EFFICIENT MECHANISM FOR HANDLING MATERIALIZED XML VIEWS

Jessica Zheng[‡], Anthony Lo[‡], Tansel Özyer[‡], Reda Alhajj[‡,⊗]
[‡]*Department of Computer Science, University of Calgary, Calgary, Alberta, Canada*

[⊗]*Department of Computer Science, Global University, Beirut, Lebanon*

Keywords: Materialized views, XML, deferred update, query performance, object-oriented database.

Abstract: Materialized views provide an effective and efficient mechanism to improve query performance. The necessity to keep consistency between materialized views and the underlying data raises the problem of when and how to update views efficiently. This paper addresses the issue of deferred incremental update on materialized XML view. The proposed approach mainly extends our previous work on materialized object-oriented views. The overlap between XML and the object-oriented paradigm has been the main driving motivation to conduct the study described in this paper.

1 INTRODUCTION

XML, a mark-up language, is widely used for publishing and exchanging data on the Web. To increase the performance of frequently requested queries against XML documents, the queries can be defined as XML materialized views to be used efficiently later. This paper discusses how to maintain materialized XML views.

When real data in XML documents are modified by insertion, update and deletion, XML views derived from the modified XML documents need to be updated so that they are consistent with the underlying data. The update of views could be done immediately after the update of the XML documents or deferred until the view is accessed.

Deferred update means that the view update occurs at the time the view is requested. The objective of the deferred update is to improve the performance of the database system. Only modifications that may affect a view will be considered when doing deferred update on the view. Let's take the Library database system as an example. If a book is added into the database, before relevant view is to be updated, the book is deleted from database. This book needs not to be considered at all when updating related views in a deferred way.

In this paper, an approach to incrementally update materialized XML view in a deferred way is proposed. The approach consists of two parts:

1. Modification Information Schema (MIS), which is defined to keep track of modifications done to XML data. It contains four parts: a) element nodes: XML schema is represented in a hierarchical structure. An element node refers to a node in the XML schema; b) instances: an instance refers to an element of the XML document; c) modification list: each element node has a modification list, which records modifications done to instances of element node, including insertion, update and deletion; d) view: a view is a virtual element node that contains information about an XML view.
2. Update algorithm, which describes the process of how XML views are updated.

The remainder of the paper is organized as follows. Section 2 discusses related works. Section 3 describes the proposed approach in detail. Section 4 concludes the paper and provides future research.

2 RELATED WORK

XML has received considerable attention. There have already been developed prototype view implementations. Papers on implemented views address also the area of semistructured data, closely related to XML data. These contributions show trends, but the subject still requires much more research and development. Some approaches that have influenced XML views are discussed below.

Baru (1999) discusses a mapping between XML schema and relational schema. Informally, the mapping from the XML schema to the relational

Zheng J., Lo A., Özyer T. and Alhajj R. (2006).
EFFICIENT MECHANISM FOR HANDLING MATERIALIZED XML VIEWS.
In *Proceedings of the Eighth International Conference on Enterprise Information Systems - DISI*, pages 151-156
DOI: 10.5220/0002451501510156

schema involves the following steps. The structure of the XML schema is a hierarchical structure. Each element in the XML schema is assigned a unique ID. A table is created for each element. The ID of the element is set as the primary key of the table. To represent the relationship between parent element and child element, the primary key of the parent element is set as foreign key in the child element.

The technique to derive XML view schema from a given relational schema is more complicated and user input is required. The process comprises two steps. The first step is to transform the relational schema into a directed graph. The second step is to use graph-processing technique to find candidate XML view schemas. The process requires user guidance to make some decisions. For example, the graph may have cycles and the user decides on the root element and the sub-element to break the cycle.

Chen, et al (2002) propose a technique to design XML views, which are guaranteed to be valid. A valid XML view is an XML view that does not violate the integrity constraint and the semantics of the original XML document. The approach comprises two parts:

- ORA-SS schema, which is built based on the XML document. The schema describes the tree structure of the XML document and the relationships among elements in the document.
- A set of rules to guide the design of the valid XML views. An XML view is designed by applying selection, projection, join and/or swap operation. The swap operation is to exchange the position of parent element and child element. When applying these operations to the design of a view some rules must remain valid in order to guarantee the validity of the XML view.

Braganholo, et al (2003) study the problem of updating relational databases through XML views. In other words, how to guarantee that translating an update on an XML view into a set of update on the underlying relations will not introduce additional updates to the XML view. Not all XML views over relational database are updateable. They focus on Nest-Last XML views and Nest-Last-Project-Select-Join XML views (NLPSJ view). NLPSJ XML is a special sub-set of Nest-Last XML.

A Nest-Last XML view is a view expressed in the nested relational algebra; the nest operator is the last operator to be applied. Update to Nest-Last XML views could be translated into a set of corresponding relational view updates. If the corresponding relational views are updateable, the Nest-Last XML view is updateable.

By updating relational databases through XML views, the approach hides the underlying relational databases from users. Because the problem of updating XML views is translated into updating relational view, we argue that the techniques from the relational model can be utilized.

Shah and Chirkova (2003) use materialized XML views to improve query performance over relational databases. To query XML data from relational databases, XML query is first translated into SQL query. Then the SQL query is executed. At last the result is translated back into XML document. To reduce the response time of a query, this approach selects frequently accessed data and translates them into XML document. When answering a XML query, the database first checks materialized XML views to determine whether they could be used to answer the query. If yes, the answer is extracted from the view directly. Otherwise the aforementioned steps are executed to get the result. This approach assumes the stored data in database do not change frequently; otherwise it would involve the overhead of updating XML views frequently.

To decide which data to select, a column called access count is added into the relation. Initially, access count is set as null. Each time a tuple is accessed by a query, the access count value for the tuple increments by one. When access count value reaches the predefined threshold value, the data in the tuple is added into XML views.

This approach could reduce the query response time if data in XML views is carefully selected. However, this approach can not guarantee that the result retrieved from XML views is correct since the data to be selected is determined only by the access count value, not by some filtering conditions that specify the XML view.

Kang and Lim (2002) discuss how to update XML views over relational databases in deferred way. XML view is update when the view is requested by an application. Each update to the underlying relational database will be recorded to the update log chronologically for later use by XML view update. Our approach presented in this paper has some common points with this approach. Both update XML views incrementally. However, the approach we adopted does not constrain the database engine to relational database only. Instead it could be applied to any database at the backend.

The approach that will be discussed in this paper utilizes some techniques from our previous work on materialized views (Alhajj and elnagar, 1999). The latter approach is a mechanism to incrementally update materialized object-oriented views over object-oriented database. The basic idea is a view is updated when it is requested. Modification information is recorded to be used when performing

view updating. The structure of an object-oriented database is hierarchical. Each node represents a class. A class may have inheritance relationship and composition relationship with other classes. Inheritance represents the relationship between parent class and child class. When values of some attributes of a class are derived from another class, the two classes have composition relationship.

A view is a virtual class. It is different from base class, which refers to an original class in the database. A view may be derived from base classes and other views. To update a view, it is necessary to consider all modification information that may affect the update of the view. Such information consist not only the modification done to the dependent base classes and virtual classes, but also the modification done on other base classes that have inheritance relationship and composition relationship with the dependent base classes. Each class maintains a modification list to keep track of related modifications.

Since the structure of XML databases has many things in common with the structure of object-oriented databases, our main argument for the approach proposed in this paper is the techniques proposed by Alhajj and Elnagar (1999) could be applied with some adjustment to update XML views.

```
<Library>
    <Holdings>
        <Books>
            <ID> 1 </ID>
            <title> C++ programming </title>
            <due date> Aril 23, 2004 </due date>
            <status> check out </status>
            <Authors>
                <name>John Walmart </name>
            </Authors>
        </book>
    </Holdings>
    <Members>
        <library card #> 2 </library card #>
        <name> Eva Jen </name>
        <phone> 345-456 </phone>
        <Address>
            <city> Calgary </city>
            <street>23 Ave NW</street>
        </Address>
    </Members>
</Library>
```

Figure 1: Example XML Document.

3 THE PROPOSED APPROACH FOR XML MATERIALIZED VIEWS

This section describes our model and the algorithm for deferred incremental update of materialized XML views. The running example XML document given in Figure 1 and its schema are to be used to illustrate the different aspects of our approach.

A basic element refers to elements that have basic XML data type, such as integer, string, etc. A basic element does not have any children. A complex element refers to elements that are composed by one or more basic or complex elements. For example, the elements Library, Books, and Members are all complex, while title in books and name in members are basic element.

3.1 Modification Information Schema

In our deferred update approach, we defined a Modification information schema (MIS) to be used for modelling the XML document and the modifications done to the XML document. The recorded information is used in the algorithm for deferred view update. There are two first class objects used in MIS: element node and view.

Element node: an element node in the MIS model is designed to store information about elements that are defined in the XML document. It keeps the following:

For each element node defined in the XML document, four pieces of information are kept in the corresponding MIS: ChildList, ReferenceList, InstanceList, and ModificationList.

A *ChildList* contains a list of the children of the element. A child of an element can either be a basic element, which has no child, or a complex element, which is composed from other element nodes.

If any child of the element references to other elements, the name of the former child element and the name of the referenced element(s) are stored in the *ReferenceList*.

In *InstanceList*, we store all instances that belong to the element. For each instance, its name, unique identifier, a child list, and a reference list are maintained.

A *ModificationList*(M_List) contains a list of Modification Tuples (M_Tuples). For each view that depends on the current element/view, there is one and only one modification tuple in the modification list. Each modification tuple contains three lists, which are the Insertion list, the Update list, and the Deletion list. An insertion list records all inserted instances of the element node. An update list records

all updated instances of the element node. A deletion list records all instances that are deleted.

View: for each view, four pieces of information are kept: DependingNodeList, FilteringCondition, InstanceList, and ModificationList.

A view is a virtual element node. It is composed by different elements or views. All elements and views which the current element depends on are stored in a *DependingNodeList*.

A view can also have filtering conditions. The conditions are defined together with the view. It allows users to focus on a subset of the instances in the underlying element. Instances from the target nodes and satisfying the filtering condition are said to be instances of the view; and these instances are stored in the InstanceList.

Since views can be nested, it is necessary to store modification information for each view dependent on the current one. This is done by using the already defined modification list.

The following examples are based on the example XML document shown in Figure-1 and its schema. Example of the Element nodes:

Books:
- child list ::= {ID, title status, due date, Authors}
- referenceList::={} o M_list::= M_tuple(view1)}

Authors:
- child list ::= {name}
- referencelist::= {Authors.name:Members.name}
- M_list ::= {M_tuple (view1)}

Example Views:

View1: find overdue books
- Depending node list ::= {Books}
- Filter condition: due date of the book < current date **AND** status = checked out
- Instance list::={} o M_list::={M_tuple(view2)}

View2: find members who have overdue books and who are living in Calgary
- Depending node list ::= {view1, Member }
- Filter condition: address = ”Calgary” **AND** due date of the holding < current date **AND** status = check out
- Instance list ::= {} o M_list ::= {}

3.2 The Algorithm for Deferred View Update

Deferred view update means that the view update is done when the view is requested by an application. The other well known modes of update are immediately update after every changes made to the underlying XML data, and periodical update is performed at designated time instances. The process of deferred update consists of the following steps:

Step-1: Each modification to the XML data, including insertion, updating, and deletion is recorded.

Step-2: When a view is requested, the recorded modification information is checked to find which modifications will affect the view.

Step-3: Generate update information needed by the view based on the modifications located in Step 3.

Step-4: Update the view based on the update information.

Next, we elaborate more on each of these steps:

Step-1: Record modification information: MIS defined in Section 3.1 is utilized to store modifications made to the XML data. Each element node maintains a modification list. In the modification list, M_tuple is created for each view that depends on the element. M_tuples are ordered in M_list. When a modification is made to the element node, the modification information is always stored in the last M_tuple in the M_list.

Not only modifications done to the dependent element node affect a view, but also the modifications made to all descendants of the dependent nodes affect the view as well. In order to maintain the modification information for the target view, all descendants of the dependent element nodes also need to create an M_tuple for each view depending on its parent. For example, in the previous example of element nodes, view1 is depends on the element Books. Therefore, in the M_list of Books, there is M_tuple for view1. Since Authors is a child of Books, there is also M_tuple for view1 in its M_list.

Step-2: Extract the relevant modification information: This step will find out all the modifications that may affect the update process of a XML view, which is from now on referred to as the target view. First, all descendents of the dependant element nodes are located. Second, relevant modifications done to the dependent element nodes and their descendents are retrieved. Since it is possible that the target view depends on views that depend on other views, the M_list of all views which the target view depends on directly or indirectly must be considered.

Consider the case when view2 is updated. Since view2 depends on view1 and Member, the M_list of both view1 and Members are considered. In addition, as view1 depends on Books, the M_list of Books needs to be considered as well.

To extract the modification information from the M_list of each element node relevant for the target view, the following process is executed:

- Locate the M_tuple created for the target view in the M_list.

- Merge the content of the target M_tuple and the M_tuples behind it. Since the M_list is ordered and new changes are appended at the end of the list. By merging the content of target M_tuple with the M_tuples behind it, all modifications that happened since the last update of the target view are obtained.
- Add content of the target M_tuple into its immediate predecessor. The content in the target M_tuple may be required for updating other views. Therefore, it is necessary to store it elsewhere in the list because the target M_tuple will be removed. Correctness of the model is maintained by storing it in the immediate predecessor.
- Empty the target M_tuple and move it to the end of the M_list. This means that the view has just been updated and no modification has happened since the target M_list is empty.

Step-3: This step filters the extracted modification information based on the filtering conditions of the target view. Each view has its own filtering method since the filter condition for each view may be different.

Step-4: Update the XML view based on the filtered modification information.

The actual algorithms to handle the above outlined process are *inheritanceModification* and *UpdateView*. The former algorithm is used to find modification information for a dependent element node and its children, with respect to the target view.

```
Algorithm: inheritanceModification
Input: view Vid, element-node Nid
Output: three instance list: I_list, U_list, D_list
Begin
   let I_list = insert list contains the inserted instances
   let U_list = update list contains the updated
instances
   let D_list = deletion list contains the deleted
instances
   let M_list(node) = modification list of element node
   let M_tuple(Vid) = M_tuple for view Vid
   Set I_list = U_list = D_list = {}
   child list=findChildren(Nid)
   // find all direct/indirect children of Nid
   for each node in child list {
    // extract modification of a view since its last
update
     extractModificationFromMTuple(M_Tuple of
current node, Vid)
   }
End
Method: extractModificationFromMTuple
Input: M_tuple and Vid
Output: I_list, U_list, D_list
Begin:
   let M_tuple(Vid) be at position k M_list(node)
   // find all the modification done to node since last
update
   // of Vid and add them into I_list, U_list and D_list
   while not end of M_list(node) {
        I_list += M_list[k].I_list
        U_list += M_list[k].U_list
        D_list += M_list[k].D_list
        k++
   } //end of while
   If M_tuple(Vid) has immediate predecessor
M_tuple(X) {
          Add content of M_tuple(Vid) into M_tuple(X)
          Empty the three lists in M_tuple(Vid)
          Move M_tuple (Vid) to the end of
M_list(node)
   }
End
```

The second algorithm, *updateView*, is a recursive function for updating the view. It first checks each dependent element node in the depending node list of the target view. If the dependent element node is an element node instead of a view, it uses the *inheritanceModificaiton* algorithm to extract all modification information of the dependent node and its children. If the dependent element node is a view, it updates this view first and extracts its modification information. After all the modification information is retrieved, that information is filtered. Finally, the target view is updated based on the filtered information.

```
Algorithm: updateView
Input: view Vid, depending node list of Vid
Output: the update version of view Vid
Begin
set I_list = U_list = D_list = {}
for each node in depending node list of Vid {
  if node is a element node {
 // find all the modification done to node since last
 // update of Vid and add them into I_list, U_list and
D_list
     find M_tuple(Vid) in M_list(node)
     if found {
 // inheritanceModification() return three lists
 // I-list(node), U-list(node), D-list(node)
      inheritanceModification()
      I_list += I-list(node)
      U_list += U-list(node)
      D_list += D-list(node)
      } else if not found {
 // recursively find all children of Nid including its
indirect
 // children e.g. grand child, grand-grand child
        child list = findChild(node)
        create M_tuple(Vid)
        add M_tuple(Vid) to the end of M_list(node)
        for each child node in child list {
```

```
            create M_tuple(Vid)
            add M_tuple(Vid) to the end of M_list(child
             node)
          }
        }
    } else if node is a view {
       updateView(node)
       find M_tuple(Vid) in M_list(node)
       if found {
    extractModificationFromMTuple(M_Tuple of current
                node, Vid)
       } else if not found {
            I_list += node's instance list
        create M_tuple(Vid) and add it to end of
                  M_list(node)
        }
    }//end of if node is a view
   }//end of for each
   I_list = I_list - D_list
   U_list = U_list - D_list
   I_list = I_list + U_list
    //filter modification information
   call Vid's filter(I_list, D_list)
   let M_tuple(X) is the last M_tuple in M_list(Vid)
   add the filtered modification information into
M_tuple(X)
   Vid's instance list = Vid's instance list - D_list
   Vid's instance lsit= Vid's instance list+ I_list
   End
```

4 CONCLUSIONS AND FUTURE WORK

This paper discussed an approach for deferred update of XML views. An XML view is updated only when it is requested. All modifications done to the underlying XML data is recorded in order for the system to update the view at a later time. MIS is developed to keep track of the modification information. MIS stores not only the modification information, but also the structure of the XML data and pointers to the XML data. This paper also shows how to use the information stored in MIS to update materialized XML views.

The approach assumes that XML views are derived from the source XML data, which conforms to the same XML schema. To apply the approach to XML views that are derived from heterogeneous XML data, we are currently considering the following problems: 1) XML view schema based on the set of source XML schemas; 2) filtering conditions for an XML view that spans multiple XML schemas; 3) reconstructing the query result based on changes to the XML view schema.

REFERENCES

Baru C., 1999. “XViews: XML Views of Relational Schemas,” *SDSC TR-1999-3, San Diego Supercomp. Centre, University of California- San Diego*.

Chen Y.B., Ling T.W., Lee M.L., 2002. “Designing Valid XML views,” *Proc. of ER*, London, UK.

Braganholo V.P., Davidson S.B., Heuser C.A., 2003. “On the Updatability of XML Views over Relational Databases,” *Proc. of WebDB*, San Diego.

Shah A., Chirkova R., 2003. “Improving Query Performance Using Materialized XML Views: A learning-Base Approach,” *Proc. of the International Workshop on XML Schema and Data Management*.

Kang H., Lim J., 2002. “Deferred Incremental Refresh of XML materialized Views,” *Proc. of CAISE*.

Alhajj R. and Elnagar A., 1999. “Incremental Materialization of Object-Oriented Views,” *Data & Knowledge Engineering*, Vol.29, pp.121-145.

Wang L. and Rundensteiner E.A., 2004. “On the Updatability of XML Views Published over Relational Data,” *Proc. of ER*.

Coox S., 2003. “XML Database Schema Evolution Axiomatization,” *Programming and Computer Software*, Vol.29, No.3, pp.140-146.

Gupta A., Mumick I.S., 1999. “Materialized views: techniques, implementations, and applications,” *MIT Press*, Cambridge, MA.

Lo A., Alhajj R. and Barker K., 2004. “Flexible User Interface for Converting Relational Data into XML,” *Proc. of FQAS,* Springer-Verlag, Lyon, France.

Shanmugasundaram J., et al, 2001. “Querying XML Views of Relational Data,” *Proc. of VLDB*.

Simanovsky A., 2004. “Evolution of Schema of XML Documents Stored in a Relational Database,” *Proc. of ACM Baltic DB&IS*, Riga, Latvia.

Wang B., Lo A., Alhajj R. and Barker K., 2004. “Converting Legacy Relational Database into XML Database through Reserve Engineering,” *Proc. of ICEIS,* Porto.

Abiteboul S., 1999. “On Views and XML,” *Proc. of PODS*, pp.1-9.

Abiteboul S., et al, 1997. “Views for Semistructured Data,” *Proc. of the Workshop on Management of Semistructured Data*, Tucson, Arizona.

Lacroix Z., 2001. “Retrieving and Extracting Web data with Search Views and an XML Engine,” *Proc. of the Workshop on Data Integration over the Web, in conjunction with CAiSE*, Switzerland.

Lahiri T., Abiteboul S., and Widom J., 1999. “Integrating Structured and Semistructured Data,” *Proc. of DBPL*.

Afrati F., Chirkova R., Gupta S., and Loftis C., 2005. “Designing and Using Views to Improve Performance of Aggregate Queries,” *Proc. of the International Conference on Database Systems for Advanced Applications*, Beijing, China.