Updating XML Views

Roel Vercammen*

University of Antwerp, Belgium
roel.vercammen@ua.ac.be

Abstract

Views are an important feature of database systems to provide abstraction and guarantee a certain level of security. The problem of updating a database through view updates is well-established and has been widely studied in the context of relational databases. This work tries to point out what complications arise when considering the view-update problem in an XML setting and what research questions come up. Since both views and updates on XML are not yet completely understood, we first give some hints at the implications of design choices made in the construction of languages for both defining views and updating XML.

1 Introduction

The problem of updating databases through views was first reported in 1974 by Codd [5] and has been widely studied ever since. Views are often used as a security mechanism or just for the convenience of the user who knows the base schema, but updates through views are often impossible or ambiguous. Even when it is possible to uniquely identify which updates have to be performed on the original data, updating instances of the base schema can have undesired side effects. In order to eliminate side effects, [3] introduces the notion of constant-complement view updates, where each update is reversible and does not have any effect on the "complement" of the view. Hegner [12] refines this notion and introduces the notion of open and closed update strategies. In *open view update strategies* it is assumed that the view is only defined for convenience. In this case, an update strategy allows as many updates as possible and the user is expected to be aware of consequences that updates can have. On the other hand, *closed views* are totally encapsulated and it is assumed that the user does not have any knowledge of the base schema. Hence the look and feel of the view should be the same as that of a base schema and the effects of the updates should be limited to that part of the base schema which is reflected in the view.

*Roel Vercammen is supported by IWT – Institute for the Encouragement of Innovation by Science and Technology Flanders, grant number 31581.

Most work on view updatability in the relational model has concentrated on open view update strategies because closed view update strategies are often found too conservative. However, closed view update strategies are a very interesting subject of study, since they represent the class of updates that users can fully understand [18] and they offer more security by limiting the scope of the effects of user updates to that part of the database that can be viewed by these users. Furthermore, closed views are especially interesting in an XML setting, where one can publish XML views on the web, which can be updated by others and where security becomes a much more important issue.

In this paper we look at both open and closed view update strategies in the context of XML views and refer to related work in the relational model. We also point out relevant considerations (such as the choice of a view definition and update language), that have to be made in the design of generators for view update strategies and relate them to current tendencies in the design of XML update languages.

This paper is structured as follows. In Section 2 we introduce the notion of XML views and briefly discuss the current status of update languages for XML. In Section 3 we take a look at both closed and open view update strategies and discuss their applicability to an XML setting. Finally, we conclude this paper with a discussion on future work in Section 4.

2 XML Views and Updates

Before we can discuss updatability of XML views, we first have to get a grasp of what XML views are and how updates on XML are performed.

2.1 XML Views

The notion of a view is a very fundamental service to be provided by a database system, however, there is no standard for supporting views on XML databases yet. The first work on XML views was done by Abiteboul in [1], where a possible view architecture is introduced and updates of XML views are discussed for the very first time. At the moment there is not yet a generally accepted view definition language for defining virtual XML documents. Still, in some sense, it is already possible to define virtual XML documents by transforming existing XML documents through a

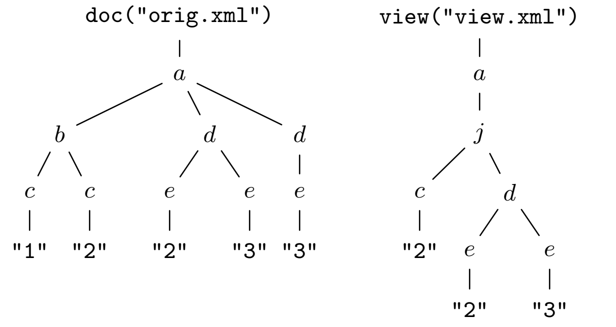

Figure 1: Original document and resulting view from Example 2.1

query/transformation language like XSL or XQuery. In Example 2.1 we illustrate how such a view can be created using an XQuery-like syntax.

Example 2.1 *Suppose we define the virtual document "*`view.xml`*" as follows*[1]*:*

```
declare view { "view.xml" } {
  element {"a"} {
    let $input := doc("orig.xml")
    for $c in $input//c return
      for $c in $input//d
      where $c/text() = $d/*/text()
      return <j> {$c, $d}</j> } }
```

A possible input document "`orig.xml`*" and its result view "*`view.xml`*" are illustrated in Figure 1. Informally, this view performs a join of c's and d's in j elements on the text values of their descendants. The resulting elements are contained within one a element.*

Some problems arise when defining views the way we did in the previous example.

- **Structure of the result view.** A first issue is whether we should include all descendants of the "selected nodes" in the view or not. Including all descendants corresponds to serializing a query result in XQuery, but obviously, it is sometimes desirable to hide some of the descendants of selected nodes. However, showing only the selected nodes raises the problem of how to represent them. If we consider all the selected nodes to be children of the "view node" then we can only get flat views. A straightforward solution is to use the ancestor-descendant relationship of the original XML document to create a parent-child relationship for this view. Nevertheless, if we apply the latter approach then we either need to write our own serialization phase to show only these nodes the user is allowed to see in the result, or we need to construct new nodes for each selected node of the view.
- **Creation of New Nodes.** The creation of new nodes introduces several problems. First of all, it might be possible that, for a number of reasons, users do not want to evaluate node construction each time a query on the view is evaluated. One of these reasons is the presence in XQuery of node identity comparisons and the non-deterministic behavior of node construction w.r.t. the placement in document order of the newly created nodes. For example, a user might want the a node in the result of Example 2.1 to be always the same node. Another issue is the fact that children of newly constructed nodes are always deep copies of the original nodes, which is related to the problem of the structure of the result view. Furthermore, the view can contain several copies of the same node in the original document, but updating a copied node can be complicated. For example, suppose M_1 is a base schema instance M_1 with view N_1 and n is a node in M_1 for which there are two different copies n_1 and n_2 in N_1. If updating node n_1 in N_1 yields N_2, then changing the value of n in M_1 yields another base schema instance M_2 for which the view cannot be N_2 (since n_1 and n_2 have the same value).

[1]We use our own notation here, since there is no standard way of defining views yet.

The previous problems indicate that XQuery still lacks certain features that are needed to create views in an elegant manner. Moreover, the full XQuery language is way too complex to handle as a view definition language, especially for updates through views, but limiting ourselves to XPath may be too restrictive. Obviously, the choice of a view definition language can have a huge influence on the updatability of views. The more complex views that can be generated, the harder it is to find corresponding update strategies. Hence we want to restrict the expressive power of the view definition language. Since there is no standard view definition language yet, we can use the formal framework of LiXQuery [14], in order to pin down the semantics of XML views based on the XQuery syntax. LiXQuery is a succinct sublanguage of XQuery with the same expressive power and a clearly specified formal semantics.

2.2 XML Update Languages

There exist some XML update languages [17, 22, 23], but none of them is a generally accepted standard yet. At the moment, a W3C working group is creating extensions to XQuery to support updates. In the design of such a language, several issues have to be kept in mind:

- *Node Identity.* Not all currently available XML update languages support the conservation of node identifiers through update operations. Hence some update operations, such as moving a node around in a tree, are only expressible up to isomorphism in some languages, yielding the moved nodes to have other node identifiers in the

result than in the original document. Note that it can be problematic to return a different but isomorphic result tree, since the XQuery specifications [6] allow node identifiers to be also used outside the scope of the XQuery processor.
- *Granularity of Operations.* Updates can operate on (leaf) nodes or on entire trees. One possible strategy is to allow entire trees to be inserted, deleted or moved with one operation. Another strategy is that all operations can only operate on nodes. The deletion of a node n can have as an effect that either all descendants of n are also removed or that the children of node n become children of the parent of n.
- *Creation of New Documents.* It is not clear whether the creation of a new document with a given URI is an update or not. It can be considered an update in the sense that the web is our database which we extend by creating a new document.
- *(Non-)Deterministic Behavior.* Node construction in XQuery is in a sense non-deterministic since the resulting nodes can be different each time, which also includes different positions w.r.t. document order. Non-determinism can also be allowed for the insertion of new elements in such a way that the XQuery processor can choose the position where he puts nodes when inserting multiple nodes in complex updates. This is a viable solution to cope with certain ambiguities which can occur in complex updates as is illustrated in Example 2.2

These design choices can have an influence on the assessment of updates on views.

Example 2.2 *Consider following UpdateX [22] expression:*

```
update
  let $x := doc("orig.xml")/a/d[2]
  for $y in (1,2)
    insert <f>{$y}</f> after $x/e
    insert <f>{$y}</f> before $x/e
```

If we take "`orig.xml`*" from Figure 1 as input then this expression adds four children in the second d. It is however not clear in what order they have to be added. If we evaluate this against the implementation of UpdateX in Galax 0.5 [9] then the contents of the second d is "*`<f>2</f><f>2</f><f>1</f><e>3</e><f>1</f>`*".*

Several update languages have already been proposed for XML. In [23] a set of extensions to the XQuery syntax is provided. These extensions include deletions, insertions, renamings and replacements of child nodes, and nested updates in for-expressions. UpdateX [22] is another XQuery-based XML update language and includes FLWUpdate statements for complex updates and snapshot semantics to enforce consistency of these complex updates. Both languages however lack a clear and formal description of its semantics. Finally, the XUpdate language [17] is another candidate language for defining XML updates. It is a pure descriptive language which is designed with references to the definition of XSL Transformations.

3 View Update Strategies

We first give the necessary notations borrowed from [12]. This is followed by an introduction to some existing approaches. Finally we discuss the applicability of these approaches to an XML setting.

3.1 Preliminary Notations

To each database schema $\mathbf{D}$ a set $LDB(\mathbf{D})$ of states of the schema is associated. Hence we can associate to each database mapping $f : \mathbf{D}_1 \rightarrow \mathbf{D}_2$ a function $f : LDB(\mathbf{D}_1) \rightarrow LDB(\mathbf{D}_2)$. If $\gamma : \mathbf{D} \rightarrow \mathbf{V}$ is such a database transformation and a surjective function as well, then we call the pair $\Gamma = (\mathbf{V}, \gamma)$ a view of the schema $\mathbf{D}$. In this setting, $\mathbf{V}$ is called the view schema. Finally, an update strategy is defined as a partial function $\rho : LDB(\mathbf{D}) \times LDB(\mathbf{V}) \rightarrow LDB(\mathbf{D})$ that maps a pair of current base state and new view state to a new base state. Figure 2 illustrates these definitions. It states that if we update the view N_1 to a new view N_2 then the corresponding translated update must return a database state M_2 that has as view also N_2. Since γ has to be surjective, each state of the view schema has to be the view result of γ applied to some state of the base schema. Therefore we know that there exists at least one M_2 such that $\gamma(M_2) = N_2$.

$$\begin{tikzcd}
M_1 \arrow[r, "\text{translated update}"] \arrow[d, "\gamma"'] & \rho(M_1, N_2) \arrow[d, "\gamma"'] \\
\gamma(M_1) = N_1 \arrow[r, "\text{view update}"] & N_2 = \gamma(\rho(M_1, N_2))
\end{tikzcd}$$

Figure 2: Relationship of views and update strategies.

Intuitively, in closed view update strategies all updates of the view can be undone by other updates of the view, i.e., the initial database state of the base schema can be recovered, and updates can be applied sequentially or all at once. More precisely, closed view update strategies have to obey the following rules:

- *Reflexivity.* The identity view update is allowable and corresponds to an identity update on the base schema state: $\forall M \in LDB(\mathbf{D}) : \rho(M, \gamma(M)) = M$
- *Symmetry.* Every view update is globally reversible: $\forall M \in LDB(\mathbf{D}), N \in LDB(\mathbf{V}) : \rho(\rho(M, N), \gamma(M)) = M$
- *Transitivity.* A series of view updates may be applied incrementally or all at once: $\forall M \in$

$LDB(\mathbf{D}), N_1, N_2 \in LDB(\mathbf{V}) : \rho(M, N_2) = \rho(\rho(M, N_1), N_2)$

All these conditions have to hold if the updates that occur in them are allowable. Open view update strategies are more liberal in the sense that they allow updates to have some side effects. In what follows we discuss both closed and open update strategies.

3.2 Existing Approaches in RDBMS and OODBMS

One of the first efforts to systematically address the view update problem have been made by Dayal and Bernstein. They explored the notion of correct translatability of view updates in [8]. In this work they required the view update to have a unique set of corresponding updates on the base schema.

Bancilhon and Spyratos introduced the constant-complement approach [3]. A complement of a view is another view, such that together they form a lossless decomposition, i.e., the state of the base schema can be recovered from the combined states of both views. Every complement Γ_2 of a view Γ_1 defines an update strategy on Γ_1. Furthermore, it is shown in [3] that every closed update strategy is defined by a constant complement update strategy. However, it is known that there exist complements that do not define closed update strategies. Cosmadakis and Papadimitriou show that finding a minimal complement of a given view is NP-complete [7]. Lechtenbörger and Vossen demonstrate how to compute reasonable small complements for relational views that only contain projection, selection, renamings and joins [19]. For some special cases these complements are even shown to be minimal. Hegner presents an order-based approach to find a unique "natural" complement which defines the only reasonable update strategy [13]. He assumes the set $LDB(\mathbf{D})$ to be partially ordered. Many database transformations preserve this order structure. For example, a natural order structure in the relational model is a relation-by-relation inclusion for which the only base operation of the relational algebra that is not monotonic is the difference. Furthermore, the notion of closed update strategies is extended by adding three extra order-related conditions.

Gottlob et al relaxed the requirement of a constant complement to capture the entire class of views where the effect of updates on the base schema is unambiguously determined when the effect of an update on the view schema is determined [10]. This class of "consistent views" is obtained by allowing the complement of a view to decrease according to a suitable partial order. Masunaga presented an approach to design view update translators for relational databases [20]. In his approach, views are presented as trees where the base relations are leaves, the root is the view itself and intermediate nodes represent relational algebra operations and rules are introduced to process updates on the view to the lower levels of the trees. Furthermore, it is suggested that end-users should be involved in resolving semantic ambiguities. Keller developed another translation mechanism of view updates, where he considers select-project-join views on BCNF relations [15] and in order to eliminate anomalies, update translations have to be valid and satisfy five criteria. This method gives a complete list of alternative (acceptable) translations for the considered view updates. Finally, Tomasic showed that the number of translations for a view update is exponential in the size of the database and that this number can be reduced by using annotations [24].

The view update problem has also been studied for object-oriented databases. For example, [21] shows that the view update problem for object-oriented databases is much more feasible than in the relational model. This conclusion is based on the object preserving operator semantics of the query language they use. In this setting, objects in the view are objects of the database, and hence updating the view is directly an update of the objects of the database. However, these techniques are not applicable to an XML setting if one prefers views to have the same look and feel as normal data, since in this case views should behave like trees.

3.3 Existing Work on XML View Updates

In [16] a stack-based approach is used to create XML views that can be updated. This is made possible by including imperative statements in the view definitions, which have to explicitly state the intents of updates. This approach is very liberal, but puts a high responsibility on the shoulders of the person who defines the view. Moreover, their framework does not use any of the current XML standards.

Focal [11] is a language to define bi-directional tree transformations. This language uses the notion of lenses consisting of two partial functions, namely a get function that corresponds to the γ function of a view and the put function which is an update strategy similar to ρ. In order for a lens to be well-behaved they have to obey the reflexivity and the symmetry requirements of a closed view update strategy, but the transitivity requirement is dropped. This work is, however, not directly applicable in an XML setting and joins are not expressible in Focal.

In addition to updating pure XML and relational views, there has also been some work on XML views on relational databases. Braganholo et al study how to use the view update results of the relational model to assess updatability of XML views over relational databases [4]. They define XML views by means of query trees, i.e., templates containing information on the values that have to be included. A relational view can be associated with these query trees, and updates on the XML view are translated to updates on the relational view. Hence the problem of updating XML views on relational data is converted to that of updating relational views.

3.4 Applying Existing Solutions to XML

The work of Hegner in [13] is abstract and general. Applying it to an XML setting is however not straightforward. A first challenge is to define a suitable partial order on the set of possible states of the web. Since the web is an ordered forest, a natural order might be the subforest relation, but whether this is apropriate heavily depends on our update and view definition languages. Moreover, we want to see which operations are monotonic under this order structure. Last but not least, we want to study the complexity of finding an order-based closed update strategy for a number of fragments of the view definition and update language.

The node identity in XML and the semantics of XQuery would suggest a close connection between [21] and the view update problem in XML. Note however that XQuery allows us to create new nodes. Furthermore, the combination of preserving node identity and restructuring the nodes for creating the view, is problematic, as explained in Section 2.

Finally, similar to [15] one might consider only a limited set of views for which the base schema is in some kind of normal form, like XNF [2]. However, which documents are used by the view cannot always be determined at view-definition time[2] and hence we have to assume that all documents on the web conform to this normal form.

4 Discussion

In this paper we pointed out some problems that appear when considering updates of XML views. We discussed the need for an XML view definition language and an update language. In the near future we plan to take the design issues discussed in Section 2 into account while designing a general XML view definition language and an XML update language, both based on our earlier work on LiXQuery [14]. These languages have to be easily dividable into fragments in order to study the effect of expressive power of these languages on assessing the updatability of XML views defined in these fragments. We want to use this framework to investigate order-based closed update strategies, as introduced in [13]. More precisely, we want to see what a good natural order is on XML and which fragments of our view definition language guarantee monotonicity. Furthermore, we want to study the computational complexity of finding order-based closed update strategies for our framework. Finally we want to see whether updatability in our framework subsumes updatability in the relational model when we assume that relations are represented by their canonical view.

[2]Since a "`doc`" function call can be applied to any string, we can, for example, load a string from one document and use it to load another document.

References

[1] S. Abiteboul. On views and XML. In *PODS*, pages 1–9, 1999.

[2] M. Arenas and L. Libkin. A normal form for XML documents. In *PODS*, pages 85–96, 2002.

[3] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM TODS*, 6(4):557–575, 1981.

[4] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. From XML view updates to relational view updates: old solutions to a new problem. In *VLDB*, pages 276–287, 2004.

[5] E. F. Codd. Recent investigations in relational data base systems. In *IFIP Congress*, pages 1017–1021, 1974.

[6] W. W. W. Consortium. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery, 2005.

[7] S. S. Cosmadakis and C. H. Papadimitriou. Updates of relational views. *J. ACM*, 31(4):742–760, 1984.

[8] U. Dayal and P. A. Bernstein. On the updatability of relational views. In *VLDB*, pages 368–377, 1978.

[9] M. Fernández and J. Siméon. *Galax, the XQuery implementation for discriminating hackers.* Lucent Technologies – Bell Labs, v0.5 edition, 2005. `http://www.galaxquery.org`.

[10] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM TODS*, 13(4):486–524, 1988.

[11] M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. A language for bi-directional tree transformations. In *PLAN-X*, 2004.

[12] S. J. Hegner. Foundations of canonical update support for closed database views. In *ICDT*, pages 422–436, 1990.

[13] S. J. Hegner. An order-based theory of updates for closed database views. *AMAI*, 40(1-2):63–125, 2004.

[14] J. Hidders, J. Paredaens, R. Vercammen, and S. Demeyer. A light but formal introduction to xquery. In *XSym*, pages 5–20, 2004.

[15] A. M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *PODS*, pages 154–163, 1985.

[16] H. Kozankiewicz, J. Leszczylowski, and K. Subieta. Updatable XML views. In *ADBIS*, pages 381–399, 2003.

[17] A. Laux and L. Martin. XUpdate — XML Update Language, 2000.

[18] J. Lechtenbörger. The impact of the constant complement approach towards view updating. In *PODS*, pages 49–55, 2003.

[19] J. Lechtenbörger and G. Vossen. On the computation of relational view complements. *ACM TODS*, 28(2):175–208, 2003.

[20] Y. Masunaga. A relational database view update translation mechanism. In *VLDB*, pages 309–320, 1984.

[21] M. H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *DOOD*, pages 189–207, 1991.

[22] G. M. Sur, J. Hammer, and J. Siméon. UpdateX - an XQuery-based language for processing updates in XML. In *PLAN-X*, 2004.

[23] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD Conference*, 2001.

[24] A. Tomasic. View update translation via deduction and annotation. In *ICDT*, pages 338–352, 1988.