Classifying XML Materialized Views for their Maintenance on Distributed Web Sources

Tuyêt-Trâm Dang-Ngoc, Virginie Sans, Dominique Laurent

LICP Laboratory, University of Cergy-Pontoise
2, avenue Adolphe Chauvin
95302 Cergy-Pontoise Cedex. France
{*Prenom.Nom*}@dept-info.u-cergy.fr

Résumé. Ces dernières années ont mis en évidence la croissance et la grande diversité des informations électroniques accessibles sur le web. C'est ainsi que des systèmes d'intégration de données tels que des médiateurs ont été conçus pour intégrer ces données distribuées et hétérogènes dans une vue uniforme. Pour faciliter l'intégration des données à travers différents systèmes, XML a été adopté comme format standard pour échanger des informations. XQuery est un langage d'interrogation pour XML qui s'est imposé pour les systèmes basés sur XML. Ainsi XQuery est employé sur des systèmes de médiation pour concevoir des vues définies sur plusieurs sources. Pour optimiser l'évaluation de requêtes, les vues sont matérialisées. La difficulté est de maintenir incrémentalement des vues matérialisées lors de la mise à jour des sources, car dans le contexte de sources web, très peu d'informations sont fournies par les sources. Les méthodes habituellement proposées ne peuvent pas être appliquées. Cet article étudie comment mettre à jour des vues matérialisées XML sur des sources web, au sein d'une architecture de médiation.

1 Introduction

In industry as well as in research, data information systems were often built separately. This results in highly heterogeneous and distributed systems, which generates fragmented views of data and information. In 1992, Wiederhold proposed a mediation architecture (Wiederhold 1992) for combining information from multiple data sources. The goal of a mediation architecture is to integrate the sources in order to present a uniform view to the final user or application. Generally speaking, a mediation architecture is composed of mediators and wrappers. A wrapper is associated to each source with the roles of extracting and transmitting information from the sources to the integrator. Before transmission, the data are transformed into a predefined format used by the mediator. The mediator integrates information provided by the wrappers and presents a queryable uniform view to the user application.

To facilitate the integration of data across different systems, XML has been adopted as the standard format for information exchange and XQuery, a standard query language for querying XML, has become a major requirement for XML-based systems. Recently, mediation architectures based on XML and XQuery have been proposed and implemented (Dang-Ngoc and Gardarin 2003) (Draper *et al.* 2001). We also note

that previous works as (Gupta *et al.* 1999) use other formats for representing semi-structured data (eg. OEM) for querying these data (eg. Lorel, XML-QL) with a similar approach.

In such applications, the use of views is crucial. Indeed, similarly to the case of relational databases, an XML view can be seen as a collection defined by an XQuery request. XQuery is used to filter data and integrate them to present the different data sources as a single source to the user.

View materialization consists in processing the view in a local cache for better response delay during execution time. However, maintaining materialized views when the source data are updated is not an easy task in general. This problem has been studied in (Laurent *et al.* 2001) for data warehouses. In the context of XML views, the additional following points have to be taken into account: (a) In a distributed environment information exchanges must be optimized. (b) In a web context, the sources are autonomous and thus do not notify their changes, and do not provide internal properties (such as oids).

When a data source is modified, materialized XML views defined on this data source have to be updated accordingly in order to remain consistent. As noticed in (Abiteboul *et al.* 1998) it is not reasonable to recompute the whole view at each source update, but it is better to use incremental techniques as (Abiteboul *et al.* 1998) (El-Sayed *et al.* 2002). (Abiteboul *et al.* 1998) has proposed an incremental maintenance algorithm for materialized views for semi-structured data based on the data model OEM (different than XML) and the query langage LOREL. However, the query operations, the possible update operations and the update method form only a very limited subset of possible requests on semi-structured data. Moreover in this approach, the internal identifiers of the objects must be known to import the data in the materialized view, which is not conceivable in the case of autonomous sources. Recently in the RAINBOW project, (El-Sayed *et al.* 2002) has proposed a method based on the decomposition of an XML source document update into basic primitive update operations (insert/delete/change of attribute or element) which apply to an identified position in the XML tree. However the assumption that the position of an update is known cannot apply to autonomous web sources, which do not provide this kind of information.

In this paper, we consider materialized views on web sources defined by an XQuery request on a mediator/wrapper architecture. We assume that the contents of the views are stored in a local XML database. This solution leads to two main questions: (1) what information is needed for updating the views ? and (2) how to process updates on the materialized views ?

In this paper, we answer these questions as follows: (1) we classify the different kinds of updates and propose a solution to maintain the materialized view, based on the notion of fragment, and (2) starting with the case of view defined by *one* operator, we study the general case of views defined by the combinaison of several operators.

The rest of this paper is organized as follows. The next section focusses on update notification information and Section 3 describes the various update cases. We conclude in Section 4 by summarizing the contributions and discussing future research.

2 Update Information

The interface between the mediator and the wrapper for querying sources with XQuery / XML (Dang-Ngoc and Gardarin 2003) has been extended to support materialized view maintenance. To this purpose, XML View registration and update notification from the wrapper have been incorporated in the wrapper functionnalities. The web wrapper loads web sources and keeps necessary information (essentially checksum and RDF) to check if sources have changed since last time it has been loaded, and send the necessary modification to the mediator.

Moreover, we assume, in this section and in Sections 3.1 and 3.2 that the view is defined by one operation: projection, restriction, cartesian product or join. The general case of a view definition involving several operators is outlined in Section 3.3. We refer to (Dang-Ngoc *et al.* 2004a) for more details.

2.1 Update Information

To know the information needed to update the view, we consider the parameters that charaterize the view computation and the source update. To do so, we first introduce the notions of collections and fragments.

2.1.1 Collections and fragments

An *XML collection* is a set of XML documents representing the same entity. In other words, a collection is a set of objects having a similar structure. For example, a collection of libraries should only have library objects. With semi-structured data, the structures of two objects in a collection can be equal or not (e.g an author object can be composed of a firstname only, whereas another is composed of a lastname, and a firstname).

An *XML fragment* is an identified subtree of an XML tree. A collection can be either a full web site where each page is a document relative to the collection, or a single page, where each XML fragment at the same level represents an object of this collection.

By analysing the XQuery request that defines the view, we can deduce the paths in the data source used by the request. We call this path set $\mathcal{S}$ the *useful XML fragment*. This fragment is a subtree of the tree collection of the source.

For example, let us apply the following XQuery request applied on the XML document of Figure 1 (b).

```
FOR $i in ("libraries.xml")/library/authors/
WHERE $i/author/year=1973
RETURN <result> {$i/author/firstname} {$i/author/books/book/title} </result>
```

To determinate the place of the fragment as precisely as possible and to minimize the necessary information for updates, we calculate what we call the *maximal prefix*. The maximal prefix is the longuest prefix common to the paths of the set $\mathcal{S}$. Then, the minimal fragment template is created by using only the useful paths and is annotated as in (Chen *et al.* 2003) (Dang-Ngoc *et al.* 2004b) to take into account mandatory and

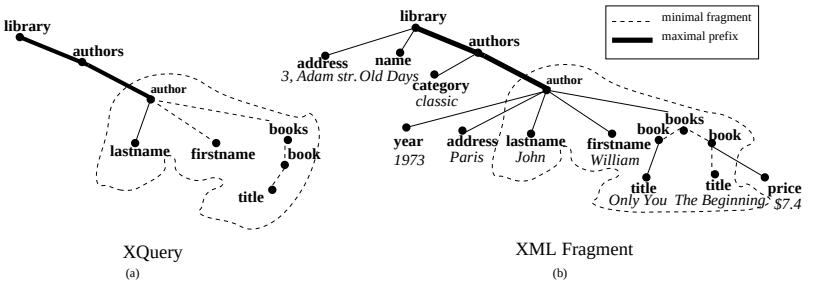

Figure 1: Useful fragment of an XQuery request

optional paths (resp. in solid and dotted links in Figure 1 (a)). In the previous XQuery example, all the paths used by the request have the path *library/authors* in common, and thus, this path is the maximal prefix.

Necessary information The necessary update information provided by the wrappers to notify an update to the materialized view is: (i) the XML minimal fragment (before and after update), (ii) the position of the fragment in the collection and the triple *(datatype, update, operation)*. where *type* is the datatype concerned by the source update (collection, fragment), *update* the type of source update (insertion, deletion, modification), and *operation* the operation defining the view (projection, restriction, cartesian product and join, see Section 3.3 for the general case).

Moreover, an analysis of the operators appearing in the view definition has enabled us to build a classification (see next section) of the different maintenance cases and has also allowed us to characterize the additional information necessary for the view maintenance. As we shall see in the next section, this information can be computed, either from the view itself or from the source.

2.2 Update Process

Having characterized the necessary information provided by the wrappers, we give an overview of the way views are maintained. The process to maintain the view can now be decomposed into two or three steps as follows:

1. *Update detection and notification / Analysis of the triple*: When an update on the source has been detected by the wrapper, the view materializer is notified about the update. The only information given at that time is the triple *(datatype, update, operation)*. Based on this triple, it is possible to determine which additional information is needed in order to maintain the view.

2. *Additional information (optional)*: a request builder queries the source or the view itself about the eventual additional information needed to maintain the views. We note that in the context of web sources, it is important to minimize the access to the sources. Thus, when the additional information is present in the view, we do not query the sources.

3. *Build the view update request / update the view*: At this stage of the process, we know the triple, and the information necessary for processing the update. Based on our analysis of each case, the view update request can be generated and processed.

In the architecture, we propose the following components for the processing of updates: A *request analyzer*, in order to know which operator (or sequence of operators) is used in the view definition. The other component is a *request builder*, in order to query the source or the view (when needed), and to compute the XQuery update request that is stored and processed according to a specified policy (see end of Section 2.1).

3 Update Classification

We present our method of view maintainance, using an example. To do so, we first recall from (Dang-Ngoc and Gardarin 2003) the notions of XRelation and XTuple.

3.1 XRelation

(Dang-Ngoc and Gardarin 2003) presents the physical algebra XAlgebra based on the relational operators designed for XML. These operators are called *XOperators*. This algebra aims to construct execution plans for the evaluation of XQuery and processes tuples (called *XTuples*) of tree structures. In an XRelation, domains are XML trees of given path sets, attributes (called XAttribute) are XPaths referencing nodes in the XML trees. Each XAttribute can be multi-valued (when referencing several sub-trees), or empty (when referencing no subtree). XRelations are ordered collections of XTuples, where each XTuple is composed of XPath named attributes, whose values reference subtrees in the collection of trees. As a result, the schema of an XRelation is of type $R(XPath+, [Path+])$, where $XPath$'s are the attributes and $Path$'s compose the path set of the XML trees. Figure 2 shows an XRelation and two of its XTuples. We note that the second attribute is multivaluated in the first XTuple, whereas this attribute is empty in the second XTuple.

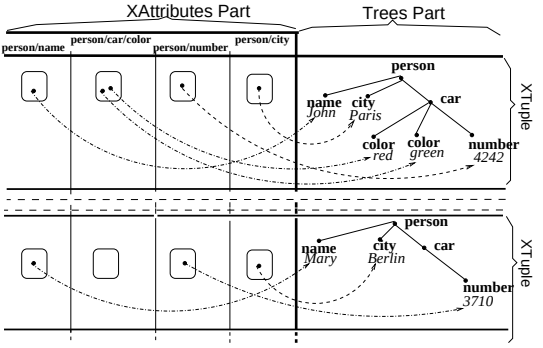

Figure 2: XTuple.

For the sake of lisibility, we simply write XTuples as ordered sets of values sets, ignoring references. It is important to note that this simplification implies that we consider the values in the leaves of trees. We refer to (Dang-Ngoc *et al.* 2004a) for the general case, including internal nodes. Let R be an XRelation defined by an expression of the XAlgebra on sources.

Now, we introduce *XTuple Identifiers* (XTID). An XTID is a pair *(source_num, XTuple_num)*. Each XTuple from the sources involved in the construction of one XTuple of R is associated with a distinct XTID (*XTuple_num* are generated incrementally). This XTID is associated to each XAttribute of the XTuple so that all its XAttributes have the same XTID. Once this XTID is fixed for each XAttribute of each XTuple, it cannot be changed in any operation applied to the XRelation. Duplicate XTuples have distinct XTID.

Referring to the XRelation in Figure 2 , the two XTuples are written as:
$(\{John\}_{11}, \{red, green\}_{11}, \{4242\}_{11}, \{Roma\}_{11})$
and $(\{Mary\}_{15}, \{\}_{15}, \{3710\}_{15}, \{Berlin\}_{15})$.

Below are examples of XRelations computed from two sources S_1 and S_2.

S_1

pers/name	*pers/car/col*	*pers/num*	*pers/city*
$\{John\}_{11}$	$\{red, green\}_{11}$	$\{4242\}_{11}$	$\{Roma\}_{11}$
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12}$	$\{London\}_{12}$
$\{John\}_{13}$	$\{red, green\}_{13}$	$\{4242\}_{13}$	$\{Roma\}_{13}$
$\{Mary\}_{14}$	$\{\}_{14}$	$\{3710\}_{14}$	$\{Berlin\}_{14}$

S_2

sal/num	*sal/stat*
$\{3710\}_{21}$	$\{baker\}_{21}$
$\{9999\}_{22}$	$\{grocer\}_{22}$

- P_1: Projection of S_1 on $pers/name, pers/car/col, pers/num$.

P_1

pers/name	*pers/car/col*	*pers/num*
$\{John\}_{11}$	$\{red, green\}_{11}$	$\{4242\}_{11}$
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12}$
$\{John\}_{13}$	$\{red, green\}_{13}$	$\{4242\}_{13}$
$\{Mary\}_{14}$	$\{\}_{14}$	$\{3710\}_{14}$

- R_1: Restriction of S_1 on $pers/num > 4000$

R_1

pers/name	*pers/car/col*	*pers/num*	*pers/city*
$\{John\}_{11}$	$\{red, green\}_{11}$	$\{4242\}_{11}$	$\{Roma\}_{11}$
$\{John\}_{13}$	$\{red, green\}_{13}$	$\{4242\}_{13}$	$\{Roma\}_{13}$

- J_1: Join of S_1 and S_2 on per/num and sal/num.

J_1

pers/name	*pers/car/col*	*pers/num, sal/num*	*pers/city*	*sal/stat*
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12,31}$	$\{London\}_{12}$	$\{baker\}_{21}$
$\{Mary\}_{14}$	$\{\}_{14}$	$\{3710\}_{14,31}$	$\{Berlin\}_{14}$	$\{baker\}_{21}$

- C_1: Cartesian product of S_1 and S_2 (in the resulting XRelation of a join or of a cartesian product, an XAttribute can be associated with more than one XTID).

C_1

pers/name	*pers/car/col*	*pers/num*	*pers/city*	*sal/num*	*sal/stat*
$\{John\}_{11}$	$\{red, green\}_{11}$	$\{4242\}_{11}$	$\{Roma\}_{11}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12}$	$\{London\}_{12}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{John\}_{13}$	$\{red, green\}_{13}$	$\{4242\}_{13}$	$\{Roma\}_{13}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{Mary\}_{14}$	$\{\}_{14}$	$\{3710\}_{14}$	$\{Berlin\}_{14}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{John\}_{11}$	$\{red, green\}_{11}$	$\{4242\}_{11}$	$\{Roma\}_{11}$	$\{9999\}_{22}$	$\{grocer\}_{22}$
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12}$	$\{London\}_{12}$	$\{9999\}_{22}$	$\{grocer\}_{22}$
$\{John\}_{13}$	$\{red, green\}_{13}$	$\{4242\}_{13}$	$\{Roma\}_{13}$	$\{9999\}_{22}$	$\{grocer\}_{22}$
$\{Mary\}_{14}$	$\{\}_{14}$	$\{3710\}_{14}$	$\{Berlin\}_{14}$	$\{9999\}_{22}$	$\{grocer\}_{22}$

3.2 Classification

We study all update cases introduced in Section 2.1.1. More precisely, we consider the following cases: for the type of element to modify: (i) a whole XML fragment, (ii) a part of an existing single element, (iii) a multivalued element, and (iv) a non-existing element. The update types we consider are insertion, deletion and modification (even though a modification can be seen as a deletion followed by an insertion). Finally, every elementary view operation is studied separately.

3.2.1 Identifying an XTuple

The method for identifying an XTuple, even after a succession of XOperations having combined columns, deleted or duplicated XTuples is the following: (a) from the succession of XOperations, we can deduce the initial range of each column, (b) from the XTID, we can know to which initial XTuple an XAttribute belongs. Finally, we can identify the original XTuple in spite of the mix implied by the XOperators. Note that by extending this method, we could also deduce which part of each original source has been used in the views.

3.2.2 Insertion of a fragment

Suppose a new fragment F represented by the new XTuple N is added in source S_1. An XTID is computed for the new XTuple and is added to every XAttribute of the Xtuple.

Projection: Unprojected columns of the new XTuple are removed and the new XTuple is added to the result XRelation.

For example: the insertion of $(\{Thomas\}_{15}, \{\}_{15}, \{5678\}_{15}, \{London\}_{15})$ into S_1 gives the following XRelation P_2.

P_2

pers/name	*pers/car/col*	*pers/num*
$\{John\}_{11}$	$\{red, green\}_{11}$	$\{4242\}_{11}$
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12}$
$\{John\}_{13}$	$\{red, green\}_{14}$	$\{4242\}_{13}$
$\{Mary\}_{14}$	$\{\}_{14}$	$\{3710\}_{14}$
$\{Thomas\}_{15}$	$\{\}_{15}$	$\{5678\}_{15}$

Restriction: The XTuple is added to the resulting XRelation if the predicate is satisfied.

For example, insertion of $(\{Peter\}_{16}, \{\}_{16}, \{1234\}_{16}, \{Roma\}_{16})$ does not change R_1, and the insertion of $(\{Thomas\}_{15}, \{\}_{15}, \{5678\}_{15}, \{London\}_{15})$ into S_1 gives the following XRelation R_2.

R_2

pers/name	*pers/car/col*	*pers/num*	*pers/city*
$\{John\}_{11}$	$\{red, green\}_{11}$	$\{4242\}_{11}$	$\{Roma\}_{11}$
$\{John\}_{13}$	$\{red, green\}_{13}$	$\{4242\}_{13}$	$\{Roma\}_{13}$
$\{Thomas\}_{15}$	$\{\}_{15}$	$\{5678\}_{15}$	$\{London\}_{15}$

Cartesian product: In this case, all XTuples NM for all XTuples M in S_2 have to be inserted into the view. To create these XTuples, we identify XTuples belonging to S_2 in the view itself, using the XTIDs.

For example, the insertion of $(\{Thomas\}_{15}, \{\}_{15}, \{5678\}_{15}, \{London\}_{15})$ into S_1 gives the following XRelation C_2. To compute C_2, we have considered an XTuple in the S_1 part with XTID 11 and selected all XTuples where 11 appears in. The columns associated to S_2 of these XTuples constitute the content of S_2.

C_2

pers/name	*pers/car/col*	*pers/num*	*pers/city*	*sal/num*	*sal/stat*
$\{John\}_{11}$	$\{red, green\}_{11}$	$\{4242\}_{11}$	$\{Roma\}_{11}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12}$	$\{London\}_{12}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{John\}_{13}$	$\{red, green\}_{13}$	$\{4242\}_{13}$	$\{Roma\}_{13}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{Mary\}_{14}$	$\{\}_{14}$	$\{3710\}_{14}$	$\{Berlin\}_{14}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{John\}_{11}$	$\{red, green\}_{11}$	$\{4242\}_{11}$	$\{Roma\}_{11}$	$\{9999\}_{22}$	$\{grocer\}_{22}$
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12}$	$\{London\}_{12}$	$\{9999\}_{22}$	$\{grocer\}_{22}$
$\{John\}_{13}$	$\{red, green\}_{13}$	$\{4242\}_{13}$	$\{Roma\}_{13}$	$\{9999\}_{22}$	$\{grocer\}_{22}$
$\{Mary\}_{14}$	$\{\}_{14}$	$\{3710\}_{14}$	$\{Berlin\}_{14}$	$\{9999\}_{22}$	$\{grocer\}_{22}$
$\{Thomas\}_{15}$	$\{\}_{15}$	$\{5678\}_{15}$	$\{London\}_{15}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{Thomas\}_{15}$	$\{\}_{15}$	$\{5678\}_{15}$	$\{London\}_{15}$	$\{9999\}_{22}$	$\{grocer\}_{22}$

Join: We first compute from the view, the set of all XTuples from S_2 that must be joined with N. We distinguish two cases: (1) If the resulting set is not empty, the new XTuples are created as in the cartesian product. (2) Otherwise, loading S_2 is necessary to compute the join with N.

For example, the insertions of $(\{Helen\}_{15}, \{\}_{15}, \{3710\}_{15}, \{London\}_{15})$ (case (1)), and of $(\{Steve\}_{16}, \{\}_{16}, \{9999\}_{16}, \{London\}_{16})$ (case (2)) into S_1 gives the following XRelation J_2. Moreover, the insertion of $(\{Bill\}_{17}, \{\}_{17}, \{8888\}_{17}, \{Roma\}_{17})$ does not change J_2. Note that S_2 has to be loaded to reach this conclusion.

J_2

pers/name	*pers/car/col*	*pers/num, sal/num*	*pers/city*	*sal/stat*
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12,31}$	$\{London\}_{12}$	$\{baker\}_{21}$
$\{Mary\}_{14}$	$\{\}_{14}$	$\{3710\}_{14,31}$	$\{Berlin\}_{14}$	$\{baker\}_{21}$
$\{Helen\}_{15}$	$\{\}_{15}$	$\{3710\}_{15,31}$	$\{London\}_{15}$	$\{baker\}_{21}$
$\{Steve\}_{16}$	$\{\}_{16}$	$\{9999\}_{16,32}$	$\{London\}_{16}$	$\{grocer\}_{22}$

3.2.3 Deletion of a fragment

Suppose a fragment F is deleted from source S_1, and that F is represented by the XTuple N with XTID a. Then to maintain the view, all XTuples such that at least one of their XAttribute has this XTID, are deleted from the XRelation.

For example, let us consider the deletion from S_1 of the fragment $F = (\{John\}, \{red, green\}, \{4242\}, \{Roma\})$. First, we identify all XTuples in the view matching the fragment, *i.e.*, the two XTuples with XTIDs 11 and 13. Considering one of these XTIDs, for example 11, we remove all XTuples in the view having XTID 11 in their XAttributes. The resulting XRelations of this deletion for each XOperation considered in this example, are shown below.

- P_3: the resulting state of P_1 after deleting F from S_1

P_3

pers/name	*pers/car/col*	*pers/num*
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12}$
$\{John\}_{13}$	$\{red, green\}_{13}$	$\{4242\}_{13}$
$\{Mary\}_{14}$	$\{\}_{14}$	$\{3710\}_{14}$

- R_3: the resulting state of R_1 after deleting F from S_1

R_3

pers/name	*pers/car/col*	*pers/num*	*pers/city*
$\{John\}_{13}$	$\{red, green\}_{13}$	$\{4242\}_{13}$	$\{Roma\}_{13}$

- C_3: the resulting state of C_1 after deleting F from S_1

C_3

pers/name	*pers/car/col*	*pers/num*	*pers/city*	*sal/num*	*sal/stat*
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12}$	$\{London\}_{12}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{John\}_{13}$	$\{red, green\}_{13}$	$\{4242\}_{13}$	$\{Roma\}_{13}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{Mary\}_{14}$	$\{\}_{14}$	$\{3710\}_{14}$	$\{Berlin\}_{14}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12}$	$\{London\}_{12}$	$\{9999\}_{22}$	$\{grocer\}_{22}$
$\{John\}_{13}$	$\{red, green\}_{13}$	$\{4242\}_{13}$	$\{Roma\}_{13}$	$\{9999\}_{22}$	$\{grocer\}_{22}$
$\{Mary\}_{14}$	$\{\}_{14}$	$\{3710\}_{14}$	$\{Berlin\}_{14}$	$\{9999\}_{22}$	$\{grocer\}_{22}$

- J_3: the resulting state of J_1 after deleting F from S_1

J_3

pers/name	*pers/car/col*	*pers/num, sal/num*	*pers/city*	*sal/stat*
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12,31}$	$\{London\}_{12}$	$\{baker\}_{21}$

We note that modifications of fragments are treated by deletions followed by insertions.

3.2.4 Inserting, Deleting and Modifying an Element

Insertion: We consider the case of inserting a node (which can be the root of a whole subtree) N in an existing fragment F of the source S_1 used by the XTuple N with XTID a. In this case, we just have to add the new node in the set of nodes of the XAttributes identified by the XTID a. Note that prefix/suffix must be considered, because, all XAttributes referencing the path must be updated.

For example, in S_1, consider the insertion of element color *blue* in the fragment, for which the minimal fragment after projection gives:

$$(\{John\}, \{red, green\}, \{4242\}, \{Roma\})$$

. By identifying a matching XTuple (see Section 3.2.1), we see that the insertion affects the XTuples in the view P_1 having XTIDs 11 and 13. As these Xtuples are equal in the view, we take one of them, for example the one having XTID 11, and we add the value *blue* in the set ($\{red, green\}_{11}$). We obtain the following XRelation.

P_4

pers/name	*pers/car/col*	*pers/num*
$\{John\}_{11}$	$\{red, green, \mathbf{blue}\}_{11}$	$\{4242\}_{11}$
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12}$
$\{John\}_{13}$	$\{red, green\}_{13}$	$\{4242\}_{13}$
$\{Mary\}_{14}$	$\{\}_{14}$	$\{3710\}_{14}$

We note that in the case of elements, modifications are treated by identifying the element in the value set and replacing it by the new value.

Deletion: We consider the deletion of a node (which can be the root of a subtree) N from an existing fragment F of the source S_1 used by the XTuple N with XTID a. In this case, we have to delete the node from the set of nodes of the XAttributes identified by the XTID a in the right column. If the node is alone in the set in a column used as a predicate in the XOperation, all final XTuples having an XAttribute with XTID a must be removed. Note that, here again, prefix/suffix must be considered, because all XAttributes referencing the path must be updated.

For example, in S_1, we consider the deletion of element color *green* in one occurence of the fragment $(\{John\}, \{red, green\}, \{4242\}, \{Roma\})$. In C_1, this deletion affects one of the XTuples having XTID 11 or 13. If we choose 11, then we delete the element

color *green* in each Xtuple where the element *color* has the XTID 11, and get the following XRelation C_4.

C_4

pers/name	*pers/car/col*	*pers/num*	*pers/city*	*sal/num*	*sal/stat*
$\{John\}_{11}$	$\{red\}_{11}$	$\{4242\}_{11}$	$\{Roma\}_{11}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12}$	$\{London\}_{12}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{John\}_{13}$	$\{red, green\}_{13}$	$\{4242\}_{13}$	$\{Roma\}_{13}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{Mary\}_{14}$	$\{\}_{14}$	$\{3710\}_{14}$	$\{Berlin\}_{14}$	$\{3710\}_{21}$	$\{baker\}_{21}$
$\{John\}_{11}$	$\{red\}_{11}$	$\{4242\}_{11}$	$\{Roma\}_{11}$	$\{9999\}_{22}$	$\{grocer\}_{22}$
$\{Mickael\}_{12}$	$\{\}_{12}$	$\{3710\}_{12}$	$\{London\}_{12}$	$\{9999\}_{22}$	$\{grocer\}_{22}$
$\{John\}_{13}$	$\{red, green\}_{13}$	$\{4242\}_{13}$	$\{Roma\}_{13}$	$\{9999\}_{22}$	$\{grocer\}_{22}$
$\{Mary\}_{14}$	$\{\}_{14}$	$\{3710\}_{14}$	$\{Berlin\}_{14}$	$\{9999\}_{22}$	$\{grocer\}_{22}$

3.3 Combination of XOperators

In this section, we outline the case where the view is defined by a combination of XOperators. In order to maintain it when sources change, we must introduce the notion of *hidden columns*. A column is marked hidden when a projection eliminates it, while it is used in a predicate in XOperators (as restriction, or join). Notice that if a hidden column does not appear in the client application using the view, then this column is manipulated as all other columns when maintaining the view.

Figure 3 (a) shows a detailed example of an XAlgebra tree representing a view defined on two sources S_1 and S_2 mapped to the XRelations R_1 and R_2 respectively. For the sake of lisibility, only the P part of XRelations has been represented, but Figure 3 (b) and Figure 3 (c) show how the second (resp. third) XTuple of S_1 (resp. S_2) is really represented. Figure 3 (d) shows the second XTuple of the final XRelation. Every lower case letter represents a distinct set of values references, and numbers on the right side are XTIDs. Upper case letters represent names of columns (*i.e., paths expressions*). Note that in this example, we have duplicate rows (2^{nd} and 3^{rd} on R_1) and multivaluated nodes (g_{12}). During XOperations (restriction, join, projection), columns and rows are combined, deleted or created, but it is alway possible to compute the *useful* columns. In our example, in the final XRelation R_F, we can reconstruct the useful part of the third initial XTuples. Finally, hidden columns can appear after projection. They are represented in grey on the figure.

In (Dang-Ngoc *et al.* 2004a), we describe how to support combinations of XOperations in the view definition based on the final view, and the simple operations in the view definition. To do so, we replace in the triple as defined in Section 2.1.1, *operation* by *request* where *request* is the XML request.

In our example, suppose we insert a new fragment in the source S_1 represented by the XTuple $(a_{16}, y_{16}, g_{16}, h_{16})$. Applying the projection on B, C, D gives (y_{16}, g_{16}, h_{16}). Then applying the join means that from the final XRelation, we are able to rebuild the M, N columns of XTuples where $N = h$. For example, we select the M and N columns of XTuples with XTID 12, and we get the pairs (h_{23}, u_{23}) and (h_{24}, w_{24}) that are combined with (y_{16}, g_{16}, h_{16}). So finally, the XTuples to be added to the final view are $(y_{16}, g_{16}, h_{16,23}, u_{23})$ and $(y_{16}, g_{16}, h_{16,24}, w_{24})$. (See insertion in Figure 3 (a)).

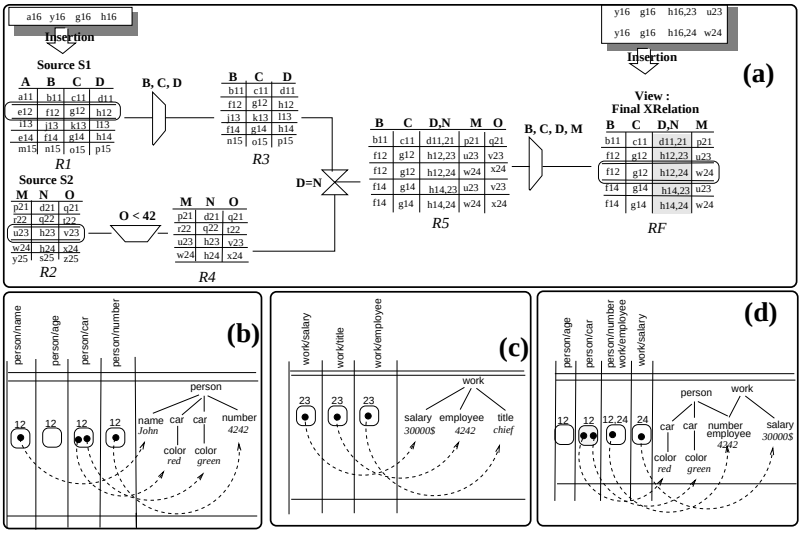

Figure 3: An execution plan with XRelations and XTID.

4 Conclusions and Future Works

Integrating web information has become an urgent need, and taking into account the specificities of web information is a challenge. Indeed, web sources are semi-structured, the sources can be unreachable and they frequently change but do not provide any information about their changes.

In this paper, we have proposed a solution for integrating web sources into a mediation architecture based on materialized view. Update information are sent to the mediator as an XML minimal fragment. Then we have given a classification of all cases of maintenance of materialized views by considering the update type, the datatype concerned by the update and the operations defining the view.

We are currently implementing our approach through a module called *XML View Refresher* to be integrated in the full XML mediation architecture XLive.

In this paper, we have considered materialized views whose definitions do not incorporate any reconstruction phase. However this feature is commonly used in XQuery requests in order to output the resulting data according to a format specified in this phase. As this point is important for view definition, we are currently studying the impact of the reconstruction phase in our approach.

References

Abiteboul, S., McHugh, J., Rys, M., Vassalos, V., and Wiener, J. L. (1998). Incremental Maintenance for Materialized Views over Semistructured Data. In *Proc. of the Twenty-fourth Intl Conf VLDB*, 1998.

Chen, Z., Jagadish, H. V., Lakshmanan, L. V., and Paparizos, S. (2003). From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Procs of 29th Intl Conf VLDB*, 2003.

Dang-Ngoc, T.-T. and Gardarin, G. (2003). Federating heterogeneous Data Sources with XML. In *Proc. of IASTED IKS Conf.*, 2003.

Dang-Ngoc, T.-T., Laurent, D., and Sans, V. (2004a). Maintenance of Materialized Views on Distributed Sources (to appear). Technical report, LICP - University of Cergy-Pontoise.

Dang-Ngoc, T.-T., Gardarin, G., and Travers, N. (2004b). Tree Graph View: On Efficient Evaluation of XQuery in an XML Mediator. In *Bases de Données Avancées BDA (to appear)*, 2004.

Draper, D., Halevy, A. Y., and Weld, D. S. (2001). The Nimble Integration Engine. In *SIGMOD Record (ACM Special Interest Group on Management of Data)*, volume 30, 2001.

El-Sayed, M., Wang, L., Ding, L., and Rundensteiner, E. A. (2002). An Algebraic Approach for Incremental Maintenance of Materialized XQuery Views. In *Proc. of the Fourth Intl Workshop on Web Information and Data Management (WIDM-02)*, 2002.

Gupta, A., Ludäscher, B., Baru, C., Velikhov, P., Marciano, R., and Papakonstantinou, Y. (1999). XML-based information mediation with MIX. In *SIGMOD 1999, Proc. ACM SIGMOD Intl Conf on Management of Data*, 1999.

Laurent, D., Lechtenbörger, J., Spyratos, N., and Vossen, G. (2001). Monotonic Complements for Independant Data Warehouses. *Intl Journal of VLDB*, **10**(4), 2001.

Wiederhold, G. (1992). Mediators in the Architecture of Future Information Systems. *Computer*, **25**(3), 1992.

Summary

In the last years, with the growing and diversity of information in electronic form accessible from the Web, data integration systems -such as mediators- were designed to integrate these distributed and heterogeneous data in a uniform view. To facilitate the integration of data across different systems, XML has been adopted as the standard format for information exchange. XQuery, a powerful language for querying XML, has become a major requirement for XML-based systems, and is used in mediation systems to design views on several data sources. To optimize query evaluation, views are materialized. The difficulty is to maintain incrementally materialized views while sources are updated. In the context of web sources, very few informations are provided by sources, and methods usually proposed do not apply in this context. This paper studies how to update materialized XML views on web sources in the context of a mediation architecture.