Incremental Method for XML View Maintenance in Case of Non Monitored Data Sources

Zohra Bellahsene, Xavier Baril

Incremental Method for XML View Maintenance in Case of Non Monitored Data Sources

Xavier Baril and Zohra Bellahsène

LIRMM - UMR 5506 CNRS / Université Montpellier 2,
161 Rue Ada - F-34392 Montpellier Cedex 5
{baril, bella}@lirmm.fr

Abstract. In this paper, we are dealing with the topic of view maintenance which consists of maintaining materialized views in response to data modifications on the data sources. We propose an incremental method to maintain XML views. This is achieved by defining first how to store XML views, which may be obtained over different data sources, in a relational DBMS. The identifiers used to store the view definition (in particular mapping patterns, unions and joins) allow the definition of the incremental method in the sense that the materialization of the view does not require re-computing all stored data to maintain XML views.

1 Introduction

To provide data access in large scale and/or dynamic environments with autonomous data sources, pertinent data are often collected and stored in a redundant way using a data warehouse. At abstract level, a data warehouse can be defined as set of materialized views. An important feature in a data warehouse is taking changes arising on the data sources into account. In this paper, we are dealing with the topic of view maintenance which consists of maintaining the materialized views in response to data modifications on the data sources. More precisely, we are interested in maintaining XML views that are stored in relational DBMS.

The main contribution of this paper is how to specify source patterns, and how to maintain materialized views of such source patterns. This is achieved by defining first how to store XML views (which may be obtained over different data sources) in a relational DBMS. We have designed a method using a relational DBMS for storing XML views. The originality of this method lies in the use of multi view graph, which allows representing common sub expressions between different views in order to reduce the cost of the view maintenance. This means that each view is not materialized as a whole part but as a set of fragments. Work that we present in this article was implemented like a functionality of the prototype DAWAX [2].

This paper is organized as follows. In Section 2, we present our view model for integrating XML data. The related view specification language is described in Section 3. Our storage method is presented in Section 4. The algorithms of view maintenance are presented in Section 5. Section 6 presents an overview of related work and Section 7 contains the conclusion and future work.

J. Wiedermann et al. (Eds.): SOFSEM 2006, LNCS 3831, pp. 148 – 157, 2006.

2 The View Model VIMIX

We have designed the VIMIX (VIew Model for Integration of XML sources) to integrate XML data sources. Due to the lack of space, we will not present the whole view model [2]. XML data are represented by a graph having three types of nodes (element, attribute and text). Moreover, operations were defined to handle the nodes of the graph (navigation and treatment of strings). The integration process consists in: (i) specifying the data to be extracted from the sources by defining patterns on them (`source-pattern`), (ii) reorganizing the views data by using relational like operations: `union` and `join` and (iii) specifying the result form of the XML views.

Our view specification language is based on pattern-matching: the data of the sources are related to variables which are declared in a pattern describing the source. The definition of the variables is done using a mechanism of research axes like in XPath for location steps in an XML document.

```
<<source-pattern name="sp_authors_biblio" source="biblio">
<search-axis function="children">
    <source-node reg-expression="author" type="element">
        <search-axis function="children">
            <source-node reg-expression="firstname"
                     type="element"
                     bindto="fname">
            </source-node>
            <source-node reg-expression="lastname"
                     type="element"
                     bindto="lname">
            </source-node>
      </search-axis>
    </source-node>
</search-axis>
</source-pattern>
```

Fig. 1. VIMIX Source Pattern

Figure 1 gives an example of a VIMIX source pattern, which retrieves first and `lastname` of authors. This pattern is named `sp_authors_biblio` and is defined over the data source `biblio`. The first element `search-axis` is the principal research axis of the pattern, meaning that one applies the function `children` starting from the root of the document. This function returns a set of nodes, which will be filtered starting from the contents of this axis: the element `source-node` specifies that the nodes must match the regular expression `author` and be of type `element`. The specification of this source node is supplemented by a research axis specifying that the nodes must have two subelements `firstname` and `lastname`. These subelements are bound to variables (attribute `bindto`).

In our approach, the data extracted by patterns are stored in relational tables. Therefore, this allows makes it possible to restructuring the data by using relational algebra. We namely adapted two operations of them which are relevant for data integration: union and join. The operation of union that we defined takes as input several operands, which result from patterns, either union or join. The result is stored in a relational table whose attributes are computed as the union of the attributes of the operand tables. This table values is built as the union of the tuples of the sources and

by assigning NULL value to the attributes which do not exist at a source. Unlike the relational `union` ours may be applied when the sources have different schemas. Moreover, our operation of union allows to filter the data and to solve conflicts of identity by eliminating the duplicates coming from different sources. To solve the conflicts we use a mechanism specifying a priority source.

The join operation allows to "cross" information coming from two sources, patterns, union or other join. Its result is stored in a relational table, whose columns are computed as the union of the attributes of the sources. The join predicate is evaluated by applying a function which returns the textual representation of the nodes.

3 Specification of the Views

A VIMIX view is defined as a tuple including the following properties: (i) its name, (ii) a source pattern, union or join which contains the data to populate the view (iii) a pattern which describes the result structure using a tree. This tree has three types of nodes: element, attribute and expression. The expression nodes allow to populate the view result. We have defined conversion functions of types to facilitate this task. Finally, *aggregation* functions and *group by* expressions may be also used in the view result specification.

Figure 2 describes the view computing for each author: its name, the number of books which he wrote, the average price and titles of these books.

The name view is `v_books` is defined over the data source `j_books_lirmm`. The tree specifying the result of the view is the element `source-node` and is structured as follows: each element `author` will have three attributes: name, number of books and their average price. The titles of the books of an author are sub-elements.

```
<view name="v_books"
     source="j_ books "
     order-by="author"
     group-by="author">
<result-node type="element" value="author">
  <result-node type="attribute" value="name">
    <result-node type="expression" value="text(author)" />
  </result-node>
  <result-node type="attribute" value="nb- books">
    <result-node type="expression" value="count()" />
  </result-node>
  <result-node type="attribute" value="Avrage-price">
    <result-node type="expression" value="avg(float(price))" />
  </result-node>
  <result-node type="element" value="book">
    <result-node type="expression" value="text(title)" />
    </result-node>
</result-node>
</view>
```

Fig. 2. Integrated view of the books for each author

4 Storage of VIMIX Views

Our storage architecture avoids redundancy of XML data in the warehouse since we separate the data storage of that of the metadata (i.e. the mappings).

4.1 Generic Schema of XML Data Storage

The generic schema which we utilize to store the XML data is described in Figure 3. The generic schema is designed to store nodes coming from the sources, without storing all the data of these sources. For each source (or document) one needs simply to know his identifier and his URL, without being concerned with root of the document which is not necessarily stored in the data warehouse. The table `Document` contains the `urls` of the data sources.

The tables `Element` and `Attribute` are dictionaries of the elements and attributes. They contain a code identifying the element or the attribute like its name. The dictionaries accelerate the queries involving an element name or attribute.

The table `XmlNode` stores the data nodes. Each node has an identifier: `nodeID`. Our data model considers three types of nodes: Element, Attribute and Text which respectively represents elements, attributes and text in a XML document. The columns `elemID` and `attID` provide the type of a node of the table. If `elemID` is not NULL, the node is of type element and `elemID` indicates its name. If `attID` is not NULL, the node is of type attribute and `attID` indicates its name. The value of the attribute is stored in the column `value`. Lastly, if `elemID` and `attID` have both NULL value, the node is of type text and `value` contains the string.

The table `Children` contains the composition links between the nodes of the stored documents and has as attributes:

- `parentID` contains the identifier of the parent node,
- `childID` contains the identifier of the child node,
- `rank` contains the row of the son.

Table	Column	Role
Document	docID, url	Dict. of sources
Element	elemID, name	Dict. of elements
Attribute	**attID**, name	Dict. of attributes
XmlNode	nodeID, *elemID*, *attID*, value, *docID*	XML Nodes
Children	*ParentID*, *childID*, rank	Links
Descendants	*ParentId*, *childId*, rank	Links

Fig. 3. The generic schema for storing the XML data

The table `Descendants` contains the descendance links between the nodes. These links can be computed from the children table, using the fact that descendants are children and children of children and so on. However, SQL doesn't offer a way to compute children at any level; it's why we prefer storing them in the `Descendants` table, which is made up of the following columns:

- `parentID` contains the identifier of the parent node,
- `childID` contains the identifier of the children node,
- `rank` initially contains the rank of the children by considering an in-depth traversal. The identifiers `parentID` and `childID` are foreign keys of the column `nodeID` in the table `XmlNode`.

The tables Element and `Attribute` contain the metadata of the elements and the attributes. The table `XmlNode` contains all the nodes of the source which are stored in the warehouse. The table `Children` contains the composition links between the nodes of XML documents in the warehouse. It involves the following columns:

- `parentID` contains the identifier of the parent node,
- `childID` contains the identifier of the child node,
- `rank` contains the rank of the child node.

4.2 Storage of the Mappings

The key idea of our approach is: rather than materializing complete views, it is better to materialize portions (fragments) of views, to allow reuse and improved incremental maintenance. Obviously, it allows reducing space storage. Figure 4 describes the graph of mapping between the data sources and the views. The nodes of this graph are tables names storing the data specified by patterns on the sources, union or join.

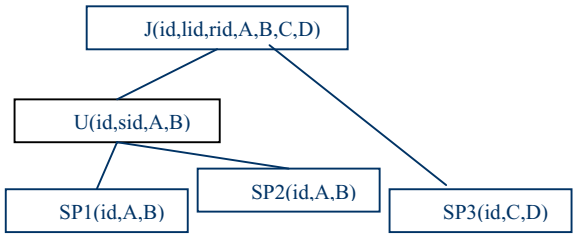

Fig. 4. The Graph of Mappings

The table schema of the pattern *SP*: *SP(id, variables$_{sp}$)* where *SP* is the name of the pattern, *id* is a numerical identifier of integer type. This identifier value corresponds to the order in the table defined by the extraction of the data. Finally, *variables$_{sp}$* describe the pattern variables.

The schema of the union table is: *U(id; sid; variables$_u$)* where *U* is the name of the union and *id* is numerical identifier of real type, whose semantics of the integer and decimal parts is defined as follows. The integer part contains the sequence number of the source whose the tuple comes from data of the union. This sequence number is obtained by the position of the source in the list of the sources of the union.

The decimal part contains the identifier of the source whose data come from. If this identifier is a real number, it is transformed into an integer number by concatenating the integer and decimal parts. For example, if the identifier of a tuple of the source to be inserted in the union is 2123, the transformation will give the result 2123. Moreover, the decimal part of the source at the position *i* is preceded by n-k zeros, with *n* and *k* are defined as follows. 10^n is the minimal upper limit of the number of sources of the union and 10^k is the minimal upper limit of *i*. For example, if a union is defined over a list of 11 sources, the minimal upper limit of the number of sources of the form 10^n is 10^2, therefore one has n = 2. The decimal part of the identifiers of the tuples coming from the sources at the position *i*, for $i \in [1..9]$ will be preceded by one zero, because the minimal upper limit of *i,* is 10^1, say k = 1, one thus has n - k = 1. The decimal part of the identifiers of the tuples coming from the sources at the position *i,* for $i \in [10..11]$ will be preceded by no zero, because the minimal upper limit of i is 10^2, say k = 2, one thus has n- k = 0. This method allows to preserve the order between the tuples of the tables containing the sources of the union. In this way, this identifier preserves the order defined during the retrieval of the data.

The column *sid* is the identifier of the inserted tuple, in the source from which it comes. This column is of real type, because it must contain the identifiers of the data sources of the union, which can be of type integer or real. Finally, $variables_u$ is the set of the variables specified by the union *U*. Each variable references a data node stored in the generic schema.

The schema of the join mapping table is: $J(id; lid; rid; variable_j)$ where J is the name of the join; *id i* is a numerical identifier of real type, whose semantics of the integer and decimal parts is defined as follows. The integer part contains the identifier of the data coming from the left part of the join. If this identifier is a real number, it is transformed into an integer number. The decimal part contains the identifier of the data coming from the right part of the join. If this identifier is a real number, it is transformed into an integer number. *lid* is an identifier of real type, which references the identifier of the tuple used to calculate the left part of the join. This column is of real type, because it must contain the identifiers of the two sources of the join which can be of type integer or real. *rid* is also identifier of real type, which references the identifier of the tuple used to calculate the right part of the join. This identifier is of real type, because it must contain the identifiers of the two sources of the join which can be of type integer or real. $variables_j$ are the variables specified by the join. Each variable references a data node stored in the generic schema.

5 Maintenance of VIMIX Views

5.1 Refreshing a *Pattern* XML View from a Data Source

Data sources available on the Web or produced by various applications cannot easily be monitored. Therefore, the smallest operation of refreshing the data stored in the warehouse is thus the one of pattern matching expression defined over one source. Our method is incremental because it does not require to re-compute the entire view but only the view fragment defined over the source that has been updated.

The algorithm1 runs as follows. For each pattern *sp* defined over this source, the content of the table `Tsp` is copied into table `Tdeletes` then the table `Tsp` is cleared. This process of data extraction is repeated to populate again the table `Tsp`. At this stage, the data of a pattern on the source are updated. It is then necessary to propagate the update to the related mappings: union and join. For that, the function `refresh-mappings` is executed for all the tables which are parents nodes of in the graph of mappings. Lastly, one removes the XML data coming from the source, which are stored in the tables of the generic schema. The deletion cannot be made earlier, because the data of the old mappings could be necessary for the maintenance.

Algorithm 1. `Refresh-Pattern` *(s)*
Result : refreshing data from a source *s*
foreach *pattern sp related to the source s* **do**
copy the table $\mathcal{T}sp$ in $\mathcal{T}deletes$;
clear the table $\mathcal{T}sp$;
extract the data from the source *s* to populate $\mathcal{T}sp$;
foreach $\mathcal{T}parent$ *parent of* $\mathcal{T}sp$ *in the graph of mappings* **do**
`refresh-mappings(`$\mathcal{T}parent$`,` *sp*`,` $\mathcal{T}deletes$`,` $\mathcal{T}sp$`)` ;
end
end
delete the data coming from the source *s*;

5.2 Refreshing the Union and the Join Views

The algorithm 2 presents the strategy of updating a table representing a union or a join in the graph of mappings. The function *refresh-mappings* has four parameters: (i) the table containing the data of a union or a join which must be updated, (ii) the *child* source representing the pattern, the union or the join which were updated and (iii) the table $\mathcal{T}delete$ containing the deleted tuples and the table $\mathcal{T}insert$ containing the added tuples.. This Algorithm is incremental, because it uses the data removed and added to the updated source to carry out only the necessary modifications. The updates are propagated to the parent tables of the graph of mappings. This propagation is carried out by a recursive call of the function. The condition is carried out by the tables which do not have a parent, which is ensured by the fact that the graph of mappings is acyclic.

Algorithm 2. `refresh-mappings(`$\mathcal{T}s$`,` *child*`,` $\mathcal{T}deleteschild$`,` $\mathcal{T}insertchild$`)`
Result : Refreshing the mappings of an union or a join $\mathcal{T}s$
computes in $\mathcal{T}deletes$ the tuples to be deleted in $\mathcal{T}s$ (by using $\mathcal{T}deleteschild$) ;
compute in $\mathcal{T}inserts$ the tuples to be added in $\mathcal{T}s$ (by using $\mathcal{T}insertchild$) ;
delete in $\mathcal{T}s$ the tuples of $\mathcal{T}deletes$;
add in $\mathcal{T}s$ the tuples of $\mathcal{T}inserts$;
foreach $\mathcal{T}parent$ *parent of* $\mathcal{T}s$ *in the graph of mappings* **do**
`refresh-mappings(`$\mathcal{T}parent$`,` *s*`,` $\mathcal{T}deletes$`,` $\mathcal{T}inserts$`)`;
end

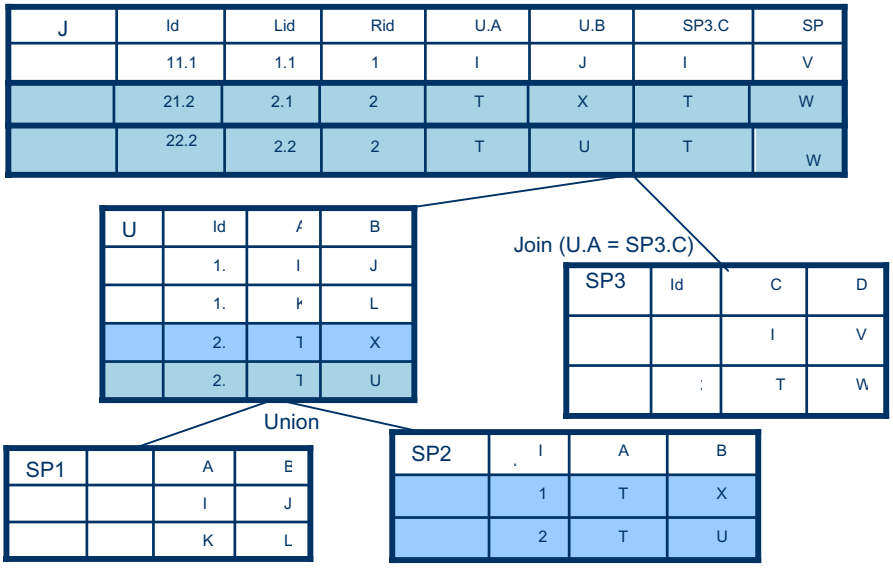

Fig. 5. Illustration of the update propagation

Figure 5 illustrates the update propagation. The graph of mappings used on this example consists of three patterns on three sources, noted `SP1`, `SP2` and `SP3`. The variable `u` contains the union of the data of the patterns of `SP1` and `SP2`. Finally, `j` contains the join of the data of `u` and `SP3`. To facilitate the legibility of the example, the patterns store the values of the elements XML corresponding to the instantiation of the variables rather than the references to these elements (which should be stored by using the generic schema). This example illustrates the propagation of updates when the source of the pattern `SP2` is modified. The dashed tuples of the mappings corresponds to those which have been updated. As it is shown in this figure, the maintenance is incremental.

6 Related Work

There are mainly two approaches for storing XML data: the *flat storage* and the *meta-modelling*. In Flat storage approach, XML data are stored by using their textual form. It is the simplest method to implement, because it is sufficient to use a files system, or the type `BLOB` of a DBMS to store the documents in a database. This method is very efficient when one tries to find the whole document or large contiguous parts of an XML document. The main disadvantage of this method is the need for parsing the document to discover its structure: that results into slow down the query processing.

In meta-modelling approach XML data are stored in the target DBMS by using *transformation rules* [3, 6]. This method is very efficient when queries are based on the structure of the stored data. Indeed, the data were already analyzed at the time of their transformation to be stored in the target DBMS. The principal disadvantage of

this method lies in the transformations which are necessary to store and rebuild the data of the XML documents. When the XML documents to be stored are bulky, this phase of transformation is costly.

Inside the meta-modelling, there is two approaches:.(i) *generic s*chema which can be used for any XML data instance and (ii) The schemas *depending of the data* which must be generated for each data instance to store. Intuitively, the use of a generic schema can be simpler when the data to be stored come from heterogeneous documents. Indeed, if one uses a schema depending of the data, it would be necessary to generate a schema of storage for each document.

There is also a third family of solutions which combines the two previous approaches: the hybrid approaches. There are two ways of doing it.

The first one is redundant; it consists in storing the data by using the two methods. That allows a fast querying of the documents thus stored, but naturally the updates are slowed down and storage spaces it is far from being optimal because all the data are duplicated. The second method consists in using a mixed approach: starting from a certain level of granularity called threshold, the data are stored as flat whereas with the top of this level they are stored in a DBMS by using the meta-modelling [7], [8].

The maintenance of XML views is a recent problem which is currently studied. Early work was on semi-structured views had been considered for OEM data [1]: proposed an algorithm which calculates a set of queries used to propagate a source modification on a view. An index for accelerating the update of XML data was proposed in [4]: APIX. This work has been done in the case of monitored data sources. Another algorithm was proposed to calculate the changes between two XML documents [5]. XML views can be used like interfaces to update relational sources [3], [9].

The major differences between the related work and our approach are the following. Related work is based on the full materialization of the view therefore view maintenance is performed view per view. Our contribution is the first one dealing with the XML view maintenance for fragment-based approach. In our approach, view maintenance is performed regarding all the materialized views therefore it encourages the reuse of materialized fragments.

7 Conclusion and Future Work

In this paper, we present a storage method and algorithms for the maintenance of XML views stored in a relational DBMS. We have designed a storage method which separates the storage of XML data from that of the metadata describing the mappings. The identifiers used to store the view definition (in particular mapping patterns, unions and joins) allow incremental maintenance in the sense that the materialization of the view does not require re-computing all stored data to maintain XML views. The other originality of this method lies in the use of multi view graph, which allows representing common sub expressions between different views. Consequently, this allows reducing the storage space and the view maintenance time. Querying XML views stored in a relational DBMS requires a phase of rebuilding them. As future work, we are planning to develop a cache strategy to store the XML data which are frequently rebuilt.

References

1. Abiteboul, S., Hugh, J.M., Rys, M., Vassalos, V., Wiener, J.: Incremental Maintenance for Materialized Views over Semistructured Data. Intern. Conf. on Very Large Databases (1998)
2. Baril, X.: Un modèle de vues pour l'intégration de sources de données XML : VIMIX. PhD thesis, Université Montpellier II, December (2003)
3. Braganholo, V., Davidson, S., Heuser, C.: On the Updatability of XML Views over Relational databases. WebDB'2003, San Diego, California (2003)
4. Chen, L., Rundensteiner, E.A.: APIX: An Efficient Approach to Maintain Web Views. Technical Report WPI-CS-TR-00-08, Worcester Polytechnic Institute, Dept. of Computer Science (2000)
5. Cobena, G., Abiteboul, S., Marian, A.: Detecting Changes in XML Documents. In: Proceedings of the 18th International Conference on Data Engineering, ICDE'2002, San José, California, IEEE Computer Society (2002)
6. Florescu, D., Kossmann, D.: Storing and Querying XML Data using an RDMBS. IEEE Data Engineering Bulletin **22** (1999) 27–34
7. Kanne, C., Moerkotte, G.: Efficient Storage of XML data. Technical Report 899, Mannheim University (1999)
8. Kanne, C.C., Moerkotte, G.: Efficient Storage of XML Data. In: Proceedings of the 16th International Conference on Data Engineering, ICDE'2000, San Diego, California, IEEE Computer Society (2000)
9. Katica D., El-Sayed, M., Rundensteiner, E.A.: Order-Sensitive View Maintenance of Materialized XQuery Views. ER 2003: 144-157