A MAPPING-DRIVEN APPROACH FOR SQL/XML VIEW MAINTENANCE

Vânia M. P. Vidal, Fernando C. Lemos, Valdiana S. Araújo
Department of Computing, Federal University of Ceará, Fortaleza/CE, Brazil

Marco A. Casanova
Department of Informatics, PUC-Rio, Rio de Janeiro/RJ, Brazil

Keywords: XML Views, Incremental View Maintenance, Relational Databases.

Abstract: In this work we study the problem of how to incrementally maintain materialized XML views of relational data, based on the semantic mappings that model the relationship between the source and view schemas. The semantic mappings are specified by a set of correspondence assertions, which are simple to understand. The paper focuses on an algorithm to incrementally maintain materialized XML views of relational data.

1 INTRODUCTION

As XML becomes the facto standard for data exchange among applications (over the web), and since most business data is currently stored in relational database systems, the problem of publishing relational data in XML format has special significance. A general and flexible way to publish relational data in XML format is to create XML views of the underlying relational data. The community agrees on a certain schema, and subsequently all members of the community create XML views that conform to the predefined schema. As mention in (Bohannon et al, 2004), this is called *schema-directed XML publishing*.

The contents of views can be materialized to improve query performance and data availability (Dimitrova et al, 2003; Gupta and Mumick, 2000). To be useful, a materialized view needs to be continuously maintained to reflect dynamic source updates. Basically, there are two strategies for materialized view maintenance. *Re-materialization* re-computes view data at pre-established times, whereas *incremental maintenance* periodically modifies part of the view data to reflect updates to the database. It has been shown that incremental maintenance generally outperforms full view recomputation.

In this work we study the problem of how to efficiently maintain XML view of relational data, based on the mappings that model the relationship between the source and view schemas. The schema mappings are specified by a set of correspondence assertions (Popa et al, 2002; Vidal et al, 2006), which defines how to transforms source states to view states. The benefits of using declarative formalisms for schema mappings are well-known (Bernstein and Melnik, 2007; Jiang et al, 2007). We also note that other mapping formalisms are either ambiguous (Miller, 2007) or require the user to declare complex logical mappings (Fuxman et al, 2006; Yu and Popa, 2003), and are not appropriated to support incremental view maintenance. It is important to pointing out that the problem of generating schema mappings is outside the scope of this paper.

The views that we address are focused on schema-directed XML publishing. As such, the correspondence assertions induce schema mappings defined by the class of projection-selection-equijoin (PSE) SQL/XML queries, which support most types of data restructuring that are common in data exchange applications. We make a compromise in constraining the expressiveness of mappings so we can have an algorithm that is much more efficient and views that are self-maintainable.

In this paper, we present an algorithm to incrementally maintain materialized XML views of relational data, in the context of the SQL/XML (Eisenberg et al, 2004) standard. The algorithm has four major steps: first, it identifies the view paths that are relevant to a base update μ; second, it identifies all

M. P. Vidal V., C. Lemos F., S. Araújo V. and A. Casanova M. (2008).
A MAPPING-DRIVEN APPROACH FOR SQL/XML VIEW MAINTENANCE.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - DISI*, pages 65-73
DOI: 10.5220/0001711900650073

elements in a relevant path that are affected by μ; third, it generates the list of updates required to maintain the affected elements; and, finally, it sends the list of updates to the view. We also establish sufficient conditions, based on the correspondence assertions, to prove that a list of updates correctly maintains a view. The results we present in this paper are novel and have never been submitted for publication.

The implementation of the **View_Maintainer** Algorithm is very efficient, since most of the work is done at view definition time. For each type of view update, based on the view correspondence assertions, and at view definition time, we automatically generate: (i) The set of view paths that are relevant to the update (i.e. the view paths that may have affected elements); (ii) The query that computes the set of affected elements in a given relevant path; and (iii) The SQL/XML queries that extracts, from the base source, all information needed for propagating the update to the view.

The main features of the presented approach that distinguish it from the previous related works are as follows:

(i) The mappings are used only at view definition time. So, no mapping compilation is required at view maintenance time.

(ii) The list of updates required to maintain the view are defined based solely on the source update and current source state, that is, they need not access the materialized view.

(iii) The algorithm generates a set of view updates (instead of delta updates). So, no data combination (or merge) is required, and the view updates can be directly applied to the view without accessing the base data source.

Features (ii) and (iii) are very important when the view is stored outside the DBMS, since accessing a remote data source is possibly too slow.

This paper is organized as follows. Section 2 summarizes related work in the area of incremental view maintenance. Section 3 discusses XML Views in the context of SQL/XML. Section 4 presents the **View_Maintainer** algorithm. Finally, Section 5 contains the conclusions.

2 RELATED WORK

The problem of Incremental View Maintenance has been extensively studied for relational view (Ceri and Widom, 1991; Gupta and Mumick, 2000) as well as for object-oriented view (Ali et al, 2000; Kuno and Rundensteiner, 1998). There have been also incremental maintenance algorithms for semi-structured views (Abiteboul et al, 1998; Liefke and Davidson, 2000; Zhuge and Garcia-Molina, 1998) and XML views (Dimitrova et al, 2003; EL-Sayed et al, 2002; Sawires et al, 2005). Different data models and view specification languages have been assumed by a number of researchers. The algorithms in (Abiteboul et al, 1998; Liefke and Davidson, 2000; Zhuge and Garcia-Molina, 1998) are developed for views defined with a query over graph structures. The views considered in (Dimitrova et al, 2003; EL-Sayed et al, 2002) are defined using an XML algebra over XML trees, and the views in (Sawires et al, 2005) are defined using path expressions over XML documents. None of the above techniques can be directly applied to XML views of relational data.

The only work on maintaining XML views over relational schema that we are aware of is (Bohannon et al, 2004). The incremental algorithm in (Bohannon et al, 2004) maintains XML documents produced by an ATG, a formalism for mapping a relational schema to a predefined (possibly recursive) DTD. In their approach, a middleware system interacts with the underlying DBMS and maintains a hash index and a subtree pool for the external XML view. The main problem with this approach, not to mention the high complexity of the algorithm, is that it requires several round-trips between the middleware and the DBMS. Therefore, the view is not self maintainable, which is a desirable feature for external views (view stored outside the DBMS). Other draw backs are that the use of in-memory hash table limits the technique for large documents cached in a middleware, and it is not possible to detect irrelevant updates.

3 XML VIEWS

With the introduction of the XML datatype and the SQL/XML standard, users may create a view of XML type instances over relational tables using SQL/XML publishing functions, such as XMLElement(), XMLAgg(), etc. In this section, we propose to specify an XML view with the help of a set of correspondence assertions, which axiomatically specify how the XML view elements are synthesized from tuples of the base source.

Definition 1. Let S be a base relational schema. An *XML view*, or simply, a *view* over S is a quadruple $V = \langle e, T_e, \Psi, \mathcal{A} \rangle$, where:

(i) e is the name of the primary element of the view;

(ii) T_e is the XML type of element e, which must be a restricted complex type (T_e is defined using the *complexType* and *sequence* constructors

only, and the type of its attributes is an XML simple type).

(iii) ψ is a global correspondence assertion (GCA); A *global correspondence assertion* (GCA) is an expression of form: [**V**] ≡ [R_p[selExp]], where R_p is a relation scheme of S, and selExp is a predicate expression.

(iv) $\mathcal{A}$ is a set of *path correspondence assertions* (PCA) that specifies T_e in terms of R_p (Vidal et al, 2006).

We also say that the pair <e, T_e> is the view schema of **V** and R_p is the pivot relation scheme of the view. □

Let S be a relational schema and **V** = <e, T_e,Ψ, $\mathcal{A}$> be an XML view over S. Given a state σ_S of S, let $\sigma_S(R_p)$ denote the relation that σ_S associates with R_p. As shown in (Vidal et al, 2006), $\mathcal{A}$ defines a constructor function, denoted $\tau[\mathcal{A}]$, from tuples of $\sigma_S(R_p)$ to instances of T_e.

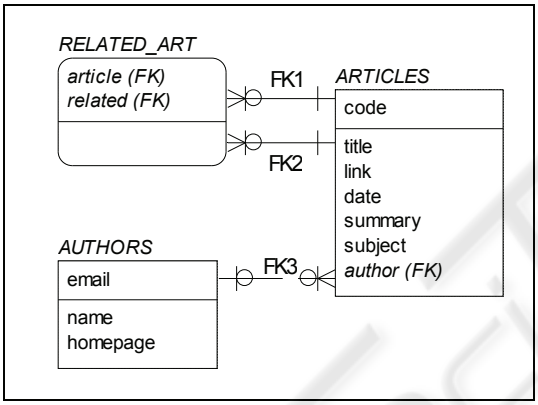

Figure 1: Relational schema ArticlesDB.

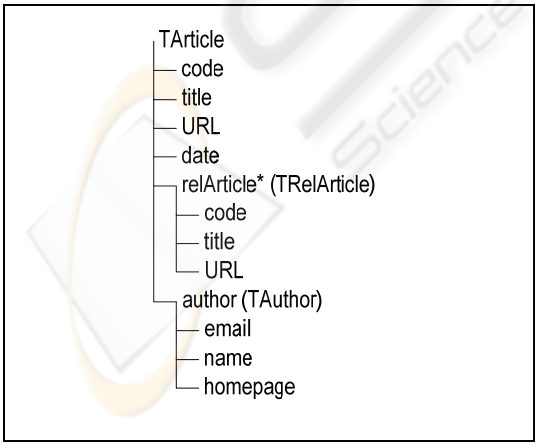

Figure 2: XML type TArticle.

Moreover, we say that an instance **$t** of T_e is semantically equivalent to a tuple **r** of $\sigma_S(R_p)$ (**$t** $\equiv_{\mathcal{A}}$ **r**) iff $\tau[\mathcal{A}]$(**r**) = **$t**. The state of **V** on σ_S is an XML document σ_v whose root element, denoted root[σ_v], contains a set **E** of <e> elements of type T_e and is defined as

E = { **$t** | **$t** is an <e> element of type T_e and there is **r**∈$\sigma_S(R_p)$ such that **r** satisfies selExp and **$t**=$\tau[\mathcal{A}]$(**r**) }.

The functional mapping defined by the correspondence assertions can be correctly translated to an SQL/XML query view definition. For example, consider the relational schema ArticlesDB in Figure 1. Suppose the XML view **Articles_XML**, whose schema is shown if Figure 2. The root element of view **Articles_XML**, contains multiple occurrences of the element <Article>, with type TArticle. The GCA of view **Articles_XML** is given by:

ψ :[**Articles_XML**] ≡ [*ARTICLES*[subject = sport]].

Figure 3 shows $\mathcal{A}$[**Articles_XML**], the path correspondence assertions which specify TArticle in terms of ARTICLES. The correspondence assertions of **Articles_XML** are generated by: (1) matching the elements and attributes of TArticle with attributes or paths of ARTICLES; and (2) recursively descending into sub-elements of TArticle to define their correspondence assertions. The problem of generating the correspondences is outside the scope of this paper.

Given a state σ of ArticlesDB, the root element of **Articles_XML** contains a set **A** of element <Article>, with type TArticle, defined as follows:

A = { **$a** | **$a** is an instance of $T_{Article}$ and
∃**r**∈ σ(ARTICLES), where
r.subject ='sport' and **$a** $\equiv_{\mathcal{A}[\text{Articles\_XML}]}$ **r** }.

Figure 4 shows an SQL/XML implementation of the constructor function $\tau[\mathcal{A}$[**Articles_XML**]]. For each tuple in table ARTICLES, the SQL/XML query uses the SQL/XML standard publishing functions to construct an instance of the XML type TArticle. The constructor function creates an instance **$a** of TArticle from a tuple **a** of ARTICLES such that **$a** is semantically equivalent to **a**, as specified by the assertions of **Articles_XML**. The constructor function contains four sub-queries, one for each element and attribute of TArticle. Each subquery is generated from the correspondence assertion of the corresponding element or attribute. Figure 4 also shows the assertion that generates each SQL/XML subqueries.

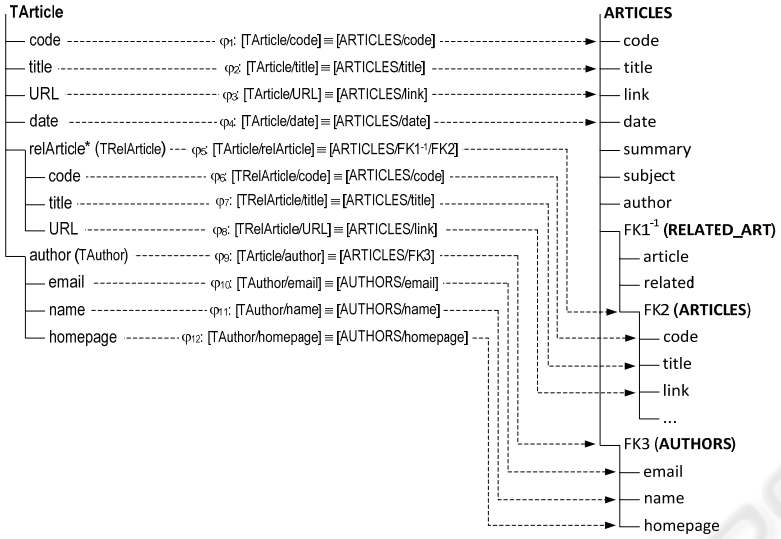

Figure 3: Correspondence Assertions of **Articles_XML** view.

```
XMLELEMENT("article",
  XMLFOREST(a.code AS "code"),
  XMLFOREST(a.title AS "title"),
  XMLFOREST(a.link AS "URL "),
  XMLFOREST(a.date AS "date"),
  (SELECT XMLELEMENT("relArticle",
     XMLFOREST(a2.code AS "code"),
     XMLFOREST(a2.title AS "title"),
     XMLFOREST(a2.link AS "URL"))
    FROM RELATED_ART r, ARTICLES a2
    WHERE r.article = a.code AND r.related = a2.code),
  (SELECT XMLELEMENT("author",
     XMLFOREST(u.email AS "email"),
     XMLFOREST(u.name AS "name"),
     XMLFOREST(u.homepage AS "homepage") )
   FROM AUTHORS u WHERE u.email = a.author) )
```

Figure 4: SQL/XML implementation of the constructor function $\tau[\mathcal{A}[\textbf{Articles\_XML}]](\mathbf{a})$.

4 INCREMENTAL VIEW MAINTENANCE

In this section, let S be a relational schema and **V** = <e_0, T_{e_0}, Ψ, $\mathcal{A}$> be a view over S, where [**V**] ≡ [$\mathbf{R}_0$[selExp]] is the GCA of **V**. We first explain the intuition behind our approach for incremental view maintenance. Then, we address the use of the view correspondence assertions to identify the view paths that are relevant to a base update μ. Finally, we present an algorithm for the incremental maintenance of V.

4.1 Our Approach

In following, we introduced the concept of view path and then we explain the intuition behind our approach.

Definition 2. Let $T_{e_1},...,T_{e_n}$ be restricted XML Schema types defined in the XML Schema of T_{e_0}. Suppose that T_{e_k} contains a property (attribute or element) e_{k+1} of type $T_{e_{k+1}}$, for k=0,...,n-1. Then, we say that:

(i) $e_1 / e_2 /.../ e_n$ is a *path of* T_{e_0}; and

(ii) $e_0 / e_1/.../ e_n$ is a *path of* **V**. □

To illustrate, consider the view **Articles_XML** in Figure 2. article/relArticles and article/relArticles/URL are examples of paths of **Articles_XML**.

In our approach, incremental view maintenance is done using the following steps:

1. Identifies the view paths that are relevant to a base update μ;
2. Identifies all elements in a relevant path that are affected by μ;
3. Generates the list of view updates required to maintain the affected elements.
4. Sends the list of updates to the view.

Formal definitions of *relevant path* and *affected element* are given in Section 4.2. An example is given below.

Example 1. Consider the view **Articles_XML** in Figure 2. Let

μ_1 = `UPDATE ARTICLES SET link = 'nyt.com/get?code=A6B1' WHERE code = 'A6B1'`

Suppose that the current state of the data source ArticlesDB is the one shown in Figure 5. Figure 6(a) shows the corresponding state of view **Articles_XML**. As indicated in Fig. 6 (a), μ_1 affects the content of the URL element of the article element **A_1** in doc("Article.xml")/article, and the content of the relArticle element **A_2** in doc("Article.xml")/article/relArticle. So the paths δ_1 = article/URL and δ_2 = article/relArticles/URL are relevant to μ_1.

The view updates required to maintain paths δ_1 and δ_2 are, respectively,

(i) Replace the URL element of **A_1** by

<URL>nyt.com/get?code=A6B1</URL>

(ii) Replace the URL element of **A_2** by

<URL>nyt.com/get?code=A6B1</URL>

The new state of view **Articles_XML**, after the updates, is shown in Figure 6(b).

ARTICLES

CODE	TITLE	LINK	DATE	SUMMARY	SUBJECT	AUTHOR
A6A5	The Bracket	nytimes.com/article?code=A6A5	01/08/2007	If you picked the …	sports	marcus@nytimes.com
A6B1	Beware of The Tigers	nytimes.com/article?code=A6B1	02/08/2007	Along the time...	sports	marcus@nytimes.com
A6B2	Watch Your Mouth	nytimes.com/article?code=A6B2	03/08/2007	Since the Heysel...	sports	marcus@nytimes.com
G6JL	More Mistakes	nytimes.com/article?code=G6JL	18/09/2007	The afternoon...	arts	shpigel@nytimes.com

RELATED_ART

ARTICLE	RELATED
A6B2	A6B1
A6B2	A6A5

AUTHORS

EMAIL	NAME	HOMEPAGE
marcus@nytimes.com	Jeffrey Marcus	http://www.nytimes.com/marcus
dargis@nytimes.com	Ben Shpigel	http://www.nytimes.com/shpigel

Figure 5: An instance of ArticlesDB.

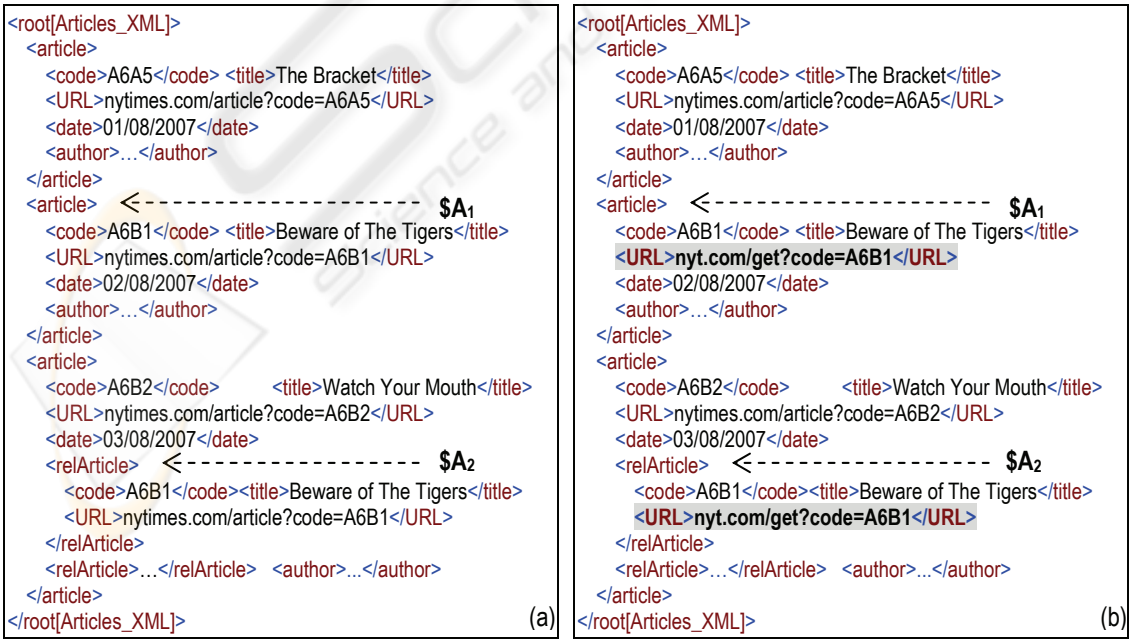

Figure 6: (a) An instance of **Articles_XML** view; (b) Instance of **Articles_XML** view after the updates.

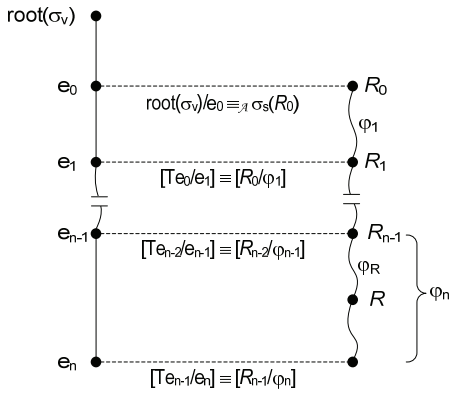

Figure 7: Path δv.

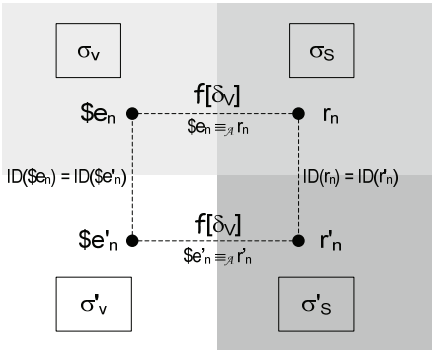

Figure 8: Mapping function f[δV].

4.2 Identifying Relevant Paths

First, we define the updates for which the path $\delta_v = e_0$ is relevant, and then for the other types of view path.

Definition 3. Let μ be a base update. The path $\delta_v = e_0$ is relevant to μ iff μ is one of the following operations: (i) insertion in $\mathbf{R}_0$; (ii) deletion from $\mathbf{R}_0$; (iii) update on attribute **a** of $\mathbf{R}_0$, where **a** is referenced in selExp. □

In the rest of this section, let:

- μ be an update over base source S;
- σ_S and σ'_S be the states of S before and after μ, respectively;
- σ_V and σ'_V be the states of **V** in σ_S and σ'_S, respectively;
- $\delta_V = e_0 /.../ e_n$, n>0, be a path of **V.**

Let $[T_{e_i} / e_{i+1}] \equiv [R_i / \varphi_{i+1}]$ be the path correspondence assertions of e_{i+1} in $\mathcal{A}$, for $0 \leq i \leq n-1$ (see Figure 7). We say that the path $e_1/.../ e_n$ of T_{e_0} *matches the path* $\varphi_1 / ... / \varphi_n$ of R_0 $(e_1/.../ e_n \equiv_{\mathcal{A}} \varphi_1 /... /\varphi_n)$.

Definition 4. Let $K=\{k_1,.., k_m\}$ be the primary key of R_n, and $[T_{e_n}/a_i] \equiv [R_n/k_i]$ be in $\mathcal{A}$ (which exists by assumption on $\mathcal{A}$), for $1 \leq i \leq m$. Given an element $\mathbf{\$e}_n$ in $\mathrm{root}(\sigma_V)/\delta_V$, the *mapping function of* δ_V, denoted by $\mathbf{f}[\delta_V]$, maps $\mathbf{\$e}_n$ into a tuple $\mathbf{r}_n$ in $\sigma_S(R_n)$ such that $k_i = \$e_n/a_i$, for $1 \leq i \leq m$. In this case, we say that $\mathbf{\$e}_n$ *matches* $\mathbf{r}_n$. □

For the purpose of our proof, we assume that each tuple in a relational table has a unique, immutable identifier. We also assume that each non-leaf element in an XML document has a unique, immutable identifier. Given a tuple (or element) **t**, let $\mathtt{ID}(\mathbf{t})$ returns the identifier of **t**. We stress that these assumptions are necessary only to establish our formal results, and the identifiers are not required by the **View_Maintainer** Algorithm.

From the definition of **V**, we can prove that, given $\mathbf{\$e}_n \in \mathrm{root}(\sigma_V)/\delta_V$, where $\mathbf{f}[\delta_V](\mathbf{\$e}_n) = \mathbf{r}_n$, then: (i) $\mathbf{\$e}_n \equiv_{\mathcal{A}} \mathbf{r}_n$; and (ii) if there is $\mathbf{\$e'}_n \in \mathrm{root}(\sigma'_V)/\delta_V$, where $\mathtt{ID}(\mathbf{\$e'}_n) = \mathtt{ID}(\mathbf{\$e}_n)$, then $\mathbf{f}[\delta_V](\mathbf{\$e'}_n) = \mathbf{r'}_n$, where $\mathtt{ID}(\mathbf{r'}_n) = \mathtt{ID}(\mathbf{r}_n)$ (see Figure 8).

Definition 5. Let

- σ_S and σ'_S be the states of S before and after μ, respectively;
- $\mathbf{r}_{n-1}$ be a tuple in $\sigma_S(R_{n-1})$
- $\mathbf{r'}_{n-1}$ be a tuple in $\sigma'_S(R_{n-1})$ where $\mathtt{ID}(\mathbf{r'}_{n-1}) = \mathtt{ID}(\mathbf{r}_{n-1})$
- $\mathtt{I}[\mu, \mathbf{r}_{n-1}/\varphi_n]$ be the set of tuples inserted in $\mathbf{r}_{n-1}/\varphi_n$ by μ
- $\mathtt{D}[\mu, \mathbf{r}_{n-1}/\varphi_n]$ be the set of tuples deleted from $\mathbf{r}_{n-1}/\varphi_n$ by μ.

(i) We say that path φ_n of r'_{n-1} *is affected by* μ iff

- if φ_n has simple type then $\mathbf{r}_{n-1}/\varphi_n \neq \mathbf{r'}_{n-1}/\varphi_n$
- if φ_n has a complex type then $\mathtt{I}[\mu, \mathbf{r}_{n-1}/\varphi_n] \neq \emptyset$ or $\mathtt{D}[\mu, \mathbf{r}_{n-1}/\varphi_n] \neq \emptyset$.

(ii) Let σ_V and σ'_V be the value of **V** in σ_S and σ'_S, respectively. Let $\$e_{n-1}$ be an element in $\mathrm{root}(\sigma_V)/ e_0/ e_1/.../e_{n-1}$ where $\mathbf{f}[e_0/.../e_{n-1}](\$e_{n-1}) = r_{n-1}$. We say that property e_n of $\$e_{n-1}$ *is affected by* μ, iff path φ_n of r_{n-1} *is affected by* μ. □

Note that, if the value of path φ_n of a tuple $\mathbf{r}_{n-1}$ in $\sigma_S(R_{n-1})$ is affected by μ, then the value of property e_n of the element $\mathbf{\$e}_{n-1}$ in $\mathrm{root}(\sigma_V)/ e_0/ e_1/.../e_{n-1}$, where $\mathbf{f}[e_0/.../e_{n-1}](\mathbf{\$e}_{n-1}) = \mathbf{r}_{n-1}$, is also affected by μ.

Definition 6. Let σ_S and σ'_S be the states of S before and after μ, respectively. $\mathtt{A}[\mu,\delta_V](\sigma'_S)$ returns the set of all tuples $\mathbf{r'}_{n-1}$ in $\sigma'_S(R_{n-1})$ such that the path φ_n of $\mathbf{r'}_{n-1}$ is affected by μ. □

Definition 7. δ_V *is relevant to* μ iff there exists a state σ_S of S such that $\mathtt{A}[\mu,\delta_V](\sigma'_S) \neq \emptyset$. □

From Definition 7, we have that the path δ_v is relevant to μ iff there exists a state σ_S of S, and there is a tuple **r** in $\sigma_S(R_{n-1})$ such that the value of path φ_n of **r** is affected by μ. In this case, the value of the property e_n of an element $\mathbf{\$e}_{n-1}$ in the path $root(\sigma_v)/ e_0/ e_1/.../e_{n-1}$, where $\mathbf{\$e}_{n-1}$ matches an affected tuple, is also affected by μ.

The following theorems establish sufficient conditions to detect when a path $\delta_v = e_0 / e_1/.../ e_n$, where n >0, is relevant to an update μ.

Theorem 1. Let μ be an insertion or deletion operation on *R*. Then, δ_v is relevant to μ iff $\varphi_n = \varphi_1 . FK^{-1} . \varphi_2$, where φ_1 and φ_2 can be null and FK is a foreign key of *R*. □

Theorem 2. Let μ be an update operation on an attribute *a* of *R*. Then, δ_v is relevant to μ iff φ_n satisfies one of the following conditions:
Case 1: $R_{n-1} = R$ and $\varphi_n = a$.
Case 2: $R_{n-1} = R$ and $\varphi_n = \{a_1,...,a_n\}$ and $a \in \{a_1,...,a_n\}$.
Case 3: $\varphi_n = \varphi . \wr . a$, where φ can be null and $\wr$ is a foreign key that references *R* or $\wr$ is the inverse of a foreign key of *R*.
Case 4: $\varphi_n = \varphi . \wr . \{a_1,...,a_n\}$, where φ can be null, $\wr$ is a foreign key that references *R* or $\wr$ is an inverse of a foreign key of *R*, and $a \in \{a_1,...,a_n\}$.
Case 5: $\varphi_n = \varphi_1 . \wr . \varphi_2$, where φ_1 and φ_2 can be null, $\wr$ is a foreign key of *R* or $\wr$ is an inverse of a foreign key of *R*, and *a* is an attribute of $\wr$. □

To illustrate, consider the example below.

Example 2. Consider the update μ_1 of Example 1. From the set $\mathcal{A}$ of path correspondence assertions of view **Articles_XML** (see Figure 3), we have that:
(i) Since URL $\equiv_{\mathcal{A}}$ link, and the value of link for the updated tuple in ARTICLES is affected by μ, then, from Definition 7, we have that the view path article/URL is relevant to μ. (This follows from Case 1 of Theorem 2).
(ii) Since relArticles/URL $\equiv_{\mathcal{A}}$ FK1^{-1}/FK2/link, and the value of link for the updated tuple in ARTICLES is affected by μ, then, from Definition 7, we have that the view path article/relArticles/URL is relevant to μ. (This follows from Case 3 of Theorem 2).

4.3 The View_Maintainer Algorithm

Figure 9 shows the **View_Maintainer** Algorithm. Given an update to μ over base source S, the algorithm generates, for each path δ_v that is relevant to μ, the list of updates U required to maintain δ_v w.r.t. μ, and then it sends the list of updates U to the view. The set of all paths of **V** that are relevant to μ, denoted by $\mathbb{P}[\mu,\mathbf{V}]$, is automatic and efficiently computed, at view definition time, using theorems 1 and 2.

In case that $\delta_v = e_o$ (cases 1-3 of the VM algorithm), then μ is an insertion, deletion or update over the pivot relation R_0 (see Definition 3). In case that μ is an insertion, if the inserted tuple r_{new} satisfy the select condition of the view's global assertion, then the view updates U consists of an insertion of an element $\$e_0$ in doc("V.xml") where $\$e_0 \equiv_{\mathcal{A}} r_{new}$. The view updates are expressed using the XQuery Update Facility (W3C, 2007). In case that μ is a deletion, if the deleted tuple r_{old} satisfy the select condition of the view's global assertion, then the view updates U consists of a deletion of the element $\$e_0$ in doc("V.xml")/e_0 where $\mathbf{f}[\delta_V](\$e_0) = r_{old}$.

In case that $\delta_v = e_0 / e_1/.../ e_n$, where n>0, (Case 4 of the VM algorithm), the algorithm first computes the set T which contains the tuples in R_{n-1} such that the path φ_n is affected by μ. The view updates U consists of replacing the value of property e_n for each element $\mathbf{\$e}_{n-1}$ in doc("V.xml")/e_0/ $e_1/.../e_{n-1}$ such that $\mathbf{\$e}_{n-1}$ matches an affected tuple in T.

The queries $\mathbf{Q}[e_0]$ (lines 5 and 12 of the **VM** algorithm) and $\mathbf{Q}[\delta_V]$ (line 19 of the **VM** algorithm), whose definitions are given bellow, are defined at view definition time, using the view correspondence assertions.

In the following definitions, let σ_S be the current state of S.

Definition 8. $\mathbf{Q}[e_0]$ is a parameterized SQL/XML query such that given a tuple **r** in $\sigma_S(R_0)$, $\mathbf{Q}[e_0](\mathbf{r}) \equiv_{\mathcal{A}} \mathbf{r}$. □

For example, for the view **Articles_XML** (see Figure 2), $\mathbf{Q}$[article] is shown in Figure 4.

Definition 9. Let $\delta_v = e_o /.../ e_n$, n>0, be a path of **V** which *matches the path* $\varphi_1 / ... /\varphi_n$ of R_0 ($e_1/.../ e_n \equiv_{\mathcal{A}} \varphi_1 / ... /\varphi_n$) (see Figure 7). $\mathbf{Q}[\delta_V]$ is a parameterized SQL/XML query such that given a tuple **r** in $\sigma_S(R_{n-1})$, $\mathbf{Q}[\delta_V](\mathbf{r}) \equiv_{\mathcal{A}} \mathbf{r}/\varphi_n$.. □

In (Vidal et al, 2006), is presented an algorithm that automatically generates $\mathbf{Q}[e_0]$ and $\mathbf{Q}[\delta_V]$ from $\mathcal{A}$. In following, we present an example for each type of update operation. In those examples, suppose that the current state of the data source ArticlesDB is the one shown in Figure 5.

Example 3. Consider the update μ_1 in Example 1.
(i) Relevant Paths: δ_1 = article/URL and δ_2 = article/relArticles/URL (see example 2).
(ii) Updates for relevant path δ_1: From Case 4 of the VM algorithm we have:

Input: a view **V**, a base update μ on table R and the current state σ_s of S

```
1.  U := ∅;
2.  For each δV in P[μ,V] do
3.      Case 1: δV = e0 and μ is an insertion operation
4.          If selExp(rnew) = true then    /* rnew is the inserted tuple*/
5.              Let $e0 := Q[e0](rnew);    /* See Definition 8 */
6.              U := U ∪ { let $e := doc("V.xml") do insert $e0 into $e }
7.      Case 2: δV = e0 and μ is a deletion operation
8.          If selExp(rold) = true then    /* rold is the deleted tuple*/
9.          U := U ∪ { let $e := doc("V.xml")/e0 [a1 = rold.k1, ..., am = rold.km] do delete $e }
                /* {k1,.., km} is the primary key of R0, and [Te0/ai] ≡ [R0/ki] is the PCA for ai in A,
for 1≤ i ≤m. */
10.     Case 3: δV = e0 and μ is an update operation
11.         Case 3.1:  selExp(rnew) = true and selExp(rold) = false
12.             Let $e0 := Q[e0](rnew);    /* See Definition 8 */
13.             U := U ∪ { let $e := doc("V.xml") do insert $e0 into $e }
14.         Case 3.2:  selExp(rnew) = false and selExp(rold) = true
15.         U := U ∪ { let $e := doc("V.xml")/e0 [a1 = rold.k1, ..., am = rold.km] do delete $e }
                /* {k1,.., km} is the primary key of R0, and [Te0/ai] ≡ [R0/ki] is the PCA for ai in A,
for 1≤ i ≤m. */
16.     Case 4: δV = e0 /…/ en, where n>0, [Tei /ei+1] ≡ [Ri / φi+1] is the CA of ei+1 in A, for 0≤ i ≤n-
1;
17.         Let T := A[μ,δV](σ's);    /* T is the set of affected tuples. See Definition 6 */
18.         For each r in T do
19.             Let I := Q[δV](r);    /* See Definition 9 */
20.             U := U ∪ { let $en-1 := doc("V.xml")/ e0 /…/ en-1 [a1 = r.k1, ..., am = r.km]
                        for $en in $en-1/en do delete $en   for $en in I do insert $en
into $en-1 };
                /* {k1,.., km} is the primary key of Rn-1, and [Ten-1/ai] ≡ [Rn-1/ki] is the PCA for ai
in A, for 1≤ i ≤m. */
21.  ApplyUpdates( V, U);
```

Figure 9: **View_Maintainer** Algorithm.

Affected Tuples (in table ARTICLES): T = { r_{new} }.
For **r** = r_{new}, we have:
U_1 = { **let** $a := doc("Article.xml")/article[code = **A6B1**]
for $u **in** $a/URL **do delete** $u,
for $u **in** I **do insert** $u **into** $a }, where
I = <URL>nyt.com/get?code=A6B1</URL>

(iii) Updates for relevant path δ_2: From Case 4 of the algorithm, we have:

Affected Tuples (in table ARTICLES): T = { r_{new} }.
For **r** = r_{new}, we have:
U_2={**let** $a:=doc("Article.xml")/article/relArticle[code=**A6B1**]
for $u **in** $a/URL **do delete** $u,
for $u **in** I **do insert** $u **into** $a }, where
I = <URL>nyt.com/get?code=A6B1</URL>

(iii) The new state of view **Articles_XML**, after applying updates U_1 and U_2, is shown in Figure 6(b).

Example 4. Consider the update

```
μ2 = INSERT INTO ARTICLES VALUES (
    'A9B6', 'So Much Soccer',
    'nytimes.com/get?code=A9B6',
    '12/09/2007', 'Soccer fans,…',
    'sports',marcus@nytimes.com').
```

(i) Relevant paths: δ_3 = article. (From Definition 3)

(ii) Updates for relevant path δ_3: From Case 1 of the algorithm, since r_{new}.subject = "sports", we have:

U_3 = { **let** $a := doc("Article.xml")
do insert $article **into** $a }, where,
$article = Q[article]($r_{new}$) =

```
<article>
  <code>'A9B6'</code>
  <title>'So Much Soccer'</title>
  <link>'nytimes.com/get?code=A9B6'</link>
  <date>'12/09/2007'</date>
  <author>…</author>
</article>.
```

Example 5. Consider the update

μ_3 = `DELETE FROM RELATED_ART WHERE ARTICLE = 'A6B2' AND RELATED = 'A6B1'.`

(i) Relevant paths: δ_4 = article/relArticle.

(ii) Updates for relevant path δ_4: From Case 4 of the algorithm, we have:

Affected Tuples (in table ARTICLES):

T = { < A6B2, ..., marcus@nyt.com> }.

For affected tuple <A6B2, ..., marcus@nyt.com>, we have:

U_4 = { **let** $a := doc("Article.xml")/article[code = A6B2]
for $u **in** $a/relArticle **do delete** $u,
for $u **in** I **do insert** $u **into** $a }, where

I = { <relArticle>
<code>A6A5</code>
<title>The Bracket</title>
<URL>nytimes.com/article?code=A6A5</URL>
</relArticle>}.

5 CONCLUSIONS

We first introduced the concept of view path and showed how to analyze the correspondence assertions to identify which view nodes in a view path are affected by a base update. Then, we presented the **View_Maintainer** Algorithm and we proved that the algorithm correctly maintains a view. We also established sufficient conditions, based on correspondence assertions, to prove that a list of updates correctly maintains a view.

The effectiveness of the **View_Maintainer** Algorithm is guaranteed for externally maintained view since: (i) View updates are defined based solely on the source update and current source state. Hence, no access to the materialized view or other data source is required. This is important, because accessing a remote data source may be too slow. (ii) The updates are applied to the view without accessing any data source. Therefore, the view **V** is self-maintainable. (iii) The implementation of the **View_Maintainer** Algorithm is very efficient, since most of the work is done at view definition time.

REFERENCES

Abiteboul, S., McHugh, J., Rys, M., Vassalos, V., Wiener, J. L., 1998. Incremental Maintenance for Materialized Views over Semistructured Data. In *VLDB*, pp. 38–49.

Ali, M. A., Fernandes, A. A., Paton, N. W., 2000. Incremental Maintenance for Materialized OQL Views. In *DOLAP*, pp. 41–48.

Bernstein, P. A. and Melnik, S., 2007. Model Management 2.0: Manipulating Richer Mappings. In *SIGMOD*, pp. 1-12.

Bohannon, P., Choi, B., Fan, W., 2004. Incremental evaluation of schema-directed XML publishing. In *SIGMOD*, pp. 13-18.

Ceri, S. and Widom, J., 1991. Deriving productions rules for incremental view maintenance. In *VLDB*, pp. 577–589.

Dimitrova, K., El-Sayed, M., Rundensteiner, E. A., 2003. Order-sensitive View Maintenance of Materialized XQuery Views. In *ER*, pp. 144–157.

Eisenberg, A., Melton, J., Kulkarni, K., Michels, J.E. and Zemke, F., 2004. SQL:2003 has been published. In *SIGMOD*, vol. 33, no. 1, pp. 119–126.

EL-Sayed, M., Wang, L., Ding, L., Rudensteiner, E., 2002. An algebraic approach for Incremental Maintenance of Materialized Xquery Views. In *WIDM*, pp. 88–91.

Fuxman, A., Hernandez, M. A., Ho, H., Miller, R. J., Papotti, P., Popa, L., 2006. Nested mappings: schema mapping reloaded. In *VLDB*, pp. 67–78.

Gupta, A. and Mumick, I.S., 2000. Materialized Views. *MIT Press*.

Jiang, H., HO, H., Popa, L., Han, W., 2007. Mapping-Driven XML Transformation. In *WWW*, pp. 1063–1072.

Kuno, H. A. and Rundensteiner, E. A., 1998. Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation. In *IEEE Transaction on Data and Knowledge Engineering*, vol. 10, no. 5, pp. 768–792.

Liefke, H. and Davidson, S. B., 2000. View Maintenance for Hierarchical Semistructured Data. In *DaWaK*, pp. 114–125.

Miller, R. J., 2007. Retrospective on Clio: Schema Mapping and Data Exchange in Practice. In *International Workshop on Description Logics*.

Popa, L., Velegrakis, Y., Miller, R. J., Hernandez, M. A., Fagin, R., 2002. Translating Web Data. In *VLDB*, pp. 598–609.

Sawires, A., Tatemura, J., Po, O., Agrawal, D., Candan, K., 2005. Incremental Maintenance of Path-expression Views. In *SIGMOD*, pp. 443–454.

Vidal, V. M. P., Casanova, M. A., Lemos, F. C., 2006. Automatic Generation of SQL/XML Views. In: *SBBD*, pp. 221-235.

W3C XML Query Update Facility, 2007. http://www.w3.org/TR/xqupdate. Visited: 12/12/2007.

Yu, C. and Popa, L., 2003. Constraint-Based XML Query Rewriting For Data Integration. In *SIGMOD*, pp. 371–382.

Zhuge, Y. and Garcia-Molina, H., 1998. Graph Structured Views and their Incremental Maintenance. In *ICDE*, pp. 116–125.