On the Maintenance of XML Materialized Views

Tuyet-Tram Dang-Ngoc, Dominique Laurent, Virginie Sans

On the Maintenance of Materialized XML Views

Tuyet Tram Dang Ngoc, Dominique Laurent, Virginie Sans

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

Abstract. Providing services by integrating information available in web resources is one of the main goals of a mediation architecture. In this paper, we consider the standard wrapper-mediator architecture under the following hypothesis: (*i*) the information exchanged between wrappers and the mediator consists in XML documents, (*ii*) wrappers have limited resources, and (*iii*) to answer queries even if sources are not available, materialized XML views are stored at the mediator level. In this setting, we focus on the problem of maintaining materialized XML views, when the sources change. In our context, wrappers send the updated document without providing any information about the type and the localization of the update in the document. Then, the problems we address are, first, identifying the updates, and, second, updating the view in such a way that accesses to the sources are restricted. Our approach is based on the XAlgebra, which allows to consider XQuery requests on XML documents as relational tables. Moreover, our solution uses identifier annotations for XAlgebra and a *diff* function.

1 Introduction

The issue of integrating heterogeneous and distributed data has been adressed in [11] by means of mediation architectures. Given heterogeneous and distributed data sources to be integrated, a mediation architecture roughly consists of two components: *wrappers* and a *mediator*.

Each wrapper is associated to a given data source and is in charge of (*i*) extracting information from this source, (*ii*) transforming this information in an appropriate format understandable by the mediator, and (*iii*) sending the transformed information to the mediator.

On the other hand, the mediator integrates the information coming from the wrappers and provides the result to the final application. In order to make mediation architectures dealing with web sources efficient, the following issues have to be investigated further:

1. *Accessibility of the data.* On the web, data may not be reachable, and thus accessing the sources as few as possible is important. This is why, we propose a mediation architecture in wich materialized views stored at the mediator level are used. In this way, queries are addressed to the mediator (instead

of being addressed to the sources). However, this implies that the views be mantained up to date, which is precisely one of the topics of the paper.

2. *Querying the web sources.* When querying a web source, the answer is the whole content of the source. In other words, for such queries, there is no query language allowing to answer sophiticated queries, as for instance, SQL which allows to query a part of a database. As a consequence, we consider in our approach that, for each access to a data source, we get the whole content of that source.
3. *Non cooperative mode.* Since web sources are autonomous, they use a push model to notify about their changes. This means that we can only assume that a wrapper knows that its associated source has changed, without any further information on the type or the localization of the change. In this paper, we propose a method for update detection that computes the needed information for the maintenance of materialized views.

Summing up the contributions of our work, we propose an approach to integrate efficiently heterogeneous and distributed web sources under the following assumptions: (*i*) the web sources are integrated through materialized XML views that are defined by an XQuery request and stored at the mediator level, and (*ii*) wrappers have limited computational resources, meaning in particular that data sources cannot be duplicated at the wrapper level.

As mentioned above, in such a mediation architecture, querying the sources amounts to querying the materialized views, thus avoiding to access the sources. However, it is well known that the price to pay with such an approach is that materialized views must be maintained up to date. Our approach to maintain the materialized views up to date follows an incremental processing, as in [2], which is based on the *XAlgebra*, introduced in [5]. The XAlgebra allows to represent an XML document as a relational table, called *XRelation*, on which relational operators, called *XOperators*, are applied. In this setting, a materialized view is seen as an XRelation, on which updates are performed in response to updates on the data sources.

On the other hand, as stated earlier (see item 3 above), the sources do not provide complete information about their changes. Since this information is necessary to maintain the views, we present an approach of update detection that can be summarized as follows:

> When the wrapper is informed that a source has been updated, it sends the whole content of the source to the mediator. Since at the mediator level, the definitions of the views are known, the part of the source used to compute the view can be recovered as an XRelation, say X. Consequently, by applying a standard *diff* function on X and the new state of the source sent by the wrapper, the mediator can identify the type and the localization of the update.

It is important to note that, in our approach, we annotate components of the tuples in XRelations with identifiers, that we call *XTID*s. We also would like to emphasize that, in our approach, the computation of the part of the source used in the materialized view, allows for an efficient maintenance, since, except in restricted cases, this information prevents from accessing the data sources.

The maintenance of materialized views in the case of relational databases has been the subject of many research works these last decades, and regained interest recently in the context of relational data warehouses, that are based on a mediator/wrapper architecture (e.g. see [9]). However, although our approach is based on a relational representation of XML documents, relational approaches do not apply in the case of semi-structured data, because, for instance, multi-valued attributes are considered in our case.

On the other hand, techniques specially designed for semi-structured data have been introduced in the literature. For example, in [1], the maintenance of materialized views is studied in the Object Embedded Model (OEM) and in the context of the language Lorel, for views that are defined with selections, projections and joins. However, the process in [1] is based on the fact that internal Ids are available, which is not the case for autonomous web sources.

In the Rainbow project [7], the authors consider the query language XQuery as we do, but their approach requires to know the exact position in the XML tree where the update should be done. We recall that, in our approach, we assume that this information is not provided by the source.

A closer approach to ours is given in [4]. This work gives a solution for the maintenance of materialized views defined over non cooperative sources. However, this work differs from ours, because in [4], wrappers are assumed to store a complete copy of the source, whereas in our work, when the wrapper detects a change, the complete source is sent to the mediator.

The paper is organized as follows. The next Section 2 introduces the main concepts of the mediation architecture we consider in our work and the XAlgebra. Section 3 presents our update detection mechanism, and Section 4 deals with the view update processing. Finally, Section 5 concludes the paper.

2 Background

2.1 Mediation Architecture

According to [11], the major two components of a mediation architecture system are: (*i*) a mediator and (*ii*) wrappers, each of which being associated with one source. Among all the available platforms for mediation, we consider in our work, the XLive architecture, first proposed in [5].

In its current state, the mediation architecture XLive does not support materialized XML views. To do so, the following components have to be added:

- *A local XML database at the mediator level.* The materialized XML views we consider are stored at the mediator level in an XML database than can support view materialization.

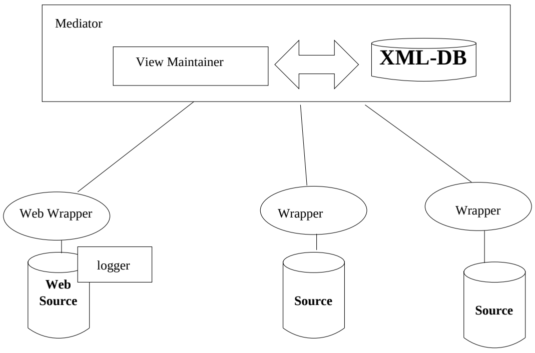


Fig. 1. Our framework

- *A view manager at the mediator level.* This component is meant to allow the mediator to partially reconstruct the sources, and to compute the XQuery requests necessary to the maintenance of the views.
- *A notification component at the wrapper level.* Usually, a wrapper is simply a translator, whereas, in our approach, a notification component, which we call *logger*, is needed in order to detect updates in the source. We recall in this respect that, upon changes on a source, the corresponding wrapper sends the modified document to the mediator.

This architecture, that we are currently implementing, is shown in Figure 1.

2.2 XAlgebra

To query an XML document, many languages have been proposed such as Lorel ([3]), XML-QL ([6]) or XQuery ([10]). In our work, we consider the language XQuery as this language provides more possibilities than the others.

Moreover, in [5], it has been shown that an XQuery request can be associated with an expression from an algebra, called the XAlgebra, which is based on relational operators designed for XML. In [5], this algebra has been used to construct execution plans for the evaluation of XQuery requests. In our work, we consider the XAlgebra for the maintenance of materialized XML views.

Roughly speaking, the XAlgebra is a set of operators, called *XOperators*, inspired by those from the standard relational algebra, that operate on “tabular” structures called *XRelations*. In what follows, we recall the basics of XRelations and of the XAlgebra.

XRelations An XRelation is composed of two distinct parts called the *XAttributes Part* and the *Trees Part*, respectively. In the Trees Part, the domain is a set of XML trees of given path sets. In the XAttributes Part, attributes, called *XAttributes*, are XPaths and their respective domains are sets of references to XML trees.

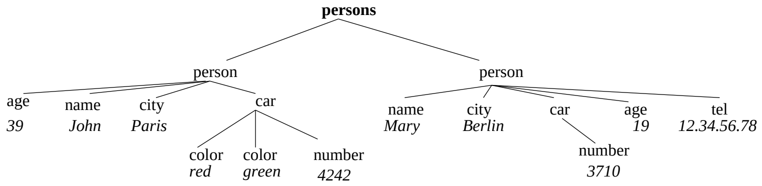


Fig. 2. XML Tree

Contrary to standard relations, each XAttribute can be multi-valued (when referencing several sub-trees), or empty (when referencing no subtree). Moreover, XTuples in an XRelation are stored according to a specific ordering that reflects the structure of the corresponding subtrees: if an XTuple x appears *before* an XTuple y, then, in the XML tree, the subtree corresponding to x appears on the *left hand side* of the subtree corresponding to y. Thus, XRelations are seen as ordered collections of XTuples, where each XTuple is:

- an XML tree, say t, in its Trees Part, and
- a tuple of sets of reference to subtrees of t, in its XAttributes Part.

As a result, the schema of an XRelation R is of type $(XPath^{+}, [Path^{+}])$, where $XPath$s are the XAttributes and $Path$s compose the path set of the corresponding XML tree.

Figure 3 shows an XRelation with two XTuples that have been obtained from the XML tree of Figure 2. We note that the second XAttribute is multivalued in the first XTuple, whereas this XAttribute is empty in the second XTuple.

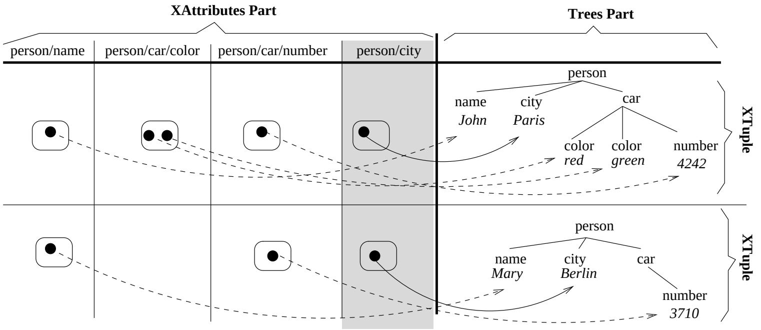


Fig. 3. XRelation

For the sake of lisibility, we simply write XTuples as ordered sets of values, ignoring references. It is important to note that this simplification implies that we consider the values in the leaves of trees.

XOperators The subset of relational XOperators from the XAlgebra considered in this paper is limited to the following ones: union, intersection, projection, restriction, join, and cartesian product.

These operators are defined in a similar way as in the relational algebra, we refer to [5] for more details. However, it is important to note that, the above simplification implies that we consider duplicates in XRelations.

Furthermore, the XAlgebra contains two specific operators, called *XSource* and *XConstruct*, that work as follows, respectively:

- Given an XML document, the operator XSource transforms the content of the source into an XRelation.
- Given an Xrelation, the operator XConstruct transforms the XRelation into an XML document.

Now, given a view V defined as an XQuery request over data sources, V can be seen as an expression of the XAlgebra, using XOperators. As a consequence, the answer to V, which is an XML document, can be seen as an XRelation.

In Figure 4, the computations of this XRelation and of the answer to V are represented in the particular case where V is defined over one single source (the case of more than one source is similar). Moreover, in this figure, the star associated with the grey box represents the fact that more than one XOperator can be used in the computations. The process of these computations works according to the following steps:

1. All sources involved in the XQuery defining V are transformed into XRelations, using the XSource operator.
2. The analysis of the XQuery request defining V allows to define an expression of the XAlgebra involving only XOperators among union, intersection, projection, restriction, join and cartesian product. This expression is computed against the XRelations obtained at the previous step.
3. The XRelation output at the previous step is transformed into an XML document, using the XConstruct operator.

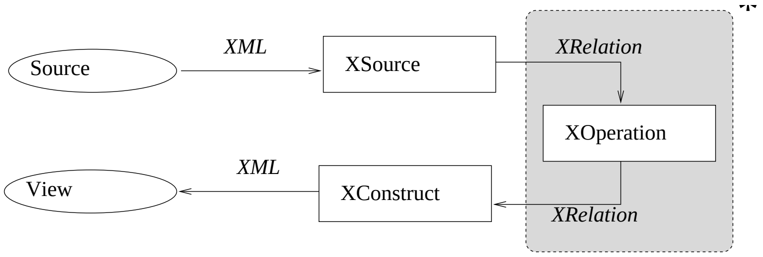


Fig. 4. Steps of the computation of an XQuery request

2.3 Mandatory, Optional and Hidden Paths

The XQuery syntax allows us to distinguish three kinds of paths, namely the *mandatory* paths, the *optional* paths, and the *hidden* paths. Given a view V defined by an XQuery request Q, these kinds of paths are defined as follows:

- Every path appearing in the *return* clause of Q is said to be optional.
- Every path appearing in the *where* clause of Q is said to be mandatory.
- Every path that is mandatory but not optional is said to be hidden.

Intuitively, hidden paths are those that are necessary to compute the answer to Q but that do not appear in the answer. It is important to note that hidden paths are necessary in our method for maintaining materialized views. Therefore, in our approach, hidden paths are stored but not displayed when the view is queried.

As an example, let us compute the following XQuery request Q on the XML document represented in Figure 2.

```
FOR $i in ("persons.xml")/persons
WHERE $i/person/city=Berlin
RETURN
<result>
{$i/person/name}
{$i/person/car/color}
{$i/person/car/number}
</result>
```

In this case, the XPath `persons/person/city` is mandatory, whereas the XPaths `persons/person/name` and `persons/person/lastname` are optional. Since `person/city` is mandatory but not optional, this path is hidden. In Figure 3, the corresponding column is displayed as a grey colomn.

3 Update Detection and Reconstruction

As mentioned earlier in the paper, we consider autonomous data sources associated with light wrappers that just send the updated document to the mediator, without providing any further information about the type and localization of the update in the document. In this case, in order to maintain materialized views, we have to identify the update. We note in this respect that the approach of [4] does not meet our restriction that wrappers have limited computational resources, since in [4], sources are duplicated at the wrapper level.

In our approach, the logger component of the wrapper is in charge of checking whether the source has changed. This can be done by computing a checksum on the source and comparing the value with a previous computed checksum. If the two values are different, then the source has been modified, and in this case, the content of the source is sent to the mediator.

Next, we show how the mediator can precisely determine which update has been performed on the source. Our method is based on the facts that (*i*) we associate XTuples from the sources with identifiers, called *XTID*s, and (*ii*) the parts of the sources that have been used in the computation of the view can be reconstructed, based on XTIDs and on the expression of the XAlgebra that defines the view.

3.1 XTID Identifiers

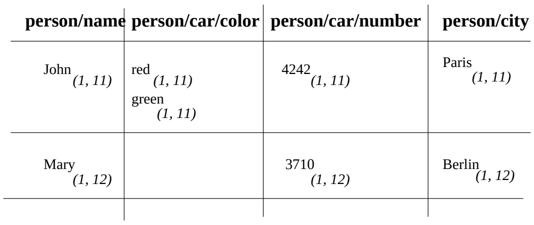
Let X be an XRelation associated to an XQuery request Q. We associate each set of pointers in any XTuple in X with a set of identifiers called *XTID*s. The role of XTIDs associated to a set S of pointers is to identify the sources and the tuples in these sources pointers in S come from.

An XTID is a pair (s, id), where s is a source identifier and id the XTuple identifier from source s. Let $x = (x^1, \ldots, x^p)$ be an XTuple in X, and let P^k be the XPath associated to x^k, for $k = 1, \ldots, p$. Assume that x is obtained as a combination of $x_1, \ldots, x_n$ where, for every $j = 1, \ldots, n$, x_j is an XTuple from the XRelation X_j associated to source j. Then, for every $k = 1, \ldots, p$, x^p is associated with a set of TIDs as follows:

> For every $k = 1, \ldots, p$ such that P^k occurs in the XPaths that define x_j, the XTID (j, id) is inserted into the set of XTIDs associated to x^k.

It is important to note that id is a unique identifier associated with x_j, which implies that duplicate XTuples have distinct XTIDs. On the other hand, recalling that an XRelation is an *ordered* collection of XTuples, the assignment of XTIDs from a given source is done in such a way that their ordering matches the storage ordering in the XRelation. In other words if, for a given source, XTuple x appears *before* the XTuple y in the corresponding XRelation, then the identifer assigned to x is *smaller* than the identifier assigned to y. Moreover, once such XTID is fixed for each XAttribute of each XTuple, it cannot be changed by any computation applied to the XRelation.

Figure 5 shows the annotation with XTIDs for the XRelation of Figure 3. For instance, with the source 1 and the first fragment, the XTuple is associated to XTID $(1, 11)$.



person/name	person/car/color	person/car/number	person/city
John *(1, 11)*	red *(1, 11)* green *(1, 11)*	4242 *(1, 11)*	Paris *(1, 11)*
Mary *(1, 12)*		3710 *(1, 12)*	Berlin *(1, 12)*

Fig. 5. Annotating an XRelation with XTIDs

3.2 Reconstruction

Let Q be an XQuery over sources $S_1, S_2, ..., S_n$, and let V be the corresponding expression of the XAlgebra. Let us denote by X_1, X_2, ..., X_n the XRelations generated by the application of the operator XSource to $S_1, S_2, ..., S_n$, repectively, and let X be the XRelation associated to V.

In what follows, we consider V as a mapping, denoted by v, that associates n XRelations to one XRelation. Thus, using our notation, we have

$$v(X_1, X_2, ..., X_n) = X.$$

As shown below, our reconstruction method is based on the inverse of v.

For a given integer σ in $\{1, \ldots, n\}$ identifying a data source involved in the computation of V, let us denote by $v_\sigma^{-1}(X)$ the set $\bigcup_{x \in X} v_\sigma^{-1}(x)$.

Intuitively, $v_\sigma^{-1}(X)$ is the part of the original XRelation associated to the source σ, that has been used in the processing of the view. In the remainder of the paper, $v_\sigma^{-1}(X)$ is called the *useful part* of the XRelation X_σ.

Let us consider an XTuple $x = (x^1, \ldots, x^m)$ in the XRelation X. Then, for every $k = 1, \ldots, m$, x^j is of the form c_{XTID} where c is the content value, and $XTID = \{xtid_1, xtid_2, ..., xtid_p\}$ is the set of XTIDs associated to c.

For every $k = 1, \ldots, m$, $xtid_k$ can be written as $xtid_k = (s_k, i_k)$, where s_k is a source identifier and i_k an XTID associated with an XTuple in s_k. The XRelation $v_\sigma^{-1}(X)$ can be computed from the XRelation X, using the following rules:

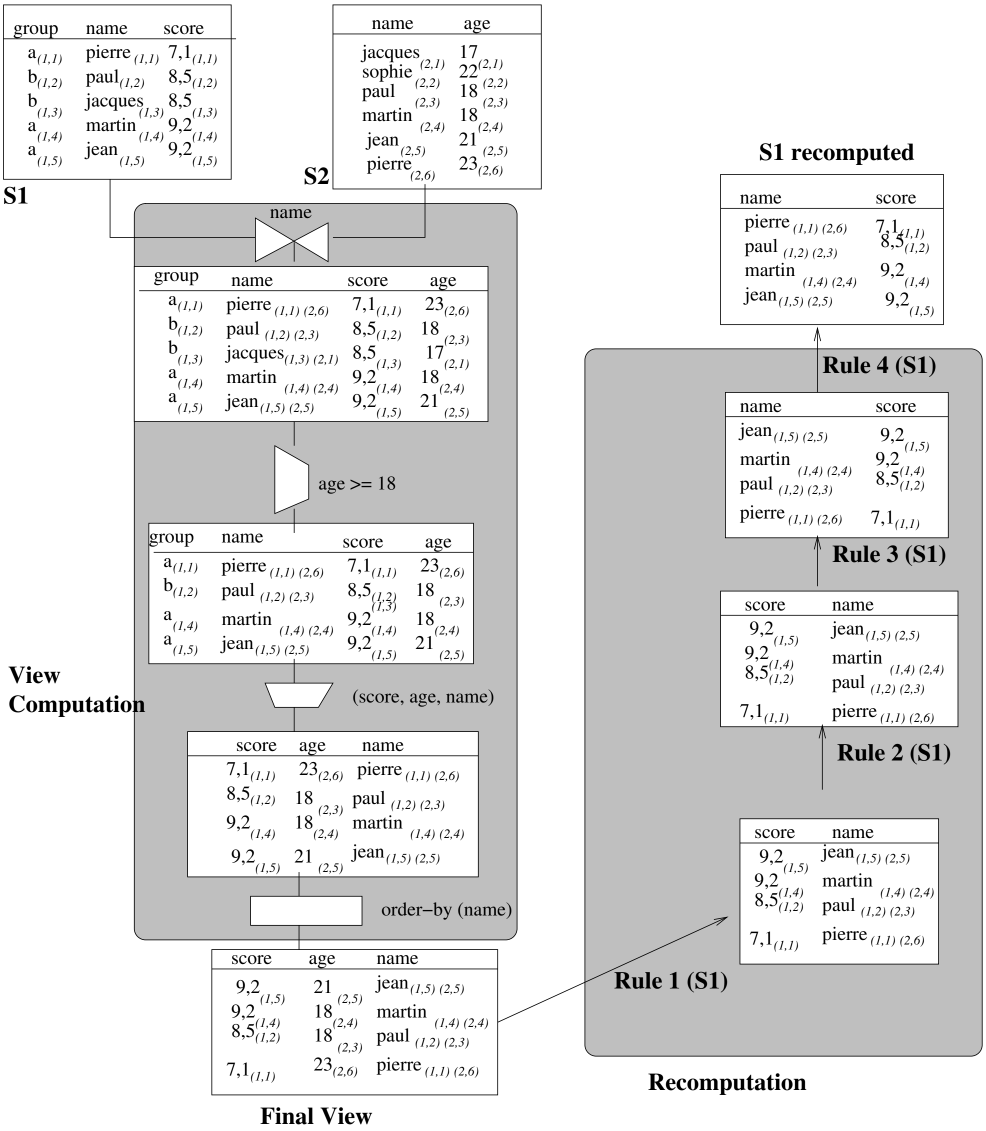
- *Rule 1.* If $XTID$ contains a pair (σ, i), then c belongs to a tuple in $v_\sigma^{-1}(X)$.
- *Rule 2.* For all k, l in $\{1, \ldots, m\}$, if $XTID_k \cap XTID_l$ contains a pair (σ, i), then c_k and c_l belong to the same XTuple in $v_\sigma^{-1}(X)$.
- *Rule 3.* For every k in $\{1, \ldots, m\}$, every s_k appearing in the set $XTID$ associated to c^k_{XTID}, the column number of the corresponding tuple containing c^k in the source s_k can be retrieved by comparing the positions of the XPaths occurring in the *for* and *return* clauses of the XQuery request Q that defines the view.
- *Rule 4.* Given a source σ and an XTuple $x = (c^1_{XTID_1}, \ldots, c^{m_\sigma}_{XTID_{m_\sigma}})$ computed by rules 1-3 above, if there exist i and j such that $(\sigma, i) \in c^k_{XTID_k}$ and $(\sigma, j) \in c^l_{XITD_l}$, then $i = j$. This value is denoted by $row(x, \sigma)$.
  If x and x' are two XTuples such that $row(x, \sigma) \leq row(x', \sigma)$, then x is inserted in $v_\sigma^{-1}(X)$ before x'.

Figure 6 shows an example of the previous rules applied on two data sources S_1 and S_2 and the following XQuery request:

```
for $n in document ("note.xml")/note
for $p in document ("person.xml")/person
where $n/name = $p/name and $age >= 18
return <result> {/p/age} {/p/name} {/n/note} </result>
```

The view computation box shows the computation of the view from the XRelations generated by XSource applied to S_1 and to S_2. On the other hand, the source reconstruction box shows the four steps for reconstructing the useful part of source S_1.

- *Rule 1:* Only values concerned by source S_1 are kept.



S1

group	name	score
a$_{(1,1)}$	pierre$_{(1,1)}$	7,1$_{(1,1)}$
b$_{(1,2)}$	paul$_{(1,2)}$	8,5$_{(1,2)}$
b$_{(1,3)}$	jacques$_{(1,3)}$	8,5$_{(1,3)}$
a$_{(1,4)}$	martin$_{(1,4)}$	9,2$_{(1,4)}$
a$_{(1,5)}$	jean$_{(1,5)}$	9,2$_{(1,5)}$

S2

name	age
jacques$_{(2,1)}$	17$_{(2,1)}$
sophie$_{(2,2)}$	22$_{(2,2)}$
paul$_{(2,3)}$	18$_{(2,3)}$
martin$_{(2,4)}$	18$_{(2,4)}$
jean$_{(2,5)}$	21$_{(2,5)}$
pierre$_{(2,6)}$	23$_{(2,6)}$

group	name	score	age
a$_{(1,1)}$	pierre$_{(1,1)\ (2,6)}$	7,1$_{(1,1)}$	23$_{(2,6)}$
b$_{(1,2)}$	paul$_{(1,2)\ (2,3)}$	8,5$_{(1,2)}$	18$_{(2,3)}$
b$_{(1,3)}$	jacques$_{(1,3)\ (2,1)}$	8,5$_{(1,3)}$	17$_{(2,1)}$
a$_{(1,4)}$	martin$_{(1,4)\ (2,4)}$	9,2$_{(1,4)}$	18$_{(2,4)}$
a$_{(1,5)}$	jean$_{(1,5)\ (2,5)}$	9,2$_{(1,5)}$	21$_{(2,5)}$

group	name	score	age
a$_{(1,1)}$	pierre$_{(1,1)\ (2,6)}$	7,1$_{(1,1)}$	23$_{(2,6)}$
b$_{(1,2)}$	paul$_{(1,2)\ (2,3)}$	8,5$_{(1,2)\ (1,3)}$	18$_{(2,3)}$
a$_{(1,4)}$	martin$_{(1,4)\ (2,4)}$	9,2$_{(1,4)}$	18$_{(2,4)}$
a$_{(1,5)}$	jean$_{(1,5)\ (2,5)}$	9,2$_{(1,5)}$	21$_{(2,5)}$

score	age	name
7,1$_{(1,1)}$	23$_{(2,6)}$	pierre$_{(1,1)\ (2,6)}$
8,5$_{(1,2)}$	18$_{(2,3)}$	paul$_{(1,2)\ (2,3)}$
9,2$_{(1,4)}$	18$_{(2,4)}$	martin$_{(1,4)\ (2,4)}$
9,2$_{(1,5)}$	21$_{(2,5)}$	jean$_{(1,5)\ (2,5)}$

Final View

score	age	name
9,2$_{(1,5)}$	21$_{(2,5)}$	jean$_{(1,5)\ (2,5)}$
9,2$_{(1,4)}$	18$_{(2,4)}$	martin$_{(1,4)\ (2,4)}$
8,5$_{(1,2)}$	18$_{(2,3)}$	paul$_{(1,2)\ (2,3)}$
7,1$_{(1,1)}$	23$_{(2,6)}$	pierre$_{(1,1)\ (2,6)}$

Rule 1 (S1)

score	name
9,2$_{(1,5)}$	jean$_{(1,5)\ (2,5)}$
9,2$_{(1,4)}$	martin$_{(1,4)\ (2,4)}$
8,5$_{(1,2)}$	paul$_{(1,2)\ (2,3)}$
7,1$_{(1,1)}$	pierre$_{(1,1)\ (2,6)}$

Rule 2 (S1)

score	name
9,2$_{(1,5)}$	jean$_{(1,5)\ (2,5)}$
9,2$_{(1,4)}$	martin$_{(1,4)\ (2,4)}$
8,5$_{(1,2)}$	paul$_{(1,2)\ (2,3)}$
7,1$_{(1,1)}$	pierre$_{(1,1)\ (2,6)}$

Rule 3 (S1)

name	score
jean$_{(1,5)\ (2,5)}$	9,2$_{(1,5)}$
martin$_{(1,4)\ (2,4)}$	9,2$_{(1,4)}$
paul$_{(1,2)\ (2,3)}$	8,5$_{(1,2)}$
pierre$_{(1,1)\ (2,6)}$	7,1$_{(1,1)}$

Rule 4 (S1)

S1 recomputed

name	score
pierre$_{(1,1)\ (2,6)}$	7,1$_{(1,1)}$
paul$_{(1,2)\ (2,3)}$	8,5$_{(1,2)}$
martin$_{(1,4)\ (2,4)}$	9,2$_{(1,4)}$
jean$_{(1,5)\ (2,5)}$	9,2$_{(1,5)}$

Fig. 6. Computing a view and reconstructing a source from the view

- *Rule 2:* Values are grouped according to their tuple identifiers (in this example, this step does not change the output of Rule 1 above).
- *Rule 3:* Columns are permuted to match their original position.
- *Rule 4:* XTuples are ordered according their tuple ids.

Finally, the useful part of S_1 has been reconstructed.

3.3 Update Detection

In this subsection, we use the same notation as in the previous sections, and we suppose that the wrapper associated to the source σ detects a change in σ.

As explained earlier, the wrapper sends the updated content of σ, without any further information. Consequently, the XSource operator is applied and then, based on the XAlgebra expression V, an XRelation Y_σ associated to this new version of the data source can be computed.

In order to detect which updates have to be performed on the XRelation X, $v_\sigma^{-1}(X)$ is computed using the previous reconstruction rules. Then, we use the *diff* operator on $v_\sigma^{-1}(X)$ and Y_σ to identify the differences between these two XRelations.

The *diff* operator, introduced in [8], is an algorithm that has been initially designed for comparing lines of two files. In our setting, the result of the algorithm is a set of update instructions, that an have one of the following three forms:

$$delete(pos),\ insert(pos, new) \text{ or } replace(pos, new),$$

where *pos* is a row number indicating the row that has to be deleted, inserted or modified, respectively, and where *new* is the new tuple to be taken into account for the insertion or the modification.

The next section describes for each type of update and for every relational XOperator (projection, restriction, join, union, intersection), how to update the view.

In Figure 7, we show how, starting from the XRelation X, and applying the reconstruction rules as explained in the previous subsection (also shown in Figure 6), we can get the update notification by using the *diff* operator. Then, by applying these updates to the XRelation X as described in next section, we get the updated view.

4 Computation of the XQuery Update Request

Let X be the XRelation associated to the view defined by an XAlgebra expression V, and let us assume a change on source σ, specified as indicated in the previous section. In this section, we show how to update X, for each type of update (deletion, insertion or modification), and in the case where only one of the operators of projection, restriction, join, union or intersection appears in V.

We note that, in the case of insertion or modification, if in the XPaths that define the XTuple *new*, at least one mandatory XPath is missing, then no update has to be performed in X. Intuitively speaking, this is so because values over non mandatory XPaths have no impact in the computation of X. In what follows, we assume that all mandatory XPaths have a corresponding value in the XTuple *new* involved in the insertion or modification.

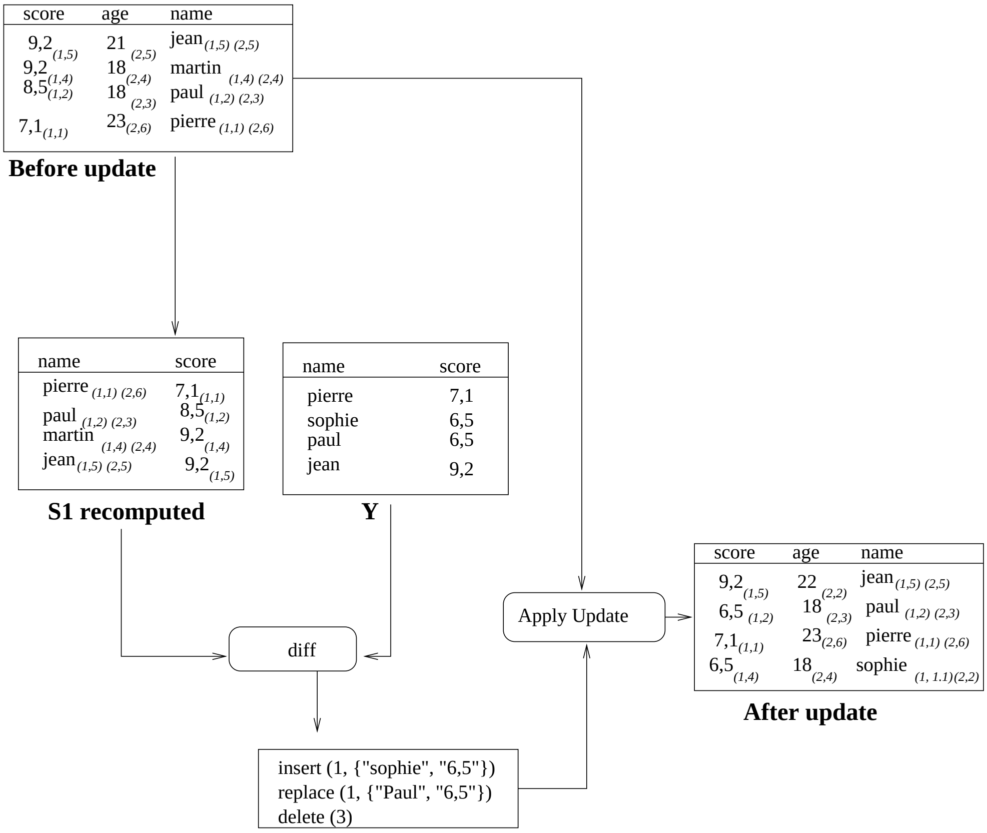

score age name
9,2 (1,5) 21 (2,5) jean (1,5) (2,5)
9,2 (1,4) 18 (2,4) martin (1,4) (2,4)
8,5 (1,2) 18 (2,3) paul (1,2) (2,3)
7,1 (1,1) 23 (2,6) pierre (1,1) (2,6)
Before update
name score
pierre (1,1) (2,6) 7,1 (1,1)
paul (1,2) (2,3) 8,5 (1,2)
martin (1,4) (2,4) 9,2 (1,4)
jean (1,5) (2,5) 9,2 (1,5)
S1 recomputed
name score
pierre 7,1
sophie 6,5
paul 6,5
jean 9,2
Y
diff
insert (1, {"sophie", "6,5"})
replace (1, {"Paul", "6,5"})
delete (3)
Apply Update
score age name
9,2 (1,5) 22 (2,2) jean (1,5) (2,5)
6,5 (1,2) 18 (2,3) paul (1,2) (2,3)
7,1 (1,1) 23 (2,6) pierre (1,1) (2,6)
6,5 (1,4) 18 (2,4) sophie (1, 1.1)(2,2)
After update

Fig. 7. Update notification and applying updates

4.1 The Case of a Deletion: *delete*(*pos*)

In this case, let y_{XTID} be the XTuple in $v_\sigma^{-1}(X)$ such that $row(y, \sigma) = pos$, and let i be such that $xtid = (\sigma, i)$ belongs to $XTID$.

Then, in the final XRelation view X, all XTuples for which $xtid$ appears among their associated XTIDs must be deleted from X. In other words, all XTuples

$$x = (c^1_{XTID_1} ... c^k_{XTID_k}) \text{ where } (\sigma, i) \in XTID_1 \cup ... \cup XTID_k$$

are deleted from X, and this applies for any of the XOperators projection, restriction, join, union and intersection.

4.2 The Case of an Insertion: *insert*(*pos*, *new*)

In this case, the XTuple new must be inserted at row pos in $v_\sigma^{-1}(X)$ and a new XTID of the form $xtid_{new} = (\sigma, l)$ is generated. Notice that, as ordering is important for future reconstruction (see Rule 4), the identifier l must be chosen so as it reflects the order of the XTuples in Y_σ. Then, considering the XTuple *new* whose components are annotated by (σ, l), the following insertions are performed in X, according to which XOperator occurs in V.

Projection In the case of a projection, the projection of *new* over the XPaths in X is inserted into X.

Restriction In the case of a restriction, if *new* satisfies the restriction predicate then *new* is inserted into X, otherwise X remains unchanged.

Join In the case of a join between sources σ and σ', in order to compute the XTuples that have to be inserted into X, we calculate: $v_\sigma^{-1}(X) \bowtie new = Z$.

If $Z \neq \emptyset$, then the XTuples of Z must be inserted into X. Otherwise, that is if $Z = \emptyset$, then the source σ' must be queried and joined with *new* to get the set of XTuples to be inserted into X.

Union and Intersection Since we consider duplicates, the case of a union is treated as a restriction where the restriction predicate is *true*. On the other hand, the case of an intersection is treated as the case of a join.

Note In all cases above, the insertion into X is simply processed by appending the new XTuple, without any ordering consideration, unless an order-by operator appears in the definition of the view. In this case, the specified ordering is of course taken into account for the insertion.

4.3 The Case of a Modification: $replace(pos, new)$

As for deletions, let y_{XTID} be the XTuple in $v_\sigma^{-1}(X)$ such that $row(y, \sigma) = pos$, and let i be such that $xtid = (\sigma, i)$ belongs to $XTID$.

For every c_{XTID} in y that is changed by the update, the column of the corresponding tuple in X containing c can be retrieved by using the XTIDs in $XTID$ and by comparing the positions of the XPaths occurring in the *for* and *return* clauses of the XQuery request Q. Then, the following updates are performed, according to the operator in V.

Projection Since the XTuple and its columns to be modified are known, the modification can be performed accordingly.

Restriction As above, in the case of restriction, the modification can easily be performed, but only under the condition that the modified XTuple satisfies the restriction predicate. Otherwise, the corresponding XTuple is deleted from X.

Join The case of a join can be treated through a deletion followed by an insertion, that is we perform $delete(pos)$ and then $insert(pos, new)$.

Union and Intersection As for insertions, the case of a union is treated as a restriction where the restriction predicate is *true*, and the case of an intersection is treated as the case of a join.

Remark We would like to end this section by pointing out that, in our approach, the maintenance of materialized XML views does not require to query the sources except in specific cases of insertions or modifications in the presence of joins or intersections. We recall in this respect that accessing the sources as few as possible is an important issue, when dealing with the integration of web sources.

5 Conclusion and Future Works

We have presented an approach to maintain materialized XML views by means of reconstruction of the sources, based on the content of the view *only.* The reconstruction aims to detect the changes in the sources, in the case where they cannot be duplicated at the wrapper level.

We are currently implementing of our method, focussing on the reconstruction module and on the management of XTIDs. Our future work will deal with other XOperations of the XAlgebra such as nesting and unnesting, and then the general case of any combinasion of XOperators will be addressed.

References

1. S. Abiteboul. On views and xml. In *Proc. ACM Symp. on Principles of Database System*, pages 1–9, 1999.
2. S. Abiteboul and al. Incremental maintenance for materialized views over semi structured data. In *Proc. Int'l Conf. on VLDB*, pages 38–49, 1998.
3. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semi-Structured Data. *Journal of the Digital Library*, 1 (1):68–88, april 1997.
4. G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. Technical report, INRIA - Columbia University, 2001.
5. T.-T. Dang-Ngoc and G. Gardarin. Federating heterogeneous data sources. In *in proc. of IASTED International Conference on Information and Knowledge Sharing (IKS 2003)*, pages 193–198, 2003.
6. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML, 1998.
7. M. EI-Sayed, L. Wang, L. Ding, and E.A. Rundsteiner. An algebraic approach for incremental maintenance of materialized xquery views. In *, in Proc. Of the 4TH intI Workshop on WIDM02*, 2002.
8. J.W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical report, Bell Laboratories, 1976.
9. D. Laurent, J. Lechtenborger, N. Spyratos, and G. Vossen. Monotonic complements for independent data warehouses. *VLDB*, 10(4):295–315, 2001.
10. W3C. An XML Query Language (XQuery 1.0). Technical report, available at http://www.w3.org/TR/xquery/, 2001.
11. G. Wiederhold. Mediators in the Architecture of Future Information Systems. *Computer*, 25(3):38–49, March 1992.