this is the case, updates its value and/or content.

Algorithm 4: Propagate Insert by Modifying Tuples **(PIMT)**

Input : insert update u, view v
Output: updated view v' to reflect u

```
1  ut = [(n_1, t_1), ..., (n_k, t_k)] ← compute-pul(u);
2  cvn ← {n ∈ v, n annotated with cont or val};
3  foreach tuple t ∈ v do
4      foreach tuple (n_i, t_i) ∈ ut do
5          foreach node n ∈ cvn do
6              if t.n = n_i or t.n ≪ n_i then
7                  Update t.n.cont (respectively, t.n.val) to
                   reflect the insertion of t_i
```

Algorithm 4 (**PIMT**) starts by computing the pending update lists. It singles out all the content- or value-annotated view nodes, in the node set cvn. The algorithm then checks, for each view tuple t, whether and how each of the t attributes corresponding to the content or value of a cvn node must change. To that purpose, Algorithm 4 requires that for all cvn nodes, i.e., for all those view nodes for which $cont$ or val is stored, element IDs must be also stored. Based on the IDs, we check whether the node providing the $cont$ attribute in tuple t is the same as, or an ancestor of the modified node n_i. If this is the case, then the insertion of t_i has to be propagated to the t attribute corresponding to $n.cont$ (respectively, $n.val$).

It is easy to see that if cvn is empty, insertions cannot modify view tuples (but only add to the view).

If cvn is of size 1 (a single view node stores val or $cont$), then Algorithm **PIMT** can be implemented by a single efficient structural join (extended to check ancestor-descendant *or* equality relationships) between v and the pending update list. In the view tuples that join with the pending update list, the $cont$ and/or val attributes must be changed.

If cvn contains more nodes, Algorithm 4 must compare several ID attributes from each view tuple t against the pending update list, and a nested loops join is needed.

We conclude by observing that in practice one cannot know in advance whether an insertion will add or modify tuples, therefore both Algorithms **PINT** and **PIMT** are run, based on the pending update list which is computed only once.

4. PROPAGATING DELETIONS

Another possible propagation scenario that we need to consider concerns the case of deletions on a node. Similarly to insertions, deletions may lead to delete an entire view tuple or to modify an existing view tuple. We start by considering the first kind of propagation, i.e., the deletion of tuples from the view.

The general Algorithm that detects and propagates the deletions to the view is similar to the one we have presented for insertions in Section 3. Union terms need to be built and possibly pruned (as shown in Algorithm 5), then evaluated, and this may result into tuples being deleted from the view. We start by presenting some examples.

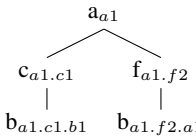


Figure 8: Sample XML document.

Algorithm 5: Propagate Delete by Deleting Tuples **(PDDT)**

Input : view v, delete update u
Output: updated view v' to reflect u

```
1  Develop the 2^k − 1 union terms to be deleted to v in case of
   deletions, and prune them based on XML constraints (anal. to
   Proposition 3.3)
2  Compute the Δ⁻ tables corresponding to u
3  Further prune terms based on the Δ⁻ tables (anal. to
   Propositions 3.6 and 3.8)
4  Δ⁻_v ← ∅ (tuples to be possibly deleted from v)
5  foreach remaining term t do
6      evaluate t (use materialized snow caps, Δ⁻ tables,
       structural joins etc.) and add its results to Δ⁻_v
7  foreach tuple t_Δ ∈ Δ⁻_v do
8      decrease t_Δ derivation count
9      if t_Δ derivation count becomes 0 then
10         remove t_Δ from v
11 If needed, update auxiliary structures
```

EXAMPLE 4.1. *Consider the view* $//a_{ID}//b_{ID}$ *and the XML document* d *shown in Figure 8, where the ID of each node is shown as a subscript to the node. Each ID is a sequence of steps, each step holding the label and the relative position of one ancestor of the node*[2].

Consider an update u_1 *deleting* $//c//b$*, which results in the node whose ID is a1.c1.b1. Since the view tuple* $(a1, a1.c1.b1)$ *had a derivation count of 1, the update leads to removing the tuple from the view.*

More generally, given a (subtree) deletion and a label l, we term Δ_l^- the ordered collection of tuples of the form $(n.id, n.val, n.cont)$ obtained from all deleted nodes n labeled l. From the algebraic semantics of the view and the Δ^- relations, we develop a union of terms much as in Section 3.1 and prune them in similar fashion to Section 3.2.

In some cases, an XML deletion only affects the derivation count of a view tuple, as illustrated below.

EXAMPLE 4.2. *Consider the view* $//a_{ID}[.//b]$ *and the document* d *in Figure 8, and an update deleting* $//c//b$*.*
The view contains a single tuple corresponding to node a. The tuple has a derivation count of 2 due to the two b nodes which satisfy the predicate in the view. Therefore, the deletion does not affect the view, since by deleting only the b node identified by a1.c1.b1, the a element still has a b descendant. The only effect of the update is to decrease the derivation count by 1.

A subsequent update deleting $//f//b$ will remove the last b of the document and thus remove the tuple from the view. The general

[2]Internally, of course, ID representation is much more compact.

algorithm for propagating deletions by deleting tuples (**PDDT**) is outlined in Algorithm 5, where Propositions analogous to 3.3, 3.6 and 3.8 (omitted for space reasons) are exploited in the case of deletions.

A second case occurs when a deletion does not remove tuples from a view, but modifies some value or content attributes. The situation is again very similar to the case when XML insertions modify existing tuples. Accordingly, we have devised an algorithm **PDMT** (Propagate Delete by Modifying Tuples) which is symmetric to Algorithm 4 and omitted for brevity.

5. EXPERIMENTAL EVALUATION

In this Section, we present a set of experiments on our proposed algorithms. Section 5.1 describes the experimental setting. Section 5.2 studies the performance of our algorithms breaking down detailed running times. Section 5.3 is concerned with scalability when document size varies, Section 5.4 compares our approach with fully recomputing the view (from scratch), finally Section 5.5 compares our approach with the closest competitor [31].

5.1 Settings

We have implemented the PINT and PIMT algorithm described in this paper using Java 6, within our ViP2P Java-based platform (http://vip2p.saclay.inria.fr). ViP2P enables distributed peers to publish XML documents, and to declare materialized views which are filled with results from all network documents, in a symmetric, decentralized publish-subscribe fashion (for our purposes, we only used one ViP2P peer). ViP2P stores view data using BerkeleyDB v4.0.71. For some operations (see below), we rely on the widely known Saxon XQuery processor v9.2.1.1j. All experiments but those described in Section 5.5 were run on a PC with Linux Kubunto v2.6, with a Pentium 4 260GHz CPU and 1GB memory.

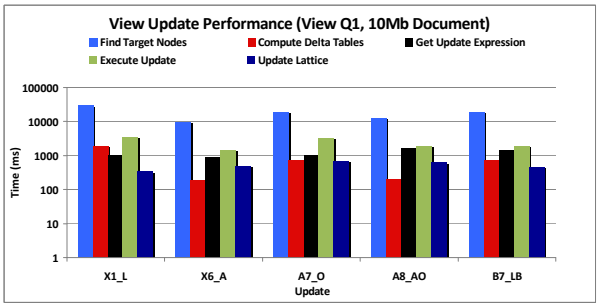
Documents, views and queries We use XMark [32] benchmark documents of different sizes. As in [11], we use queries from the (read-only) XMark benchmark as views, and a set of updates derived from the XPathMark benchmark [20] by inserting a dummy element into each of (or deleting, resp.) the nodes returned by the respective XPathMark query. To the extent possible, we used the same queries and updates as in [11], which detected the independence of the updates and the view in most of the cases. Since we are interested in the "other" cases, when updates do affect views, we enhanced the set of updates with more path expressions from the A and B subsets of the XPathMark benchmarks (the names of these queries start with the respective letter). Finally, we also added a set of path expressions of our invention, whose names start with X.

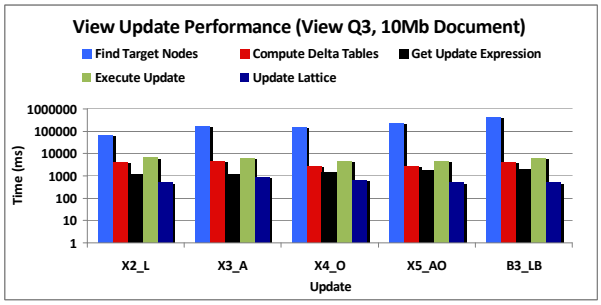
For space reasons, we report only on the results obtained with XML insertions, i.e. running algorithms **PINT** (Section 3.5) and **PIMT** (Section 3.6).

Implementation details We use Saxon for the first step of our approach, namely identifying the target update nodes that will receive new children. To actually update the document, we build the pending update list, add the new children to the target nodes using Saxon's in-memory operations, and then serialize the modified document again using Saxon.

To update the view, we extract the Δ^+ tables from the pending update list and apply all the respective steps described in Section 3. We store the necessary auxiliary structures (the snow cap lattice nodes described in Section 3.5) as ViP2P views.

All algebraic operations (notably joins) are performed using ViP2P's physical operator library. We stress that ViP2P is a Java-based prototype and improvements by constant factors could probably be obtained by further optimizing the code, using C++ etc. Nevertheless, we implemented all algorithms in the same framework, and relied on state-of-the art algorithms, e.g., for structural joins. Thus, we believe our experiments accurately highlight the various performance trade-offs involved.





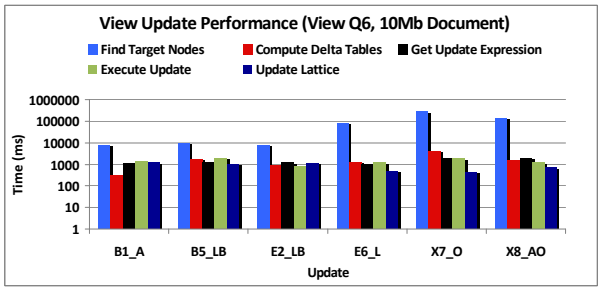


Figure 9: Time breakdown for update propagation to XMark views Q_1 (top), Q_3 (middle), and Q_6 (bottom).

Measured times In the following, we report on a set of times which were averaged over five executions. *Find Target Nodes* is the time taken by Saxon to identify the nodes involved by an update operation. *Compute Delta Tables* is the time taken to build the Δ^+ tables starting from the inserted XML fragments. *Get Update Expression* is the time to build, unfold, and prune the algebraic expression corresponding to the updates to be propagated. *Execute Update* is the time to evaluate the expression thus obtained within ViP2P's algebraic XML query evalution engine. *Update Lattice* is the time to update the auxiliary data structures (snow cap lattice terms) by computing the necessary tuples and adding them into BerkeleyDB.

5.2 Performance of the incremental maintenance algorithms

To gauge the effectiveness of our algorithms, we have run a set of experiments by using as views the queries Q_1, Q_2, Q_3, Q_4, Q_6, Q_8, Q_{13} and Q_{17} of the XMark [32] benchmark. For each query,

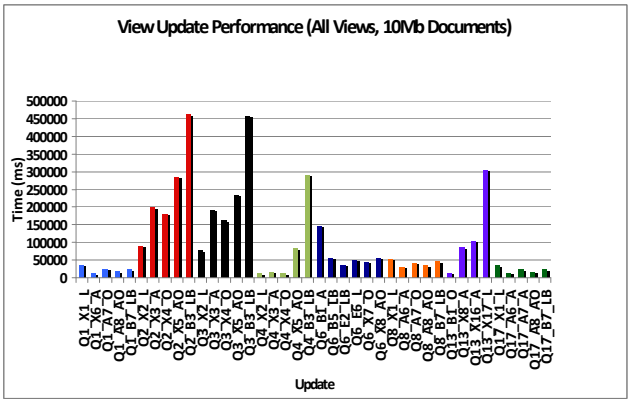


Figure 10: Performance of the incremental view maintenance algorithms for all the XMark views.

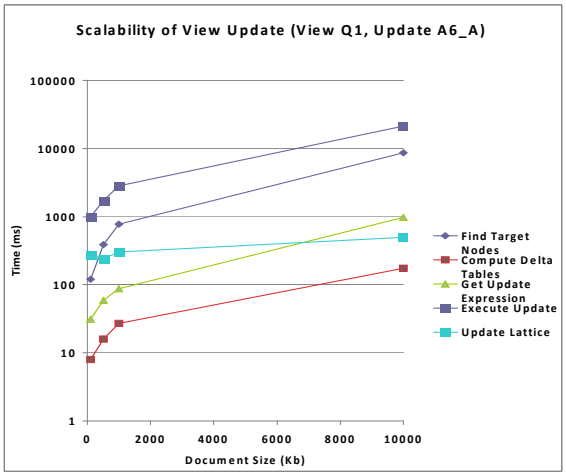


Figure 11: Scalability for XMark view Q_1 and update $A6\_A$.

we have employed a set of update path expressions (as described above), divided into five classes: (c1) *L*inear path expressions; (c2) path expressions with an *A*nd predicate; (c3) path expressions with an *O*r predicate; (c4) path expressions with an *AO* (and-or) predicate; (c5) *B*oolean path expressions.

Figure 9 shows for each pair (view, update) the impact of the view maintenance time. It can be observed that in all cases the times to *Compute Delta Tables*, *Get Update Expression* and *Execute Update* are smaller than the time to locate the target nodes. As expected, the latter times depend on the target XPath expression of the updates. Thus, for instance, the evaluation of a long linear path expression is slower than for a path expression with an AND or OR filter. Moreover, the absolute value of the time to *Update Lattice* depends on the complexity of the view considered and less on the specific update applied. For views like Q_3 (FLWR expression with conditions), it almost stays steady while increasing the complexity of the update path expression from class (c1) to (c5).

Lastly, Figure 10 shows the performance results for all the XMark views considered, by summing up the *Find Target Nodes*, *Compute Delta Tables*, *Get Update Expression*, *Execute Update* and *Update Lattice* times.

5.3 Scalability wrt. Source Document Size

We then performed a scalability test for our algorithms, to check how they perform on larger source documents. We have employed various document sizes ranging from $500KB$ to $10MB$, and have observed their performance, as shown in Figure 11, where the y axis is in logarithmic scale. The cost of updating the lattice (i.e., the auxiliary structures) is the most significant component of the view update costs, while the delta table computation and the time to get the update expression are comparably small. Moreover, executing the update, i.e., computing the join expressions which determine which tuples should be added to the view, has a cost that grows reasonably with the document size, and follows the same trend of the cost of finding target nodes. This experiment shows that the cost for view maintenance is beneficial for all the document sizes up to $10MB$. Additional experiments shown below and comparing with full view recomputation will further confirm this claim.

5.4 Comparison with Full Recomputation

We have conducted an experiment to measure the gain that our incremental algorithms have over the baseline when the view is fully recomputed from the modified document. This experiment

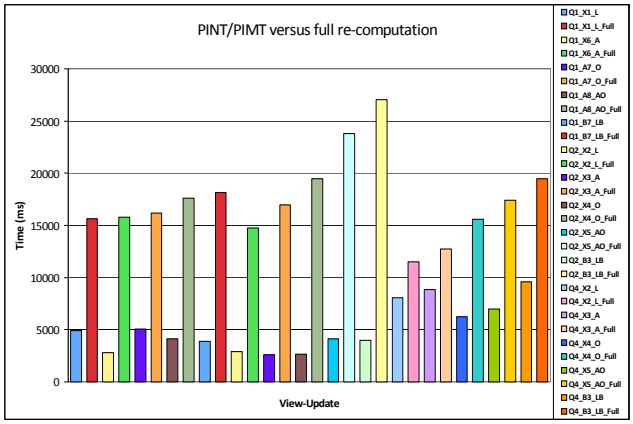


Figure 12: Incremental maintenance versus full recomputation for the XMark views Q_1, Q_2 and Q_4 and various updates.

aimed to show the time necessary to incrementally update a view, by exploiting snowcaps and pruning the term expression, versus the time necessary to compute the full unioned term expression without pruning and without relying on the lattice at all. Figure 12 shows the results for each view-update pair.

It can be noticed that the full expression recomputation becomes prohibitive for many of the scenarios, while the incremental view maintenance achieves much lower times.

5.5 Comparison with Previous Algorithm

In order to show the benefit of bulk updates for incremental view maintenance, in our framework we have re-implemented IVMA, the view maintenance algorithm described in [31]. This algorithm propagates to XPath views, updates which add or delete exactly one node at a time. For this experiment, we used insertions which add a fixed XML tree, consisting of a root node with four children. Such an insertion is handled in one shot by our algorithm, and by five consecutive calls to IVMA in [31]. These experiments have been executed on a Linux 2.6.31.13-server-1mnb, with 2.33GHz Intel(R) Xeon(R) CPU 5140 and 4GB memory. Figure 13 (in logarithmic scale) shows that our approach outperforms IVMA by (at least) one order of magnitude for the view Q_1 and a source document of 100KB.

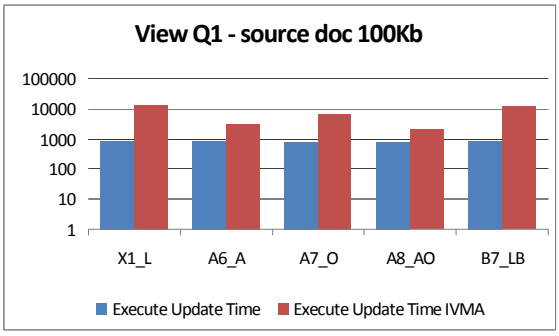


Figure 13: Comparison between our PINT/PIMT algorithms and the competitor IVMA algorithm [31] for view Q_1.

5.6 Impact of Auxiliary Structures

Finally, we have tried to quantify the trade-offs between using the snow caps, respectively the leaves of the lattice as auxiliary structures. Figure 14 shows the time to build lattice nodes, and the space occupancy, for one view Q_2.

For the various sizes of the source documents, the time to build (respectively, size of) the lattice snow cap nodes are by one order of magnitude higher than the time to build (respectively, size of) only the lattice leaves. This confirms the benefit of using the latter in the computations associated to incremental view maintenance.

6. RELATED WORK

A large body of past research has been devoted to view updates in the context of relational databases [8, 14, 21, 22]. [8, 21] focus on *the view update problem*, i.e., on how to translate a view update into a database update, while avoiding the presence of inconsistencies and side effects on the view. Recently, [14] proposed *update policies*, expressed in a bidirectional language, to guarantee that the view update is well behaved and handles arbitrary changes to the view. Optimal incremental view maintenance algorithms for relational and deductive database systems were presented in [22], where the notion of derivation count for each tuple in the view is introduced. The algorithms addressed consider both recursive and non-recursive views. In both cases, view definitions are used to generate a set of rules that compute the changes to the views using the base relations and the old views.

In the context of the XML data model, quite recently, [13] studied the problem of incremental view maintenance in its Boolean version and with respect to the XPath language. Boolean incremental view maintenance checks that, after applying the update to the base data, the XPath expression representing the view is still satisfied. Similarly to our approach, they studied the above problem and derived its complexity *per update*, i.e. by considering one update at a time. The view language they consider is slightly more limited than ours (in particular, they do not support multiple returning nodes), and their approach does not consider incorporating efficient XML query processing techniques in the view update process.

A practical fragment of XPath with $\{//, /, *, []\}$ has been used in previous work on incremental view maintenance for XPath [30, 31]. Interestingly, they also consider count() predicates, therefore view maintenance may be non-monotonic: adding some XML nodes may lead to removing data from the view, while removing XML nodes may add data to a view. In contrast, our conjunctive tree pattern dialect is monotonic. Compared to [31], we focus on (i) tree patterns with multiple return nodes, which cannot be handled by the approach of [30, 31] (based on the analysis of the XPath "view main path"). Views with multiple return nodes may lead to very efficient multiple-view rewritings [26]; and (ii) bulk updates, where several nodes can be added at the same time. As we have argued in the Introduction, the XQuery Update language gives many opportunities for such updates. Moreover, our experiments demonstrated that our algorithms, leveraging state-of-the-art techniques in XML query evaluation, outperform repeated application of the node-based algorithm of [31].

An extension of [31] is [30] which considers the case when the database and the view store are decoupled and the update has to be propagated using less information. The XPath dialect and node-at-a-time approach stay the same as in [31].

To the best of our knowledge, the only works which study the incremental view maintenance problem for XQuery views are [17, 18, 19]. [17, 18] focus on the maintenance of XQuery views over relational data. The algorithms of [19] translate updates through views expressed in the internal tree algebra of Galax. This approach is elegant due to its usage of an algebra, and handles a significantly richer XQuery subset than we do in this work. However, its tree-oriented algebra makes it differ significantly from our more traditional approach based on structural joins, delta tables, and term pruning heuristics. Moreover, it is placed at a higher level and does not consider practical aspects related to the efficient implementation of the propagation algorithms in a generic XML data management platform.

A close work in this area [1] considers the maintenance of tree pattern views (with some non-monotonic extensions) over *active documents*, that is, XML documents including calls to Web services, which return streams of answers that are inserted in the document. The solution consists of algorithms to be applied when each new answer is received and inserted in the document, a hybrid granularity between node-level (since an answer can contain several nodes) and statement level (since they do not use declarative update statements). Their solution relies on Datalog optimization techniques and on an XML database used as a black box, whereas we describe algorithms to be implemented inside the engine.

An important line of related works seeks to identify when a view is unaffected by an update [9, 11, 12]. In contrast, we provide algorithms for propagating the insertions when the view is affected.

Finally, the complexity of evaluating (XPath) tree patterns has been first established in [24].

7. CONCLUSION AND FUTURE WORK

In the current paper, we have devised algebraic incremental view maintenance algorithms, that work on a per-statement basis, as opposed to previous per-node approaches. By leveraging structural IDs and appropriate auxiliary data structures called snowcaps, our technique is efficient and scalable at the same time.

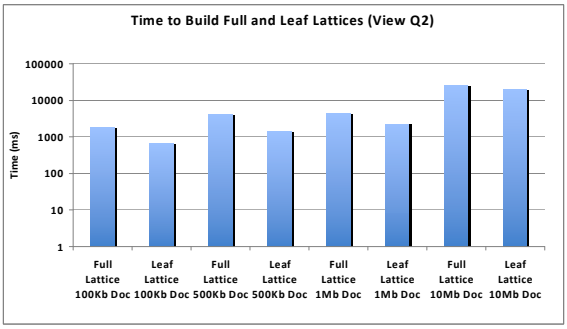
Future work is devoted to extend our approach to other XML update operations and to apply our algebraic techniques to the XML view update problem.

8. REFERENCES

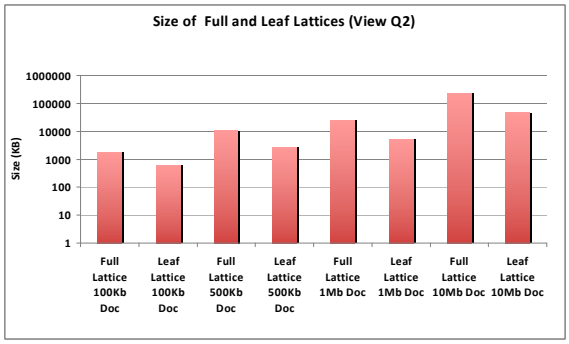
[1] S. Abiteboul, P. Bourhis, and B. Marinoiu. Efficient maintenance techniques for views over active documents. In *EDBT*, 2009.

[2] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[3] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Tree pattern query minimization. *VLDB J.*, 11(4), 2002.



(1) Lattice Build Time



(2) Lattice Size

Figure 14: Time to build (and size occupied by) the snowcaps, respectively, the lattice leaves for the XMark views Q_2.

[4] A. Arion, V. Benzaken, and I. Manolescu. XML access modules: Towards physical data independence in XML databases. In *XIME-P*, 2005.

[5] A. Arion, V. Benzaken, I. Manolescu, Y. Papakonstantinou, and R. Vijay. Algebra-based identification of tree patterns in XQuery. In *FQAS*, 2006.

[6] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.

[7] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *ACM Trans. Database Syst.*, 29(4), 2004.

[8] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4), 1981.

[9] M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Verification of tree updates for optimization. In *CAV*, 2005.

[10] M. Benedikt and J. Cheney. Schema-based independence analysis for XML updates. In *VLDB*, 2009.

[11] M. Benedikt and J. Cheney. Destabilizers and independence of XML updates. In *VLDB*, 2010.

[12] N. Bidoit, D. Colazzo, and F. Ulliana. Detecting XML query-update independence. In Bases de Données Avancées 2010 *(informal proceedings)*, 2010.

[13] H. Björklund, W. Gelade, M. Marquardt, and W. Martens. Incremental XPath evaluation. In *ICDT*, 2009.

[14] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *PODS*, 2006.

[15] Y. Chen, S. B. Davidson, and Y. Zheng. An efficient XPath query processor for XML streams. In *ICDE*, 2006.

[16] B. Choi, G. Cong, W. Fan, and S. Viglas. Updating Recursive XML Views of Relations. *J. Comput. Sci. Technol.*, 23(4), 2008.

[17] K. Dimitrova, M. El-Sayed, and E. A. Rundensteiner. Order-sensitive view maintenance of materialized XQuery views. In *ER*, 2003.

[18] M. El-Sayed, E. A. Rundensteiner, and M. Mani. Incremental maintenance of materialized XQuery views. In *ICDE*, 2006.

[19] J. N. Foster, R. Konuru, J. Simeon, and L. Villard. An algebraic approach to view maintenance for XQuery. In *PLAN-X workshop*, 2008.

[20] M. Franceschet. XPathMark: An XPath benchmark for the XMark generated data. In *XSym*, 2005.

[21] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM TODS*, 13(4), 1988.

[22] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[23] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.

[24] C. Koch. Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In *VLDB*, 2003.

[25] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *VLDB*, 2005.

[26] I. Manolescu, K. Karanasos, V. Vassalos, and S. Zoupanos. Efficient XQuery rewriting using multiple views. In *ICDE*, 2011. To appear.

[27] I. Manolescu and S. Zoupanos. XML materialized views in P2P. DataX workshop, 2009.

[28] M. Onizuka, F. Y. Chan, R. Michigami, and T. Honishi. Incremental maintenance for materialized XPath/XSLT views. In *WWW*, 2005.

[29] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting nested XML queries using nested views. In *SIGMOD*, 2006.

[30] A. Sawires, J. Tatemura, O. Po, D. Agrawal, A. E. Abbadi, and K. S. Candan. Maintaining XPath views in loosely coupled systems. In *VLDB*, 2006.

[31] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan. Incremental maintenance of path-expression views. In *SIGMOD*, 2005.

[32] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, 2002.

[33] N. Tang, J. X. Yu, M. T. Özsu, B. Choi, and K.-F. Wong. Multiple materialized view selection for XPath query rewriting. In *ICDE*, 2008.

[34] XML Path Language, 1999. http://www.w3.org/TR/xpath/.

[35] The XML Query Language, 2009. http://www.w3.org/XML/Query.

[36] The XQuery Update Facility 1.0, 2009. http://www.w3.org/TR/2009/CR-xquery-update-10-20090609/.

[37] L. Xu, T. W. Ling, H. Wu, and Z. Bao. DDE: from Dewey to a fully dynamic XML labeling scheme. In *SIGMOD*, 2009.

[38] W. Xu and M. Ozsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, 2005.