The view update problem for XML

Slawomir Staworko, Iovka Boneva, Benoit Groz

The View Update Problem for XML

Sławek Staworko[*]
University Lille 3
MOSTRARE, INRIA, Lille

Iovka Boneva
University Lille 1
MOSTRARE, INRIA, Lille

Benoît Groz
University Lille 1, ENS Cachan
MOSTRARE, INRIA, Lille

ABSTRACT

We study the problem of update propagation across views in the setting where both the view and the source database are XML documents. We consider a simple class of XML views that remove selected parts of the source document. The considered update operations permit to insert and delete subtrees of the document. We focus on constructing propagations that are 1) *schema compliant* i.e., when applied to the source document they give a document that satisfies the document schema; 2) *side-effect free* i.e., the view of the new source document is exactly as the result of applying the user update to the old view. We present a special structure allowing to capture all such propagations. We also show how to use this structure to capture only those propagations that affect minimally the parts of the document which are not visible in the view. Finally, we present a general outline of a polynomial algorithm constructing a unique propagation.

1. INTRODUCTION

Since its standardisation by the W3C [1], the use of the XML is constantly growing. Initially adopted as a data exchange format for Web applications, over the years XML has become popular to the point where native XML database management systems are constructed [2]. However, we researchers, and practitioners alike, agree that those systems are not as mature as e.g., existing relational DBMS. Many problems, well studied in the context of RDBMS, remain open for XML. In this paper, we address the view update problem well studied in the setting of relational databases [3, 4, 5, 6, 7].

The main role of database views is to provide an easy access to a portion of the data stored in the database by removing irrelevant parts and possibly restructuring the remaining parts [8]. While the view definition specifies how to select the data included in the view, it typically does not say what to do if the user wishes to change the contents of the view.

[*]Contact author: `slawomir.staworko@inria.fr`

The *view update problem* is stated as follows. Given a data t, a view definition A, and an update operation U on the view $A(t)$, find an update u of t which "correctly" propagates the changes of the view to the source document t. The precise meaning of "correctly" is to be defined. In the case of relational databases, several criteria have been considered. For instance, [4] proposes to consider only the *side-effect free* updates, that is $A(u(t)) = U(A(t))$. Intuitively, this means that the user performing the update does not see any unexpected changes in the view. In [5] Bancilhon and Spyratos additionally require that there are no side effects on the parts that are not included in the view; this is known as the *constant complement criterion.* This particular requirement guarantees the uniqueness of the constructed update propagation for certain classes of views [7]. Finally, one also requires *schema compliance* i.e. the updated database should satisfy the integrity constraints [6]. Due to the richer hierarchical structure of XML documents, the solutions proposed for relational databases cannot be directly applied and new approaches need to be developed.

In this paper, we address the view update problem for XML. We assume that both the source document and the view are XML documents (although the view needs not be materialized). The schema of the document is captured with Document Type Definitions (DTDs). We consider XML views obtained by removing selected parts of the document only. This class of views does not allow any restructuring of the data, however, it has various practical applications of which secure access to XML databases is one prominent example [9, 10]. The considered update operations are inserting and deleting a subtree. These operations are the backbone of the proposed XQuery Update facility [11] and are commonly considered in the context of incremental validation and incremental query evaluation for XML [12, 13].

We focus on constructing propagations that are *side-effect free* and *schema compliant.* While there might be an infinite number of such propagations, we present *propagation graphs* which capture all schema compliant and side-effect free propagations. Essentially, propagations correspond to paths in the propagation graphs. The graphs have size polynomial in the size of the source document, the schema, and the view update. Also, constructing updates from selected paths can be done in polynomial time. Thus, we consider the *propagation graphs* to be *compact representations* of all propagations.

One could adapt the constant complement criterion to select the propagation that does not affect the parts that are hidden by the view. While this approach produces at most one propagation, it may not exists. Instead, we select propagations that *minimally* modify the parts of the document that are not visible by the user. Such propagations always exist and their number is finite, although, it may be exponential. All such propagation can also be represented in a compact manner, using *optimal propagation graphs*, basically subgraphs induced by cheapest paths in the propagation graphs.

Finally, we discuss a general algorithm which selects one (optimal) propagation. This propagation is obtained by constructing paths in (optimal) propagation graphs. There are many ways of doing it and in this paper we discuss only a few of them. The algorithm, however, is parametrized by a general procedure selecting the desired path. We claim that if the procedure works in polynomial time, then the update can be propagated in polynomial time as well.

Organization of the paper. Section 2 presents basic notions used in this paper. In Section 3 we study the problem of constructing the inverse image of a view fragment. This problem plays an important role in our approach to update propagation which we present in Section 4. Section 5 contains an outline of a polynomial algorithm constructing a unique propagation. We discuss the related work in Section 6. Finally, in Section 7 we summarize the results and outline directions of further work.

Acknowledgments. We would like to thank Anne-Cécile Caron, Yves Roos, and Sophie Tison for their support and feedback in this work.

2. PRELIMINARIES

Trees. A *tree* over Σ is a finite structure $t = (\Sigma, N_t, \lhd_t, \prec_t, \lambda_t)$, where N_t is a finite set of node identifiers, $\lhd_t$ is the descendant relation, $\prec_t$ is the following sibling relation, and $\lambda_t : N_t \to \Sigma$ is the labeling function. We remark that both $\lhd_t$ and $\prec_t$ are irreflexive. The *size* of a tree t, denoted $|t|$, is the cardinality of N_t. A tree is empty is its node set is empty. We denote the root of a nonempty tree t by $root(t)$. Given a tree t and a node $n \in N_t$ by $t|_n$ we denote the *subtree* of t rooted at node n. Often, when considering a tree which is, or eventually will become, a subtree of another tree we call this subtree a tree *fragment*. In the sequel, we assume a fixed set Σ of node labels and by T_Σ we denote the set of all trees over Σ. A *(tree) language* over Σ is a subset of T_Σ.

In this paper we work with updates that essentially transform one tree into another. Node identifiers are used to identify the correspondence between the nodes in the tree before and after transformation. Since in this process new nodes can be inserted and some nodes deleted, we intentionally **do not assume** the set of node identifiers to be a prefix closed subset of $\mathbb{N}^*$. Also, the equality of trees should not be confused with isomorphism: two trees are equal iff all the elements of the underlying structures are the same, including the node set. Figure 1 contains an example of a tree t_0 (shown together with its node identifiers). We remark that if the particular choice of node identifiers is not important, we simply denote trees as terms over Σ for sake of clarity.

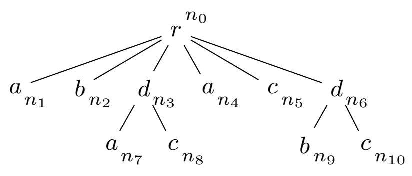


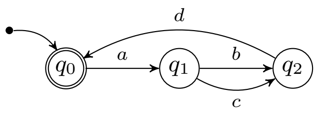
Figure 1: A tree t_0.

Automata and DTDs. A *finite automaton* over Σ is a tuple $M = (\Sigma, Q, q_0, \delta, F)$, where Q is a finite set of states, $q_0 \in Q$ is a distinguished starting state, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $F \subseteq Q$ is the subset of accepting states. By $L(M)$ we denote the set of words over Σ recognized by M. The size of M, denoted $|M|$ is the sum $|Q| + |\delta| + |F|$.

A *Document Type Definition* over Σ (DTD) is a function D that maps a symbol $a \in \Sigma$ to an automaton M_a that specifies the allowed sequences of children of a node labeled with a. A tree $t \in T_\Sigma$ *satisfies* D iff for every node $n \in N_t$ the word consisting of consecutive labels of children of n belongs to $L(D(\lambda_t(n)))$. By $L(D)$ we denote the set of all nonempty trees that satisfy D. The size of a DTD is the sum of the sizes of all automata used. Typically, DTDs specify also the required label of the root. We omit this requirement as this will allow us to easily consider tree fragments that satisfy the DTD. We remark, however, that our constructions can be easily extended to include this additional requirement.

In the examples, we specify DTDs using rules mapping symbols in Σ to regular expressions over Σ defined in the standard fashion. If for a symbol a no rule is given, then $a \to \epsilon$ is assumed. Also, we consider only satisfiable DTD, i.e. such that for every symbol $a \in \Sigma$ there exists a tree satisfying the DTD and whose root label is a. Naturally, testing satisfiability of a DTD can be done in polynomial time [14]. Figure 2 contains an example of a DTD D_0 specified with two rules and the corresponding automata. Note that t_0 satisfies D_0.

$r \to (a \cdot (b + c) \cdot d)^*$



$d \to ((a + b) \cdot c)^*$

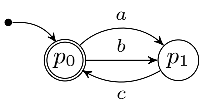


Figure 2: A DTD D_0 and two automata.

Annotations and views. In this paper, we consider views obtained by hiding selected nodes of the source document. To identify the visible nodes we use annotations. They are commonly used, for instance, to specify security views of XML documents [9, 10]. Typically, they accompany DTDs, but here we introduce them independently of the DTD.

Formally, an *annotation* is a function $A : \Sigma \times \Sigma \to \{\mathbb{0}, \mathbb{1}\}$. Given a non-empty tree t, the set $[\![A]\!]_t \subseteq N_t$ of *visible* nodes is defined recursively: 1) the root node is always visible; 2) if a node n has a visible parent p, then n is visible if and only if $A(\lambda_t(p), \lambda_t(n)) = \mathbb{1}$; 3) in all other cases the node is hidden. Note that the visibility of nodes is *upward closed* [15], i.e. all descendants of a hidden node are hidden as well.

The view of a tree consists of visible nodes only. Formally, a *view* of $t \in L$ defined by A is a tree $t' = (\Sigma, [\![A]\!]_t, \lhd_t \cap [\![A]\!]_t^2, \prec_t \cap [\![A]\!]_t^2, \lambda_t \restriction [\![A]\!]_t^2)$, where by $f \restriction X$ we denote the restriction of function f to the set X. In the sequel, we abuse the notation and by $A(t)$ denote the view of t w.r.t. A. In examples, we specify annotation only on the essential pairs of symbols; the annotation is assumed to be $\mathbb{1}$ on the remaining pairs. Figure 3 contains an example of an annotation A_0 and the view $A_0(t_0)$.

$$
\begin{aligned}
A_0(r,a) &= A_0(r,d) = \mathbb{1}\\
A_0(r,b) &= A_0(r,c) = \mathbb{0}\\
A_0(d,a) &= A_0(d,b) = \mathbb{0}\\
A_0(d,c) &= \mathbb{1}
\end{aligned}
$$

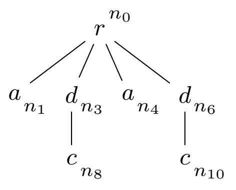


Figure 3: An annotation A_0 and the view $A_0(t_0)$.

By $A(L)$ we denote the set of all views of trees in L. We remark that a DTD capturing $A(L(D))$ can be easily derived from D and A. For instance, the view DTD for D_0 and A_0 is

$$r \to (a \cdot d)^* \qquad\qquad d \to c^*$$

Editing scripts. We consider two standard editing operations: inserting and deleting a subtree. To represent the updates performed by the user on the document we use a formalism based on tree alignments commonly used in the context of measuring similarities between trees [16]. This formalism allow us to associate with every node exactly one editing operation. For consistency, with nodes that are not affected by the update we associate a special *phantom* operation which does nothing.

Formally, an *editing script* over Σ is a tree over the alphabet $\mathcal{E}(\Sigma)$ defined as

$$\mathcal{E}(\Sigma) = \{Ins(a), Nop(a), Del(a) \mid a \in \Sigma\}.$$

$Ins(a)$ is an insertion of a node, $Del(a)$ is a deletion of a node, and $Nop(a)$ is a phantom operation. Since we consider only updates that insert and delete whole trees, we require that all descendants of an inserting node are inserting as well, and similarly all descendant of a deleting node are deleting. The *cost* of an editing script S is the number of nodes that are labeled with a non-phantom operation. Figure 4 contains an example of an editing script.

This particular representation of document updates allows us to identify not only the update but also the original and the resulting document and the correspondence between the nodes of those trees. Formally, the *input tree* $In(S)$ of an

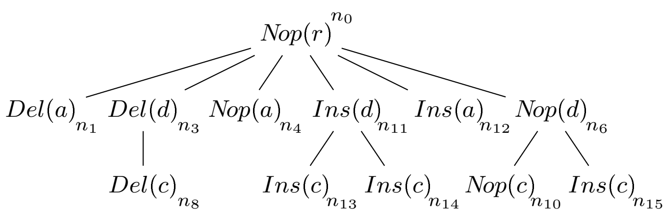


Figure 4: An update S_0 of the view $A_0(t_0)$.

editing script $S = (\mathcal{E}(\Sigma), N_S, \lhd_S, \prec_S, \lambda_S)$, is defined as

$$
\begin{aligned}
&(\Sigma, N_{In}, \lhd_S \cap (N_{In})^2, \prec_S \cap (N_{In})^2, \lambda_{In(S)}),\\
&N_{In} = \{n \in N_S \mid \lambda_S(n) \neq Ins(a) \text{ for any } a \in \Sigma\},\\
&\lambda_{In(S)}(n) = a \text{ if } \lambda_S(n) = Del(a) \text{ or } \lambda_S(n) = Nop(a).
\end{aligned}
$$

The *output tree* $Out(S)$ of S is defined analogously. For instance, the input tree of S_0 in Fig. 4 is the tree $A_0(t_0)$ in Fig 3. Its output tree is presented in Fig. 5.

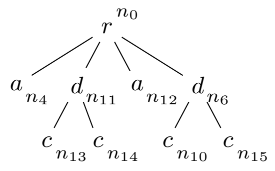


Figure 5: The output tree of S_0.

We remark that inclusion of the node identifier in the update may seem limiting for reasoning about abstract updates. However, in the setting of update propagation, the context of the update is an integral part of the problem input. Thus, from now on we do not make a formal distinction between the update and its editing script, and we refer to a script S with the input tree t as an *update* of t. Also, if t is the input tree of S and t' its output tree, we write $S(t) = t'$.

We also overload the symbols $Ins(.)$, $Del(.)$, $Nop(.)$ to trees. For instance, by $Ins(t)$ we denote the unique editing script S such that $In(S)$ is an empty tree and $Out(S) = t$.

3. WARM-UP: VIEW INVERSE

In this section we focus on the *view inversion* problem: given a view document t' construct a source document t, called an *inverse* of t', that yields exactly the same view, i.e. $A(t) = t'$. This problem is an integral part of the update propagation problem because a propagation of an update fragment which inserts a subtree is an update that inserts the inverse of the subtree. One could attempt to use the solution of view inverse problem to solve the problem of view update propagation by simply constructing the inverse of the updated view. As we point out in Section 6.2 this approach disregards the relative positions of nodes affected by the update, and consequently may yield inadequate and erroneous solutions.

Formally, the *inverse operation* of a view t' w.r.t. a tree language L and annotation A is

$$Inv(L, A, t') = \{t \in L \mid A(t) = t'\}.$$

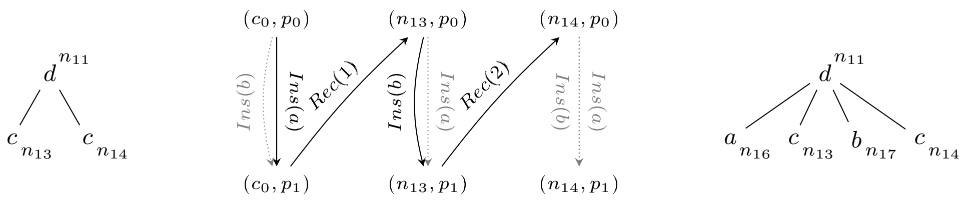


Figure 6: A view fragment, its inversion graph, and its inverse.

Note that $Inv(L, A, t')$ is not closed under isomorphism because its elements need to contain at least the visible nodes of t'.

To capture the inverse operation of a view t' w.r.t. a DTD D and annotation A we construct a collection $\mathcal{H}(D, A, t')$ of directed labeled graphs $(H_n)_{n \in N_{t'}}$, one for each node of t'. We fix a node $n \in N_{t'}$ and let $x = \lambda_{t'}(n)$ and $D(x) = M(\Sigma, Q, q_0, \delta, F)$. We also identify the sequence $m_1, \ldots, m_k$ of children nodes of n in t'. The *inversion graph* $H_n = (V_n, E_n)$ is defined as follows. The set of vertices is $V_n = \{c_0, m_1, \ldots, m_k\} \times Q$, where c_0 is a *fresh* element, different for every n, to which we will also refer as m_0. The set E_n consists of two types of edges:

(i) $(m_i, q) \xrightarrow{Ins(y)} (m_i, q')$ for any $q \xrightarrow{y} q' \in \delta$ such that $A(x, y) = \mathbb{0}$ (for $i \in \{0, \ldots, k\}$);

(ii) $(m_{i-1}, q) \xrightarrow{Rec(i)} (m_i, q')$ for any $q \xrightarrow{y} q' \in \delta$ such that $A(x, y) = \mathbb{1}$ and $\lambda_{t'}(m_i) = y$ (for $i \in \{1, \ldots, k\}$).

An *inversion path* in H_n is a (possibly cyclic) directed path from (c_0, q_0) to (m_k, q) with $q \in F$.

Now, for a given choice of exactly one inversion path in every H_n (for $n \in N_t$) we construct a source document in a bottom-up fashion. For H_n and its inversion path we construct the tree whose root node is n labeled with $\lambda_{t'}(n)$ and its subtrees are obtained by traversing the path as follows. For a (i)-edge we add a tree satisfying D with root node label y. Every time we traverse this edge, the trees used need not be the same and in particular each time we use *fresh* nodes. For a (ii)-edge we add the tree obtained from H_{m_i} and its inversion path. Finally, the source tree t is the tree obtained from $H_{root(t')}$ and its inversion path. We remark that the resulting tree depends not only on the choices of paths but also on the choice of a minimal subtrees used for (i)-edges. Figure 6 contains an example of an inversion graph $H_{n_{11}}$ for a subtree of $Out(S_0)$ at n_9 (w.r.t. D_0 and A_0), a selected inversion path, and the corresponding inverse tree.

We claim that any tree obtained from an inversion path is an inversion of t', and vice versa, i.e. for any inversion of t' (w.r.t. A and D), there exists a corresponding choice of inversion paths (together with a choice of subtrees used for traversing (i)-edges).

THEOREM 1. *For a DTD D, an annotation A, and a view tree t', $\mathcal{H}(D, A, t')$ captures $Inv(L(D), A, t')$.*

We are also interested in the set of view inversions that add a minimal amount of new (invisible) nodes. Formally, we take the set $Inv_{\min}(L, A, t')$ of size-minimal elements of $Inv(L, A, t')$. To capture this set in every inversion graph we add weights to edges. The choice of weights may be arbitrary for (ii)-edges because every inversion path must contain exactly one edge with $Rec(i)$ for every $i \in \{1, \ldots, k\}$. Here, we assign weights that not only allow constructing minimal inversions but moreover allow an easy calculation of the minimal number of nodes that need to be added to obtain the inversion.

The weight of a (i)-edge is equal to the minimal size of a tree satisfying D and with root label y. Note that this value is greater than 0 and can be easily precomputed from D in polynomial time. The weight of a (ii)-edge is set to the minimal cost of a inverting path in H_{m_i} (calculated recursively).

Now, by $H_n^\star$ we denote the subgraph of H_n induced by the cheapest inversion paths. We remark that $H_n^\star$ is acyclic. By $\mathcal{H}^\star(D, A, t')$ we denote the collection of *optimal inversion graphs* $(H_n^\star)_{n \in N_{t'}}$ for t' w.r.t. D and A. Naturally, when constructing a source tree from the optimal inversion graphs, traversal of a (i)-edge adds a minimal tree satisfying D with root label y, and traversal of a (ii)-edge adds an optimal inversion obtained from $H_{m_i}^\star$.

THEOREM 2. *For a DTD D, an annotation A, and a view tree $t' \in A(L(D))$, $\mathcal{H}^\star(D, A, t')$ captures $Inv_{\min}(L(D), A, t')$.*

Finally, we observe that both the size of $\mathcal{H}(D, A, t')$, and thus its optimal version as well, is polynomial in the size of D and t'.

4. VIEW UPDATE PROBLEM

We begin by formalizing the problem. Take a language L of admissible source documents (possibly expressed with a DTD), an annotation A, and let $V = A(L)$ be the tree language of possible views which we assume to be known to the user. Assume also some source document $t \in L$. Now, a *view update* is an editing script S such that $In(S) = A(t)$ and $Out(S) \in V$. For technical reasons, we require that the update does not use the nodes that are hidden by the view definition, i.e. $N_S \cap (N_t \backslash N_{A(t)}) = \varnothing$. This requirement prevents situations where the user attempts to add a node with identifier already used by an existing node in the source document and not visible to the user.

Now, a *propagation* of S is any editing script S' such that $In(S') = t$. We say that 1) S' is *schema compliant* if

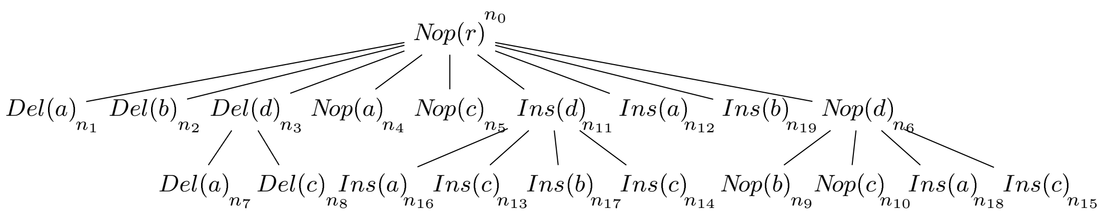


Figure 7: An optimal side-effect free propagation of S_0.

$Out(S') \in L$; 2) S' is *side-effect free* if $A(Out(S')) = Out(S)$. By $P(L, A, t, S)$ we denote the set of all pairs schema compliant and side-effect free propagations of S for t, L, and A. Fig. 7 contains a schema compliant and side-effect free propagation of S_0 (Fig. 4).

Compact representation. Now, we present a construction that allows to capture desirable propagations. We remark that this construction can be seen as an extension of inversion graphs which handles not only insertions, but also deletions and *Nop*-operations. We fix a DTD D, an annotation A, a source document $t \in L(D)$, and a view update S. We identify the set of the view nodes of the source document t that appear in the updated version of the view

$$N_\Delta = \{n \in N_S \mid \lambda_S(n) = Nop(a) \text{ for some } a \in \Sigma\}.$$

Note that $N_\Delta \subseteq N_{A(t)} \subseteq N_t$. We construct a collection of directed labeled graphs $\mathcal{G}(D, A, t, S) = (G_n)_{n \in N_\Delta}$, one for each node in N_Δ. We fix a node $n \in N_\Delta$ and let $x = \lambda_t(n)$ and $D(x) = M(\Sigma, Q, q_0, \delta, F)$. Now, let $m_1, \ldots, m_k$ be the sequence of children of n in t and $m'_1, \ldots, m'_\ell$ be the sequence of children of n in S. Typically, these two sequences have common nodes. Let

$$N_C = \{c_0\} \cup \{m_1, \ldots, m_k\} \cap \{m'_1, \ldots, m'_\ell\},$$

where c_0 is an artificial common node that will be refereed to as m_0 and m'_0.

We partition the sequence $m_0, m_1, \ldots, m_k$ into segments contained between two consecutive common nodes. Formally, the segments starting at a common node $m_i \in N_C$ is

$$seg_t(m_i) = \{m_j \in N_t \mid i \leqslant j \wedge \nexists i' \in \{i+1, \ldots, j\}.\, m_{i'} \in N_C\}.$$

Analogously, in the sequence $m'_0, m'_1, \ldots, m'_\ell$ we identify the segment $seg_S(m'_j)$ starting at a common node m'_j. We remark that for all $m \in N_C$ the elements of $seg_t(m) \backslash \{m\}$ are hidden by A and all elements of $seg_S(m) \backslash \{m\}$ are inserted by S. Consequently, we need to consider all ways of *shuffling* the contents of each pair of two corresponding segments.

Now, the *propagation graph* G_n is defined as follows. The set of vertices is $V = \bigcup_{m \in N_C}(seg_t(m) \times Q \times seg_S(m))$. The set E consists of the following edges: for $y \in \Sigma$ such that $A(x, y) = \mathbb{0}$ we have

(i) $(m_i, q, m'_j) \xrightarrow{Ins(y)} (m_i, q', m'_j)$ for any $q \xrightarrow{y} q' \in \delta$; (*invisible insert*)

(ii) $(m_{i-1}, q, m'_j) \xrightarrow{Del(y)} (m_i, q, m'_j)$ if $\lambda_t(m_i) = y$; (*invisible delete*)

(iii) $(m_{i-1}, q, m'_j) \xrightarrow{Nop(y)} (m_i, q', m'_j)$ for any $q \xrightarrow{y} q' \in \delta$ and if $\lambda_t(m_i) = y$; (*invisible nop*)

and for $y \in \Sigma$ such that $A(x, y) = \mathbb{1}$ we have

(iv) $(m_i, q, m'_{j-1}) \xrightarrow{Ins(y)} (m_i, q', m'_j)$ for any $q \xrightarrow{y} q' \in \delta$ and if $\lambda_S(m'_j) = Ins(y)$; (*visible insert*)

(v) $(m_{i-1}, q, m'_{j-1}) \xrightarrow{Del(y)} (m_i, q, m'_j)$ if $\lambda_t(m_i) = y$ and $\lambda_S(m'_j) = Del(y)$; (*visible delete*)

(vi) $(m_{i-1}, q, m_{j-1}) \xrightarrow{Nop(y)} (m_i, q', m'_j)$ for any $q \xrightarrow{y} q' \in \delta$ and if $\lambda_t(m_i) = y$ and $\lambda_S(m'_i) = Nop(y)$; (*visible nop*)

A *propagation path* in G_n is a (possibly cyclic) directed path from (c_0, q_0, c_0) to (m_k, q, m'_ℓ) such that $q \in F$. Figure 8 contains the propagation graph G_{n_6} for t_0 and S_0 (w.r.t. D_0 and A_0) with one chosen propagation path.

$$\begin{tikzcd}
(c_0,p_0,c_0) \arrow[r, dotted, "Del(b)"] \arrow[d, dotted, shift right, "Ins(a)"'] \arrow[d, dotted, shift left, "Ins(b)"] \arrow[dr, "Nop(b)"] & (c_0,p_0,n_9) \arrow[d, dotted, shift right, "Ins(a)"'] \arrow[d, dotted, shift left, "Ins(b)"] & (n_{10},p_0,n_{10}) \arrow[d, "Ins(a)"] \arrow[d, dotted, bend right, "Ins(b)"'] & (n_{10},p_0,n_{15}) \arrow[d, dotted, shift right, "Ins(a)"'] \arrow[d, dotted, shift left, "Ins(b)"] \\
(c_0,p_1,c_0) \arrow[r, dotted, "Del(b)"] & (c_0,p_1,n_9) \arrow[ur, "Nop(c)"] & (n_{10},p_1,n_{10}) \arrow[ur, "Ins(c)"] & (n_{10},p_1,n_{15})
\end{tikzcd}$$

Figure 8: The propagation graph G_{n_6}.

Now, given a choice of exactly one propagation path in every G_n (for $n \in N_\Delta$) we construct a propagation of S as follows. For $n \in N_\Delta$, the script corresponding to G_n has its root node n labeled with $Nop(\lambda_t(n))$ and its subtrees are obtained from traversing the propagation path:

- For (i)-edge we add a subtree $Ins(t'')$, where t'' is some arbitrarily chosen tree satisfying D with root label y (using *fresh* nodes).
- For (ii)-edge and (v)-edge we add $Del(t|_{m_i})$.
- For (iii)-edge we add the subtree $Nop(t|_{m_i})$.
- For (iv)-edge we let $t' = Out(S|_{m'_j})$, take any $t'' \in Inv(L(D), A, t')$, and add the subtree $Ins(t'')$. For (vi)-edge we add the script generated recursively from G_{m_i} and its propagation path.

For instance the path in Fig. 8 yields a script presented in Fig. 9.

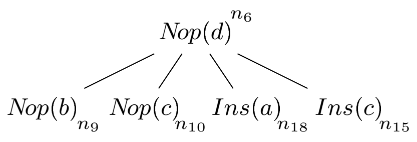


Figure 9: An update fragment obtained from G_{n_6}.

We remark that the constructed propagation depends on the selected choice of propagation paths and the selected trees used when traversing (i)-edges. Our claim is that the obtained script is a side-effect free propagation of S. Moreover, all side-effect free propagations can be obtained in this fashion.

THEOREM 3. *$\mathcal{G}(D, A, t, S)$ captures $P(L(D), A, t, S)$ for any DTD D, any annotation A, any source tree $t \in L(D)$, and any update S of the view $A(t)$.*

Optimal propagations. We remark that a view update may have infinitely many side-effect free and schema compliant propagations. For instance, consider the DTD $D_1 : r \to (a \cdot b^*)^*$ with an annotation $A_1(r, a) = \mathbb{1}$ and $A_1(r, b) = \mathbb{0}$. Regardless of the source document, inserting in the view a node labeled with a may be propagated to an update that inserts a and an arbitrary number of invisible nodes b. To limit the amount of invisible nodes that the propagation may add, we consider only the cost optimal update propagations. Formally, by $P_{\min}(L, A, t, S)$ we denote the subset of cost minimal elements of $P(L, A, t, S)$. In the previous example, an update inserting a node a is propagated to an update that inserts this node only.

To capture the set of optimal propagations, we add weights to the edges of propagation graphs. We assume D, A, t, and S to be given as before and we fix $n \in N_\Delta$. For a (i)-edge the weight is the size of a minimal tree satisfying D and with root label b. For a (ii)-edge and a (v)-edge the weight is the size of the subtree to be deleted $t|_{m_i}$. For a (iii)-edge the weight is 0. For a (iv)-edge the weight is the size of a minimal view inversion of $Out(S|_{m'_j})$, which we calculate using the optimal inverse graph. For a (vi)-edge the weight is the cost of the cheapest propagation path in G_{m_i}, which we calculate recursively.

Now, by $G^\star_n$ we denote the subgraph of G_n induced by the cheapest propagation paths of G_n. By $\mathcal{G}^\star(D, A, t, S)$ we denote the collection of *optimal propagation graphs* $(G^\star_n)_{n\in N_\Delta}$ for t and S w.r.t. D and A. Naturally, when constructing a script using path in $G^\star_n$ we use only optimal elements. In particular, when traversing (i)-edge we use a minimal tree satisfying D and whose root node is y, and when traversing (iv)-edge we take an optimal view inverse. Figure 10 contains the optimal propagation graph $G^\star_{n_0}$.

THEOREM 4. *$\mathcal{G}^\star(D, A, t, S)$ captures $P_{\min}(L(D), A, t, S)$ for any DTD D, any annotation A, any source tree $t \in L(D)$, and any update S of the view $A(t)$.*

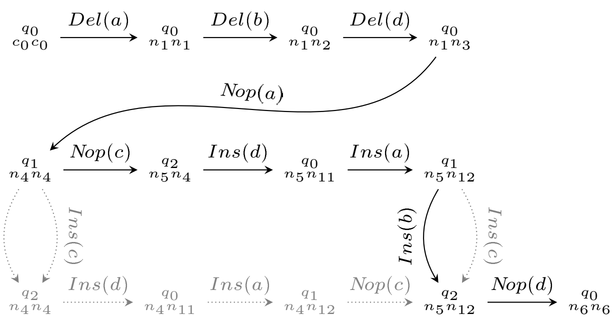


Figure 10: The optimal propagation graph $G^\star_{n_0}$.

Finally, we remark that $\mathcal{G}^\star(D, A, t, S)$ and its optimal version can be constructed in time polynomial in the size of D, t, and S.

Further results. Theorems 1, 2, 3, and 4 make propagation and inversion graphs a useful tool for reasoning about the view update problem in general. For instance, we observe that the optimal inverse and propagation graphs have only acyclic inverse and propagation paths. This shows that the number of optimal side-effect free and schema compliant propagations has an exponential upper bound. This bound is tight as illustrated by the following example. Take DTD D_2 where $r \to (a \cdot (b + c))^*$ with annotation A_2 such that $A_2(r, a) = \mathbb{1}$ and $A_2(r, b) = A_2(r, c) = \mathbb{0}$. Clearly, inserting a node labeled with a requires insertion of a node labeled either by c or b. Consequently, inserting k nodes a has 2^k optimal propagations since the choices are independent.

Using more elaborate arguments we can also show that every view update has a side-effect free and schema compliant propagation.

THEOREM 5. *For every DTD D, any annotation A, any source tree $t \in L(D)$, and any view update S, i.e. $In(S) = A(t)$ and $Out(S) \in A(L(D))$, there exists a propagation S' of S, i.e. $In(S') = t$ and $Out(S') \in L(D)$.*

5. PROPAGATION ALGORITHM

In this section we discuss a construction of a tractable view update propagation algorithm based on optimal propagation and inversion graphs.

In essence, the algorithm works as follows:

1. It constructs the collection of the optimal propagation graphs for the source document and the input view update.
2. For all new trees inserted by the view update it constructs the corresponding optimal inversion graphs.
3. It chooses exactly one propagation (inversion) path in every optimal propagation (inversion reps.) graph.

4. It recursively constructs the propagation of the view update using the propagation and inversion graphs with the selected paths.

For instance Figure 7 contains an example of propagation of the update S_0 when using the paths selected on Figures 6, 8, and 10.

We observe one peculiarity of update propagation which is a consequence of the fact that a minimal tree satisfying a DTD may be of size exponential in the size of the DTD. For instance, consider the following DTD (with $i \in \{n, \ldots, 1\}$)

$$a \to a_n \cdot a_n \qquad a_i \to a_{i-1} \cdot a_{i-1} \qquad a_0 \to \epsilon$$

One of the resulting inconveniences is that the XML view update problem is inherently exponential: propagation of a simple view update may require insertion of a subtree exponential in the size of the DTD.

One could remove the size of the DTD from complexity analysis, but we will assume that the administrator specifies default XML document fragments, called *insertlets*, that are used to insert the invisible subtrees. This assumption is quite natural and reasonable: rather than inserting an arbitrary fragments into the source document, one might prefer to specify the fragments to be use should the necessity arise. At the same time, it allows us to characterize more precisely the complexity of view update propagation.

An *insertlet package* for D is a collection $\mathcal{W} = (W_a)_{a \in \Sigma}$ containing for every $a \in \Sigma$ an insertlet W_a, i.e. a minimal tree satisfying D with root label a. We remark that in practice it will not be necessary to specify an insertlet for every symbol but here we do not enter in those details.

So far we said little as to how a unique path in every propagation and inversion graph is to be selected. Because of space limitations we only outline some approaches that can be used to reduce the number of the considered cheapest paths and eventually lead to one unique update propagation. First, we propose to use typing of nodes to identify updates which do not change the types of nodes that are preserved by the update. Formally, a document *typing* is a function Θ which maps a tree t to a function $\Theta_t : N_t \to \Gamma$, where Γ is a set of types. A propagation S' of a view update S *preserves* Θ*-typing* iff for every $n \in N_{In(S')} \cap N_{Out(S')}$ we have $\Theta_{In(S')}(n) = \Theta_{Out(S')}(n)$. One possible typing could be based on rich schema formalisms, like EDTD [17, 18]. Another possible typing could use the states of the automaton used to verify that the sequence of children is valid w.r.t. the DTD. It would require the automata to be deterministic, however, it is a commonly enforced requirement for DTDs [1, 18].

Finally, a unique update propagation can be defined by using preferences on edges to be selected when constructing the optimal propagation path in $\mathcal{G}^\star(D, A, t, S)$. For example, the selected propagation path in Figure 10 is the result of preference of *Nop*-edges over *Ins*-edges.

We assume that we are given a function Φ which allows to select the unique preferred paths in inversion and propagation graphs and that it works in time polynomial in the size of the graphs. Consequently, we obtain

THEOREM 6. *Given a DTD D, an annotation α, a source document t, and a view update S, a side-effect free propagation S' of S w.r.t. a polynomial preference function Φ and insertlets $\mathcal{W}$ can be computed in time polynomial in the size of D, t, S, and $\mathcal{W}$.*

6. RELATED WORK

6.1 View Update Problem

A recent thread of work by Foster, Pierce et al. studies so called *lenses* [19, 20]. These are bi-directional tree transformers (view definitions) that provide two operations: get and put. The *get* operation allows to compute an abstract view of a concrete tree. The *put* operation takes an updated version of the abstract view, together with the original concrete tree, and correspondingly updates the original tree. This way the view definition itself allows to compute the update propagation. In particular, this implies that the transformer defines explicitly constant tree values to be used when some information is missing. Each lens definition comes with two types, for concrete trees and abstract views. The type of the abstract view defines also the allowed updates on the view. Types also guarantee that lenses are "well-behaved" [19]. The PutGet rule from [19] corresponds to absence of side-effects.

In contrast to our view definitions by annotations, lenses allow not only node filtering, but also local tree transformations, such as inserting a node or a constant tree. Another, important difference with our work is that [19] considers so called *feature trees* – unordered, edge-labelled trees with no repeated labels among sibling edges. Consequently, working with XML requires encoding of XML trees into feature trees, whereas our approach allows to directly work on XML trees. In [20], lenses are extended in order to capture confidentiality and integrity, which allows to deal with security issues.

Several authors consider updating XML views of relational databases [21, 22, 23]. For instance, [22] focuses on translating XML view updates to relational view updates and delegating the problem to the relational DBMS, [21] studies the conditions under which a view update is translatable, and [23] provides algorithms for the translation of a rich class of view updates. There exist numerous approaches storing XML documents in relational databases, e.g. [24, 25], and one could attempt to combine them with the view propagation solutions. However, the complexity of view definitions required to *reconstruct* the XML documents is beyond the capabilities of the existing propagation solutions.

6.2 XML Repairing

One may attempt to solve the view update problem using solutions for XML repairing [26] as described below. Take a DTD D, an annotation A, and let t' be the result of applying the user update on the view $A(t)$ for some source document $t \in L(D)$. Now, let L' be the set $Inv(L(D), A, t')$ closed under isomorphism, i.e. the set of all source documents satisfying D and whose view gives t' disregarding the identifiers. This set is a regular language of trees and a way of propagating the update to the source document is choosing from L' the tree closes to the original tree t, i.e. repairing t w.r.t. L'. We argue that by dropping the node identifiers this approach inadvertently looses information allowing it to correlate the relative positions of existing and new nodes. We illustrate this with an example.

Take a DTD D_3 $r \rightarrow b\cdot(c+\epsilon)\cdot(a\cdot c)^*$ with the annotation A_3 such that $A_3(r,b) = A_3(r,a) = \mathbb{0}$ and $A_3(r,c) = \mathbb{1}$. The view DTD is $r \rightarrow c^*$. Now, let $t = r(b,a,c)$, and then $A(t) = r(c)$. Suppose that the user inserts a child c as **the last child** of the root node r resulting in $t' = r(c,c)$. There are two trees satisfying D, $t_1 = r(b,c,a,c)$ and $t_2 = r(b,a,c,a,c)$, for which the view w.r.t. A is t'. While t_1 is closer to t than t_2, it is obvious that t_2 is better suited for the updated source document. One reason is that the user inserts the new node c at a position following the node c already existing in the source document.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have addressed the view update problem in the setting of simple XML views, defined with annotations, and basic yet ample update operations. We have taken an approach constructing side-effect free and schema compliant update propagations, well established in the setting of relational databases. The solutions for relational databases are, however, incompatible due to a richer structure of XML document. Consequently, we have devised novel solutions tailored to the semi-structured databases. We have presented a construction of *(optimal) propagation graphs* which allow to capture all (optimal) schema compliant and side-effect free propagations. We have also outlined a general algorithm which uses the *propagation graphs* to construct a desired update propagation. To the best of our knowledge, our work is the first to provide a complete and self-contained solution for the view update problem for XML.

The work presented in this paper is our first step towards a comprehensive framework for XML view update propagation, a task which turns out to be as challenging for XML databases as it is for relational databases. Although the considered classes of views and updates already have several possible practical applications, our first goal will be to extend them and further increase the appeal of our framework. We believe that the framework can be extended to handle more general update operations including renaming a node, deleting an inner node , and inserting an inner node [27]. More challenging is extending our approach to more powerful view formalisms allowing restructuring of the document. In our first attempt, we will explore formalisms based on *Visibly Pushdown Transducers* [28], which allow deleting, renaming, and inserting nodes of a tree. Also, extending the framework to allow richer document schema languages, e.g. EDTDs [17, 18], should be feasible with further employment of general tree automata techniques [29].

We also intend to devise an administrator-friendly manner of defining preferences on the choice of the desired update propagation. This also includes defining further correctness criteria for update propagation with efficient algorithms for their construction. We also plan to study variants of the notion of side-effect free propagation in the setting where several user views are given. Finally, we intend to investigate security issues raised by view updates, such as confidentiality and data integrity.

8. REFERENCES

[1] W3C. Extensible markup language (XML) 1.0, 1999.
[2] A. Vakali, B. Catania, and A. Maddalena. XML data stores: Emerging practices. *IEEE Internet Computing*, 2005.
[3] E. F. Codd. Recent investigations in relational data base systems. In *IFIP Congress*, 1974.
[4] U. Dayal and P. Bernstein. On the correct translation of update operations on relational views. *TODS*, 1982.
[5] F. Bancilhon and N. Spyratos. Update semantics of relational views. *TODS*, 1981.
[6] S. Cosmadakis and C. Papadimitriou. Updates of relational views. *Journal of the ACM*, 1984.
[7] J. Lechtenbörger and G. Vossen. On the computation of relational view complements. *TODS*, 2003.
[8] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. WCB/McGraw-Hill, 2000.
[9] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Rewriting regular XPath queries on XML views. In *ICDE*, 2007.
[10] B. Groz, S. Staworko, A.-C. Caron, Y. Roos, and S. Tison. XML security views revisited. In DBPL, 2009.
[11] W3C. XQuery update facility 1.0, 2009.
[12] H. Björklund, W. Gelade, M. Marquardt, and W. Martens. Incremental XPath evaluation. In *ICDT*, 2009
[13] A Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *TODS*, 2004.
[14] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *PODS*, 2005.
[15] M. Benedikt and I. Fundulaki. XML subtree queries: Specification and composition. In *DBPL*, 2005.
[16] T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. *TCS*, 1995.
[17] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *PODS*, 2000.
[18] W. Martens, F. Neven, T. Schwentick, and G. J. Bex. Expressiveness and complexity of XML schema. *TODS*, 2006.
[19] J. Foster, M. Greenwald, J. Moore, B. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *TOPLAS*, 2007.
[20] J. Foster, B. Pierce, and S. Zdancewic. Updatable security views. In *Computer Security Foundations Symposium*, 2009.
[21] L. Wang, E. Rundensteiner, and M. Mani. Updating XML views published over relational databases: Towards the existence of a correct update mapping. *DKE*, 2006.
[22] V. Braganholo, S. Davidson, and C. Heuser. PataxÓ: A framework to allow updates through XML views. *TODS*, 2006.
[23] B. Choi, G. Cong, W. Fan, and S. Viglas. Updating recursive XML views of relations. *Journal of Computer Science and Technology*, 2008.
[24] P. Boncz, T. Grust, M. Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD*, 2006.
[25] I. Tatarinov, K. Beyer, and J. Shanmugasundaram. Storing and querying ordered XML using a relational database system. In *SIGMOD* , 2002.
[26] S. Staworko and J. Chomicki. Validity-sensitive querying of XML databases. In *EDBT Workshops (dataX)*, 2006.
[27] D. Shasha and K. Zhang. Approximate tree pattern matching. In A. Apostolico and Z. Galil, editors, *Pattern Matching in Strings, Trees, and Arrays*. 1997.
[28] F. Servais and J.-F. Raskin. Visibly pushdown transducers. In *ICALP*, 2008.
[29] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, Release 2007.