Deferred Incremental Refresh of XML Materialized Views: Algorithms and Performance Evaluation*

Hyunchul Kang Hosang Sung ChanHo Moon

Dept. of Computer Science and Engineering
Chung-Ang University
Seoul, 156-756, Korea

hckang@cau.ac.kr {hssung,moonch}@dblab.cse.cau.ac.kr

Abstract

The view mechanism can provide the user with an appropriate portion of database through data filtering and integration. Views are often materialized for query performance improvement, and in that case, their consistency needs to be maintained against the updates of the underlying data. They can be either recomputed or incrementally refreshed by reflecting only the relevant updates. With the emergence of XML as the standard for data exchange on the Web, active research is under way for efficient storing and querying XML documents with the DBMS. In this paper, we investigate XML views, their materialization and incremental refresh. The object-relational DBMS is employed for storing XML documents and their materialized views, and the update log is used for deferred view refresh. The algorithms for checking a update's relevance to views and for generating the operations and data necessary for view refresh are proposed. The experimental results show that the proposed scheme outperforms recomputation of XML views.

Keywords: XML, materialized view, deferred incremental view refresh, semistructured data

1 Introduction

In database systems, the view concept has been a useful and effective mechanism in accessing and controlling data. It is related to many aspects of data management and database design. Among others, one of the most important applications of the view is information filtering and integration, functionality which is getting even more crucial for information processing in today's Web-based computing environment where vast amount of heterogeneous information proliferates every day.

Views are often materialized for query performance, requiring their consistency to be maintained against the updates of the underlying data (Gupta and Mumick 1999). Consistency maintenance can be done either by recomputing the view from the source data or by incrementally refreshing the outdated materialized view. The latter can be done either immediately after the source update occurs or in a deferred way.

* This work was supported by grant No. R01-2000-000-00272-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

Copyright © 2003, Australian Computer Society, Inc. This paper appeared at the Fourteenth Australasian Database Conference (ADC2003), Adelaide, Australia. Conferences in Research and Practice in Information Technology, Vol. 17. Xiaofang Zhou and Klaus-Dieter Schewe, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

Since XML emerged as a standard for data exchange on the Web, many research issues in XML data management have been investigated. The view concept is also useful for XML data, and active research is being conducted on it (Abiteboul 1999, Abiteboul et al. 1999, Cluet et al. 2001, Hristidis and Petropoulos 2002, Chen et al. 2002, Chen and Rundensteiner 2002, Quan et al. 2000, Chen and Rundensteiner 2000).

In this paper, we investigate the *XML materialized view* for fast retrieval of XML documents. An XML view against XML documents is defined by an XML query language (say, XQuery (Boag et al. 2002)) expression. Its materialization which is also an XML document is maintained with deferred incremental refresh.

The problem of incremental refresh of materialized views received much attention in relational database systems (Gupta and Mumick 1999). The same problem was investigated for the views over XML data (Quan et al. 2000, Chen and Rundensteiner 2000) and for those over semistructured data (Suciu 1996, Zhuge and Garcia-Molina 1998, Abiteboul et al. 1998) in the context of a semistructured DBMS such as Lore (McHugh et al. 1997).

In this paper, however, we explore a different direction. We investigate the problem for the case where the XML documents as well as their materialized views are stored in a relational DBMS (RDBMS) or an object-relational DBMS (ORDBMS) instead of the semistructured one. Since the traditional RDBMSs and the modern ORDBMSs are in dominantly wide use, storing and querying XML documents with them is of pragmatic importance and has attracted much attention (Florescu and Kossmann 1999a, Shanmugasundaram et al. 1999, Deutsch et al. 1999, Tian et al. 2002).

The rest of this paper is organized as follows: Section 2 surveys the related work. Section 3 deals with the issues involved in XML view materialization. They include (1) design of the object-relational database schema to store not just the base XML documents but the materialized views derived from them, and some other information necessary for view refresh (The ORDBMS is preferred to the RDBMS in our work because we need to have some table columns of *structured* and *collection* types.), (2) logging of updates to the base XML documents, and (3) the algorithms for deferred incremental refresh of the XML materialized view, the core of which is relevance checking between a update and a view. Section 4 presents the experimental results on performance. Finally, Section 5 gives some concluding remarks.

2 Related Work

When the XML documents are stored in the RDBMS, efficient table schema (XML to relational mapping) and XML-SQL translations are required. Given an XML query or update, it needs to be translated into the appropriate SQL expressions, and their result sets need to be tagged and returned in XML. These issues have been well addressed in recent research on storing XML data in the RDBMS (Shanmugasundaram et al. 1999, Florescu and Kossmann 1999b, Deutsch et al. 1999), on publishing relational or object-relational data as XML (Fernandez et al. 2000, Fernandez et al. 2001, Shanmugasundaram et al. 2000, Carey et al. 2000, Shanmugasundaram et al. 2001), and on XML update (Tatarinov et al. 2001). These issues are beyond the scope of this paper. We rather focus on the aspect of materialization and incremental refresh of XML views in the context of the ORDBMS.

In (Quan et al. 2000, Chen and Rundensteiner 2000), incremental refresh of the materialized views over the XML data was investigated. The XML sources are stored in the binary form of persistent DOMs, and the views are defined in a subset of XQL (Robie et al. 1998). The updates considered are the insertion/deletion of a segment of an XML tree and modification on the value of a leaf node of the XML tree. An auxiliary information structure called the aggregate path index(APIX) which holds the collection of qualified data objects with respect to the query pattern (Chen and Rundensteiner 2000) is used to check the updates' relevance to the view. The APIX is generated when the view is initially computed and maintained against the subsequent updates on the XML sources.

In (Suciu 1996), incremental refresh of the materialized views over the semistructured database of the rooted trees with labeled edges was investigated. The views considered are defined in UnQL (Buneman et al. 1995) without joins. The updates considered are the insertion of a tree to another one as a subtree of one of its nodes, and the replacement of a subtree with a new tree. When the update to the data source occurs, it is notified to the sites where the views derived from it reside, and the new subtree for insertion or replacement is transmitted. The view site then incrementally refreshes the view with the received subtree. This scheme does not have to access the data source for view refresh, and yet it does not support the join view nor the value modification of the data source.

In (Zhuge and Garcia-Molina 1998), incremental refresh of the materialized views over the graph-structured database was investigated. An example of such a database is the linked Web pages, and any database that can be modeled as a set of objects (nodes) with pointers (edges) is applicable. The views considered are the ones defined in an extended OQL (Cattell et al. 1994), and the materialized view is represented as a set of objects satisfying the view condition without links (i.e., edges) among them. The updates to data source considered include the edge insertion between two objects, the edge deletion, and the atomic object's value modification. The insertion/deletion of objects was not considered. When the update occurs, the queries against the data source are generated and executed to figure out which objects are needed to refresh the view. The retrieved objects are inserted to or deleted from the materialized view.

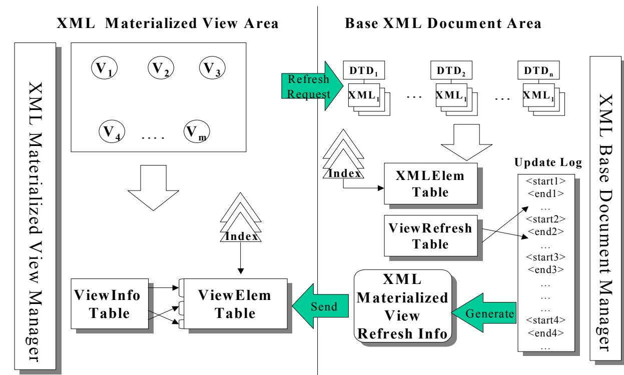


Figure 1. Management of XML Materialized Views

In (Abiteboul et al. 1998), incremental refresh of the materialized views over the semistructured data in Object Exchange Model (Papakonstantinou et al. 1995) was investigated. The views considered are the ones defined in an extended version of Lorel (Abiteboul et al. 1996), the query language of Lore (McHugh et al. 1997). The representation of a materialized view is the same to that of (Zhuge and Garcia-Molina 1998) except that the edges among objects are included. The model of update to the data source is the same as that of (Zhuge and Garcia-Molina 1998), and the view refresh is also done similarly. With a update, the queries to be executed against the data source, which are called view maintenance statements, are generated and executed. The retrieved objects are reflected to the materialized view.

3 Management of XML Materialized View

This section describes the storage structures and algorithms proposed to support the XML materialized views and their deferred incremental refresh.

The XML store consists of two areas: the underlying base document area and the materialized view area (see Figure 1). The former is managed by the base document manager, and the latter by the view manager. In the base document area, the DTDs and the XML documents conforming to them are stored. Document indexing is provided for fast access to them. In the view area, on the other hand, the materialized views and the information on the views such as their definition are stored. Indexing is also provided for fast retrieval of the materialized views.

The updates to the base XML documents considered in this paper are the document insertion/deletion and the element modification. When these updates occur, the information on the update is logged in the *update log*. This is for deferred incremental refresh of materialized views. We assume that view refresh is done when the view is requested by a user. That is, the updates are neither immediately nor periodically propagated to the relevant views. Such a materialized view access model is the one employed in (Roussopoulos 1991, Roussopoulos and Kang 1986). In all, the scenario for retrieval of an XML materialized view is as follows: When view V is requested by a user, the view manager requests the

DID	DTDID	EID	Ename	Content
1	1	01	paper	-
1	1	0101	title	Evaluation of a Storage Manager for Retrieval and Update of XML Data
1	1	0102	author	A. Smith and M. Jones
1	1	0103	abstract	XML has emerged as a standard for Web documents …..
1	1	0104	keyword	XML, Web, storage manager, performance evaluation, ...
1	1	0105	section	1. Introduction
1	1	010501	paragraph	The Web was the most influential in advance of the internet .....
1	1	010502	paragraph	The reason why Web has become the core of …..
1	1	010503	paragraph	In most applications today, the Web-based user interface is .....
1	1	0106	section	2. Types of XML Documents
1	1	010601	paragraph	The XML documents are either with the DTD or without .....
1	1	010602	paragraph	The valid XML documents are the ones that are .....
2	1	01	paper	-
2	1	0101	title	A Snapshot Differential Refresh Algorithm
2	1	0102	author	B. Lindsay et al.
2	1	0103	abstract	This article presents an algorithm to refresh the contents of database…
2	1	0104	keyword	Database snapshot, differential refresh, …
2	1	0105	section	1. Introduction
2	1	010501	paragraph	A DBMS provides a mechanism for maintaining, access, and updating…
2	1	010502	paragraph	The notion of a database snapshot was introduced in [ADIBA80]… .
2	1	0106	section	2. Snapshot Refresh Objectives
2	1	010601	paragraph	Snapshot refresh should make the snapshot reflect the current, …
2	1	0107	section	3. Alternative Refresh Methods
2	1	010701	paragraph	Several alternatives are available for implementing snapshot refresh… .
2	1	010702	paragraph	Another alternative is to buffer the changes to the base table and …
3	1	01	paper	-
3	1	0101	title	Document Link and View Update in XML Repository
3	1	0102	author	U. Fox and S. King
3	1	0103	abstract	Due to the proliferation of XML documents on the Web …
3	1	0104	keyword	XML, Web database, extended link, …..
3	1	0105	section	1. Introduction
3	1	010501	paragraph	The difference between the conventional HTML links …..
3	1	010502	paragraph	The virtual document can be implemented on the Web …..
3	1	010503	paragraph	There are so many heterogeneous types of information on the Web …..
3	1	0106	section	2. Related Work
3	1	010601	paragraph	An XML document can represent the structure of …..
3	1	010602	paragraph	The links of XML can use Xlink [6] and Xpointer [7] …..

Figure 2. XMLElem Table

document manager to send it the information necessary for V's refresh. Then, the document manager examines the update log to figure out which updates done to the base documents thus far are relevant to V, generates the *view refresh information*, and sends it to the view manager. Now the view manager refreshes V as directed by the received view refresh information, and then provides up-to-date V to the user.

3.1 Storage Structures for XML Documents and Materialized Views

We assume that the XML documents are the valid ones conforming to their corresponding DTDs. They are decomposed into elements and stored in *XMLElem table* each of whose record corresponds to an element of a document (see Figure 2). A record of XMLElem table consists of DID, DTDID, EID, Ename, and Content columns among others.[1] DID stores the identifier of the XML document, DTDID stores the identifier of the DTD to which the document conforms, EID stores the element identifier, Ename stores the element name, and Content stores the value of the element. EID assignment assumed in this paper is the following: For an element with EID *x*, the EIDs of its children elements are '*xd*' where *d* denotes the system-defined *n* digit representation of integers starting from 1 assigned to each of the children in their order in the document. For example, when *n* = 2, which is used throughout the examples of the paper, the EID of the root element is '01', and for a parent element with EID '0101', the EIDs of its first and second child elements are '010101' and '010102', respectively.

Let us consider a DTD on papers as shown in Figure 3. It consists of 'title', 'author', 'abstract', 'keyword', and 'section' elements, and 'section' element consists of 'paragraph' element. Each of 'title', 'author', 'abstract', and 'keyword' element appear once in an XML document whereas 'section' and 'paragraph' elements appear zero or more times. Figure 2 shows an example of the XMLElem table storing three XML documents on papers.

Meanwhile, *ViewInfo table* and *ViewElem table* are

```
<!ELEMENT  paper (title, author, abstract, keyword, section*)>
<!ELEMENT  title (#PCDATA)>
<!ELEMENT  author (#PCDATA)>
<!ELEMENT  abstract (#PCDATA)>
<!ELEMENT  keyword (#PCDATA)>
<!ELEMENT  section (#PCDATA, paragraph*)>
<!ELEMENT  paragraph (#PCDATA)>
```

Figure 3. DTD on Papers

[1] This table schema is based on the XML-relational mapping by the edge-inlining approach investigated in (Florescu and Kossmann 1999a). In this paper, however, we do not deal with every issue involved in storing XML documents in object-relational database tables. That is beyond the scope of this paper. Rather, we focus on XML materialized view management using the ORDBMS.

ViewID	ViewDef			DTDID
	P	CE	TE	
V_1	contains("Refresh")	title	{title, author, abstract}	1

Figure 4. ViewInfo Table

employed to store the information on the views and their materialization. Each record of ViewInfo table corresponds to a materialized view, and stores the view identifier (ViewID), the view definition (ViewDef), and the identifier of the DTD to which the view's source documents conform (DTDID). In this paper, we assume that an XML view is derived from the base XML documents that conform to the same (and therefore, a single) DTD. In order to efficiently represent the view definition, column ViewDef can be of a *structured* type having a *collection* type as one of its members. Both type constructors are provided in the ORDBMS. Complexity of XML view definition allowed directly affects the process of view refresh. Complex views require more work than simpler ones both in checking a update's relevance to them and in generating their refresh information. For incremental refresh of views to be effective, it is desirable to perform both of the above-mentioned tasks with no or least access to the source data (i.e., XMLElem table in our case). The views dealt with in this paper as the first step of our work are the ones which rarely require access to XMLElem table in their refresh, and yet are practical (see the footnote of Section 3.3.2). The filtering condition of the view is specified only on one of those elements that appear just once in the XML document. That element is called the *condition element*. For the XML documents on papers above, for example, the condition element could be any one of 'title', 'author', 'abstract', or 'keyword'. Elements 'section' and 'paragraph' are not eligible because they could appear more than once. For more complicated views, especially the ones with more than one condition elements, the process of checking relevance between a update and a view and of generating the view refresh information described in Section 3.3.2 needs extension (see the concluding remarks in Section 5). As for the *target elements* that are the elements to be retrieved for the view, however, there is no restriction. Any element can be a target element. If an element which has subelements is designated as a target, then all of its descendant elements are the targets as well. For the XML documents on papers above, for example, if 'title', 'abstract', and 'section' are designated as target elements, the 'paragraph', a subelement of 'section' is also a target element. In all, a view definition has three components: (1) filtering condition P, (2) condition element *(CE)*, and (3) a set of target elements *(TE)*. These are stored in ViewDef column of a structured type with P, CE, and TE as its members where TE is of a collection type. Figure 4 shows an example of ViewInfo table with one record for the view named V_1 whose definition is "*retrieve 'title', 'author', 'abstract' elements where 'title' contains the word 'Refresh'*".

The XML materialized view is represented as an XML document. Figure 5(a) shows the template of an XML materialized view document. Each element 'qdocu'

```
<view>
    <qdocu>
        <t1> … </t1>
            :
        <tn> … </tn>
    </qdocu>
        :
    <qdocu>
        <t1> … </t1>
            :
        <tn> … </tn>
    </qdocu>
</view>
```

(a) Template of an XML Materialized View Document

```
<view>
    <qdocu>
        <title>A Snapshot Differential Refresh Algorithm</title>
        <author>B. Lindsay et al.</author>
        <abstract>This article presents an algorithm to refresh the …</abstract>
    </qdocu>
</view>
```

(b) XML Materialized View Document for view V_1

ViewID	DID	BaseEID	Content
V_1	-	-	-
V_1	2	-	-
V_1	2	0101	A Snapshot Differential Refresh Algorithm
V_1	2	0102	B. Lindsay et al.
V_1	2	0103	This article presents an algorithm to refresh the …

(c) ViewElem Table

Figure 5. XML Materialized View Document and ViewElem Table

which stands for 'the base document qualified for the view' is for a base document that satisfies the view's filtering condition, and its subelements 't_i', i = 1, …, n, are the target elements of the view retrieved from that particular base document. Figure 5(b) shows the XML materialized view document for view V_1. An XML materialized view document is decomposed into elements each of which is stored as a record of ViewElem table. Its record consists of ViewID, DID, BaseEID, and Content columns among others. ViewID, the identifier of the view, is a foreign key referencing ViewID column of ViewInfo table. DID stores the identifier of the base document qualified for the view, from which the current element was retrieved. BaseEID stores the identifier of the current element in its base document identified by DID. Finally, Content stores the value of the element. Figure 5(c) shows ViewElem table that stores the elements of view V_1.

3.2 Logging of Updates to the Base XML Documents

The XML documents could be updated in the unit of document, element, and/or attribute. In this paper, we consider the document insertion/deletion and the element modification. What we mean by element modification is modification of data value of the element which does not have any subelement.

Each update done to the base XML documents is recorded in the update log chronologically for deferred incremental refresh of materialized views. The data

<StartUpdateLog, DTDID, DID, ObjType, OpType>
<[BaseEID, Ename, Content>[,< BaseEID, Ename, Content>,…]]
<EndUpdateLog>

Figure 6. Data Structure of a Update Log Record

structure of the *update log record* is shown in Figure 6. A log record is in a block structure, starting with <StartUpdateLog> field and ending wth <EndUpdateLog> field, to log a related series of updates as an atomic action. In case that an XML document is inserted, for example, it amounts to a sequence of element insertions, and they are logged as an atomic update.

For a update, DTDID represents the identifier of DTD to which the base document involved in the update conforms. DID represents the identifier of the updated document. ObjType denotes whether the unit of update is either element ('ELEMENT') or document ('DOCUMENT'). OpType denotes the type of the update. For element modification, it takes 'MODIFY'. For insertion of a new document, it takes 'INSERT', whereas for deletion of an existing document, it takes 'DELETE'. The triplet <BaseEID, Ename, Content> records the information on the updated element, and could be either skipped or appear once or more times in a log record depending on the value of OpType. BaseEID and Ename are the identifier and the name of the element involved in the update, respectively. Content records the value of the modified or inserted element when OpType='MODIFY' or 'INSERT', respectively. When OpType='DELETE', it is set to NULL.

3.3 Incremental Refresh of an XML Materialized View

The process of incremental refresh of an XML materialized view starts with scanning of the update log for generation of the view refresh information followed by its reflection to the materialized view.

ViewRefresh Table

ViewID	DTDID	FirstLROffset	DIDList
V_1	1	150	{2}
V_2	1	200	{1,3}
V_3	3	1000	{2,4,5,7}
V_4	2	1800	{3,5,6}
…	…	…	

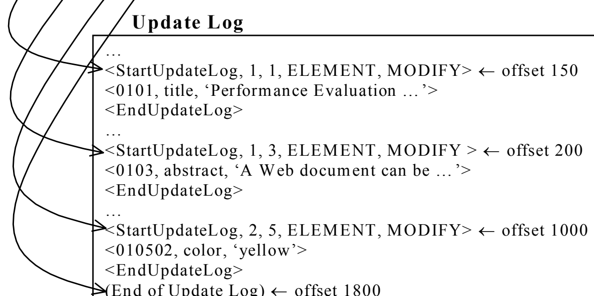


Figure 7. ViewRefresh Table and Update Log

3.3.1 Update Log Scan

Since our update log is a chronological one, it needs to be scanned only for those records logged after the view to be refreshed was refreshed last time. The pointer to the first of those records can be found from *ViewRefresh table*, which stores the information necessary for incremental refresh of views, and resides at the base XML document area of the XML store (see Figure 1). Figure 7 depicts the structure of ViewRefresh table and its relationship with the update log. Each record of ViewRefresh table is for a materialized view, storing its identifier (ViewID), the identifier of DTD to which the view's source documents conform (DTDID), the pointer to the first log record with which the log scan is to start (FirstLROffset), and the list of identifiers of the base documents satisfying the condition of the view (DIDList). Note that the data type of DIDList column needs to be a collection one provided in the ORDBMS.

FirstLROffset is represented as a byte offset from the start of the update log. The log scan starts with the log record whose first byte is stored at FirstLROffset, and ends when the end of log is reached. When a materialized view is created, FirstLROffset column value of its record in ViewRefresh table is initialized to point to the end of the update log. When the log scan for view refresh is completed, it is also set to the end of the log. In Figure 7, for example, when V_1 is to be refreshed, the log records from the offset 150 to the end (i.e., offset 1800) are scanned, and then, FirstLROffset of V_1 in ViewRefresh table is modified to 1800.

The garbage collection for the update log can be simply done by referring to all the FirstLROffset values in ViewRefresh table. First of all, we need to figure out which log record could be eliminated from the log. They are those records stored at the offset less than the minimum of all the FirstLROffset values in ViewRefresh table. Their deletion from the update log entails adjustment of the FirstLROffset values in ViewRefresh table. Each FirstLROffset value in the table is decremented by the above-mentioned minimum offset value. In Figure 6, for example, since FirstLROffset of V_1, which is 150, is the smallest, those records before offset 150 could be deleted. If that is done, FirstLROffset values of V_1, V_2, V_3, and V_4 are adjusted to 0, 50, 850, and 1650, respectively.

3.3.2 Generation of the View Refresh Information

While the update log is scanned, the view refresh information which consists of the operations and their data for view refresh is generated. The view refresh information is represented as a table whose record format consists of RefType, DID, BaseEID, and Content columns. For each update log record, the corresponding update is checked if it is relevant to the view to be refreshed. If it is, the values of the above columns are obtained and gathered to form one or more *view refresh information records* and added to the view refresh information table under construction. Note that all the above columns except RefType are those constituting

ViewElem table record. RefType denotes the *refresh type* whose value is one of 'MODIFY', 'INSERT', and 'DELETE', and it indicates the operation to be performed to the ViewElem table with other column values as its data.

Figure 8 is the C-like pseudo code of algorithm *Gen_RefreshInfo* which generates the view refresh information through update log scan. The input parameters to Gen_RefreshInfo are the identifier (ViewID) and the definition (ViewDef) of the view to be refreshed. Gen_RefreshInfo works as follows: After initializing the refresh information table *RefreshInfo* (init_refreshinfo()) and retrieving the FirstLROffset value from ViewRefresh table, it checks if there is any update in the log to examine. If none exists, it terminates by returning RefreshInfo which is empty. Otherwise, it opens the update log (open_updatelog()) and starts log scanning from the log record pointed to by FirstLROffset to the end of the log. For each log record, it performs the relevance checking (check_relevance()). The function check_relevance() returns the *relevance type* value (rel_type), which is either NULL (indicating that the update is not relevant to the view) or one of MODIFY-M, MODIFY-I, MODIFY-D, INSERT, and DELETE (indicating that it is relevant. These relevance types will be explained in detail shortly.) If the update is relevant, it generates the refresh information records appropriately depending on rel_type and add them to RefreshInfo (gen_add_refinfo()). When the end of log is reached, it modifies the FirstLROffset and the DIDList columns of the view's record in ViewRefresh table, retruns RefreshInfo, and terminates.

In the relevance check, the definition (ViewDef), DTDID, and DIDList of the view are referred to. ViewDef is given as a parameter to Gen_RefreshInfo so that condition P, condition element CE, and the set of target elements TE of the view are referred to. DTDID and DIDList are retrieved from ViewRefresh table. Given update log record U and view V, their relevance checking proceeds as follows: First, equivalence between the DTDID of U and that of V is examined. For U to be relevant to V, basically the two should be the same. If they are, further conditions described below are checked to see if U is relevant to V. There are five types of relevance: MODIFY-M, MODIFY-I, MODIFY-D, INSERT, and DELETE. For each of these types, the further conditions to be checked, how to generate the refresh information record(s), and how to modify the DIDList are described in the following where U.x and V.y denote field x of U and some information y on V, respectively.

```
Gen_RefreshInfo(ViewID, ViewDef)
{
    RefreshInfo = init_refreshinfo();  /* intialization of view refresh information table */
    i = 0;                             /* intialization of RefreshInfo table index */
    LRoffset = ViewRefresh[ViewID].FirstLROffset;   /* retrieval of FirstLROffset from ViewRefresh
                                                       table with ViewID */
    if (LRoffset != NULL && UpdateLog[LRoffset] != end_of_log) /* update exists */
    {
            DTDID = ViewRefresh[ViewID].DTDID;
            DIDList = ViewRefresh[ViewID].DIDList;   /* retrieval of DIDList from ViewRefresh
                                                        table with ViewID */
            DIDListUpdated = False;
            open_updatelog(LRoffset);       /* open update log for scan */
            do {
                    LRoffset = scan_updatelog(LRoffset, &Ulog_Rec); /* retrieval of log record */
                    rel_type = relevance_check(ViewDef, DTDID, DIDList, Ulog_Rec);
                            /* checking relevance between update and view */
                    if (rel_type != NULL) {/* if relevant */
                        i = append_refreshinfo(rel_type, RefreshInfo, DIDList, Ulog_Rec , i);
                        /* generation and addition of view refresh information records */
                        if (rel_type != MODIFY) DIDListUpdated = TRUE;
                    }
            } while (!end_of_log);
            ViewRefresh[ViewID].FirstLROffset = LRoffset;  /* update of FirstLROffset */
            if (DIDListUpdated) ViewRefresh[ViewID].DIDList = DIDList; /* update of DIDList */
    }
    return (RefreshInfo);
}
```

Figure 8. Algorithm for Generation of View Refresh Information

Type MODIFY-M:

If (U.OpType = 'MODIFY' **AND** U.Ename $\in$ V.TE **AND** U.DID $\in$ V.DIDList **AND** (U.Ename $\neq$ V.CE **OR** V.P(U.Content)) where P(x) returns TRUE if x satisfies predicate condition P and returns FALSE otherwise, then it implies that one of V's target elements of the base document (U.Ename $\in$ V.TE) which is qualified for V (U.DID $\in$ V.DIDList) was modified (U.OpType = 'MODIFY'), and that the document is still qualified for V despite the modification (U.Ename $\neq$ V.CE **OR** V.P(U.Content). As such, it is necessary to reflect the same modification to V as well. The view refresh information record added to RefreshInfo is (MODIFY, U.DID, U.BaseEID, U.Content).

Type MODIFY-I:

If (U.OpType = 'MODIFY' **AND** U.Ename = V.CE **AND** U.DID $\notin$ V.DIDList **AND** V.P(U.Content)), it implies that a base document which was not qualified for V (U.DID $\notin$ V.DIDList) is now qualified for V (V.P(U.Content)) due to the modification (U.OpType = 'MODIFY') of an element which is the condition element of V (U.Ename = V.CE). As such, it is necessary to insert the records that are to represent V's target elements of the modified document into ViewElem table. To generate required refresh information records to be added to RefreshInfo, XMLElem table needs to be accessed through the index on DID to retrieve V's target elements of the modified document unless V's condition element is the only target element.[2] Those records are in the form (INSERT, U.DID, BaseEID, Content) where BaseEID and Content values are retrieved from XMLElem Table. Also, U.DID is inserted into V.DIDList.

Type MODIFY-D:

If (U.OpType = 'MODIFY' **AND** U.Ename = V.CE **AND** U.DID $\in$ V.DIDList **AND** (**NOT** V.P(U.Content))), it implies that the modified document which was qualified for V (U.DID $\in$ V.DIDList) is now not so (**NOT** V.P(U.Content)) due to the modification (U.OpType = 'MODIFY') of its condition element (U.Ename = V.CE). As such, it is necessary to delete all the records representing V's target elements of the modified

---

[2] With the restrictions imposed on XML view definition described in Section 3.1, relevance checking can be done with no access to XMLElem table at all, and Modify-I is the only relevance type requiring access to XMLElem table for generating the view refresh information. If we relax those restrictions, accesses to XMLElem table are required not just for generation of the refresh information records but also for relevance checking.

document from ViewElem table. The record to be added to RefreshInfo is (DELETE, U.DID, NULL, NULL) so that all the records of ViewElem table whose DID equals U.DID may be deleted. Also, U.DID is deleted from V.DIDList.

Type INSERT:

If (U.OpType = 'INSERT' **AND** V.P(U.Content where U.Ename = V.CE)) where the clause 'U.Content where U.Ename = V.CE' designates the Content field of the triplet <BaseEID, Ename, Content> in U whose Ename equals V.CE, it implies that a new document was inserted (U.OpType = 'INSERT') which is qualified for V (V.P(U.Content where U.Ename = V.CE)). As such, it is necessary to insert the records representing all of V's target elements of the inserted document to ViewElem table. The necessary values to constitute the records added to RefershInfo are retrieved from the triplets <BaseEID, Ename, Content> in U where Ename equals one of the target elements in V.TE. They are in the form (INSERT, U.DID, U.BaseEID, U.Content). Also, U.DID is inserted into V.DIDList.

Type DELETE:

If (U.OpType = 'DELETE' **AND** U.DID $\in$ V.DIDList), it implies that the document which was qualified for V (U.DID $\in$ V.DIDList) was deleted (U.OpType = 'DELETE'). As such, all the records representing V's target elements of the deleted document need to be deleted from ViewElem table. The record to be added to RefreshInfo is (DELETE, U.DID, NULL, NULL) so that all the records of ViewElem table whose DID equals to U.DID may be deleted. Also, U.DID is deleted from V.DIDList.

3.3.3 Reflection of View Refresh Information to Materialized View

Figure 9 is the C-like pseudo code of algorithm *Refresh_MV* which reflects the view refresh information returned by algorithm Gen_RefreshInfo into ViewElem table. It first checks the returned RefreshInfo table. If it is empty, it means that the materialized view is already up-to-date, and as such, the view refresh is vacuously

```
Refresh_MV (ViewID, RefreshInfo)
{
    i = 0;    /* initialization of RefreshInfo table index */
    if (check_empty(RefreshInfo) != EMPTY)
            /* non-empty refresh information */
            do { RInfo = fetch(RefreshInfo[i++]);
                  switch (RInfo.RefType) {
                    case MODIFY :  /* element modification */
                            modify_content(ViewID, RInfo);
                             /* replacement of element content */
                            break;
                    case INSERT : /* document insertion */
                             i = insert_document(ViewID, Rinfo, RefreshInfo, i);
                             break;
                    case DELETE : /* document deletion */
                            delete_document(ViewID, Rinfo.DID);
                             break;
                  } /* end of switch */
            } while (!end_of_RefreshInfo)
}
```

Figure 9. Algorithm for Reflecting View Refresh Information to Materialized View

completed. Otherwise, it reads the refresh information record in RefreshInfo one at a time into *RInfo* of the type with the same structure as that of the refresh information record depicted in Figure 11, and performs the following:

First, it checks the refresh type value, RInfo.RefType. If it is 'MODIFY', it searches ViewElem table for the record to be modified with ViewID, Rinfo.DID, and RInfo.BaseEID values, and replaces its Content field with RInfo.Content (modify_content()). In doing so, ViewElem table is searched through the index on ViewID.

If RInfo.RefType is 'INSERT', it first inserts into ViewElem table a record where ViewID value is set to the identifier of the view being refreshed and the values of the remaining columns are set to NULL. Then, it inserts into ViewElem table another record with the same ViewID value where the values of the remaining columns are from Rinfo. Such insertions continue for the next records out of RefreshInfo as long as their RefType value is 'INSERT' and their DID value remains the same. In this insertion process, ViewElem table is accessed through the index on ViewID (insert_document()).

If RInfo.RefType is 'DELETE', it searches ViewElem table for the records of the view through the index on ViewID, and deletes all the records whose DID values equals RInfo.DID (delete_document()).

4 Performance Evaluation

Our proposal described in the previous section was implemented in Java with Oracle 8*i*, resulting in a prototype XML storage system running on Windows 2000 Server. In this section, the results of performance experiments with the implemented system are reported.

4.1 Overview

Two types of base XML documents were used in the experiments. One is on movies of small size consisting of about 20 elements per document on the average. The other is the plays of Shakespeare (Bosak 1999) whose average number of elements per document is about 7,000. The views were defined similarly to the one used as the running example in the previous section: They have one condition element and three target elements. Table 1 shows the performance parameters, their description, and setting for the experiments.

The major goal of our experiments is to figure out in what condition incremental refresh of the materialized view outperforms view recomputation. The most influencing performance parameter in this regard is the amount of logged updates to be examined for deferred incremental refresh. As such, in our experiments, we assumed that the number of the base XML documents remained the same with the document insertions and deletions. We also assumed that the size of the retrieved view is the same all the time. These assumptions are to have the view recomputation time to remain virtually the same despite the updates done to the base documents whereas the time with incremental refresh increases as more updates have been done. To achieve this, the number of document insertions and that of deletions are kept the same (i.e., I =

Parameter	Description	Setting
D	The number of base XML documents conforming to a DTD	20000 (movie), 1000 (play)
U	The proportion of base document updates	0, 0.1, 0.2, 0.3, 0.4, 0.5
S	View Selectivity: The proportion of base documents satisfying view's condition	0.2, 0.3
R	Relevance Ratio: The proportion of update log records relevant to the view	0.2, 0.3
I	The proportion of INSERT logging	0.2, 0.4
D	The proportion of DELETE logging	0.2, 0.4
M	The proportion of MODIFY logging	0.6, 0.2
MI	The proportion of MODIFY-I out of all the relevant MODIFY log records	0.2, 0.4
MD	The proportion of MODIFY-D out of all the relevant MODIFY log records	0.2, 0.4
MM	The proportion of MODIFY-M out of all the relevant MODIFY log records	0.6, 0.2

Table 1. Performance Parameters

D), and among them, the number of insertions relevant to the view and that of deletions relevant to the view are also kept the same (i.e., I×R = D×R). The number of element modifications of MODIFY-I relevance type and that of MODIFY-D type are also kept the same (i.e., MI = MD). One more assumption in the experiments was that the relevance ratio is the same as the view selectivity (i.e., R = S).

4.2 View Retrieval Time

The time for XML view retrieval with incremental refresh of the materialized view and that with view recomputation were measured for comparison. Figure 10 through Figure 12 compare the view retrieval times by these two methods as the proportion of base document updates (U) increases. We note that as U increases, the time with view recomputation is not changing whereas that with incremental refresh increases. In Figure 10, which is out of the experiments against the movie documents, incremental refresh outperforms view recomputation as long as U is less than about 27%. This upper limit on the amount of updates for incremental refresh to be more effective than view recomputation goes up further in Figure 11, which is also out of the experiments against the movie documents, to about 31% (S = 0.2) and about 33% (S = 0.3). It goes up to as much as about 42% in Figure 12, which is out of the experiments against the play documents.

The results in Figure 12 compared to those in Figure 10 reveal that incremental refresh gets more effective than

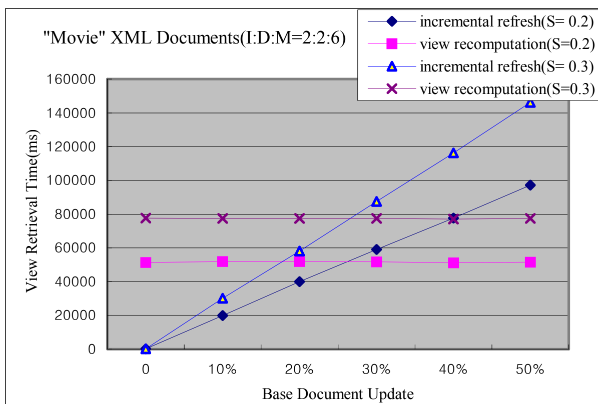
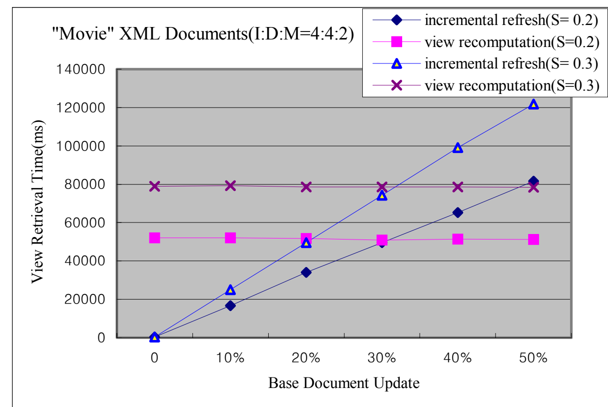


Figure 10. View Retrieval Time w.r.t Varying Proportion of Base Document Updates



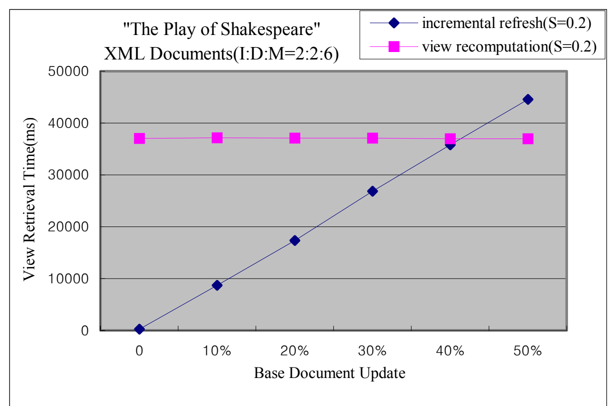


Figure 12. View Retrieval Time w.r.t Varying Proportion of Base Document Updates

view recomputation as the volume of the view's source documents gets larger. This implies that XML view materialization would be very effective in providing the database like services out of a large-scale XML warehouse like the one investigated in Xyleme project (Xyleme 2001).

Effectiveness of incremental refresh compared to view recomputation is also observed as the size of the view gets larger or as the occurrences of complex type of relevant updates decreases. As the view selectivity (S) increases from 0.2 to 0.3 in Figure 10 and Figure 11, both view recomputation time and the time with incremental refresh increase because the size of the view gets larger.

However, we can observe that view refresh is less sensitive to that especially when U is less than around 30%.

In the experiments of Figure 10, the ratio among the relevant updates I:D:M was set to 2:2:6 whereas in Figure 11, it changed to 4:4:2. The reduction of the relevant updates of MODIFY type results in reduction of MODIFY-I type updates. As explained in Section 3.3.2, MODIFY-I is the only relevance type which requires the access to the view's source documents (i.e., XMLElem table) to generate the corresponding refresh information which is not found in the update log. As such, it takes the longest time to process among all the five relevance types. We can note that the view retrieval times with incremental refresh in Figure 11 have decreased compared to those in Figure 10.

4.3 Time for Generating Refresh Information

Figure 13 shows the time it took to generate the refresh information for each relevant update type. As explained in Section 3.3.2 and in the previous subsection, MODIFY-I takes the longest time among all the five relevance type updates.

The second longest one is INSERT, and the remaining three follows without notable difference. The INSERT type update requires more time than the other three because it needs to search its log record for the condition element and the target ones whereas the other three can simply generate the refresh information.

4.4 Time for Tasks of Incremental Refresh

There are three tasks involved in the view retrieval with incremental refresh of the materialized view: (1) scanning the update log, (2) checking relevance between the updates and the view and generating the view refresh information for the relevant updates, and (3) reflecting it into the materialized view. Figure 14 compares the time it took for each task while refreshing a materialized view from the movie documents. We note that the most time consuming part is the second one, the core of the XML view materialization with the highest complexity of the three. Next is the log scanning which requires disk I/O's.

5 Concluding Remarks

In this paper, we investigated the XML view: its materialization and incremental refresh. Instead of relying on the semistructured DBMS for XML data storage, we employed the ORDBMS because of its pragmatic importance.

The object-relational database schema for storing the base XML documents, the materialized views derived from them, and other information for view refresh is designed. We adopted the deferred view refresh policy, requiring the update log. The data structure of the update log record and how the update log is managed and scanned is described. We proposed two algorithms Gen_RefreshInfo and Refresh_MV. The former scans the update log, checks if each logged update is relevant to the view to be refreshed, and generates the view refresh information that consists of the operations and their data necessary for view refresh. The latter algorithm incrementally refreshes

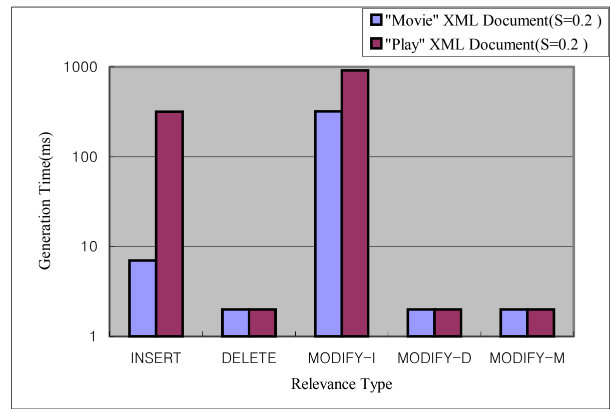


Figure 13. Time for Generating Refresh Information for each Relevance Type

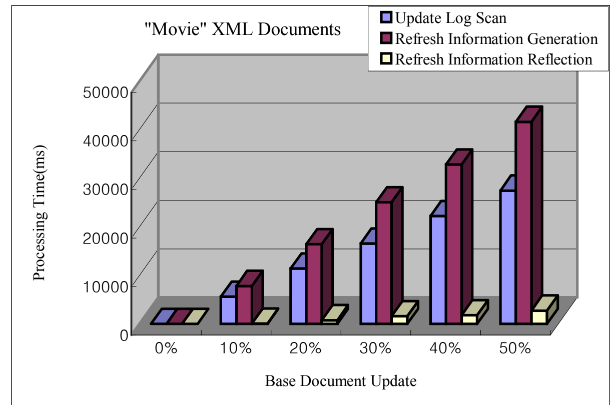


Figure 14. Time for each Task in Incremetal Refresh of Materialized View

the materialized view with the view refresh information. Finally, a detailed set of experimental results on performance were presented, showing that our proposed scheme outperforms view recomputation.

The issues requiring further investigation include the following: First, we adopted the deferred view refresh policy in this paper. That would give us an opportunity for post optimization in generating the view refresh information. Once the view refresh information is generated by algorithm Gen_RefreshInfo as described in section 3.3.2, it could be optimized by merging the related refresh information records. For example, if an element of a newly inserted document is modified later, then their refresh information records can be merged into one so that the modified value may be inserted instead of the original one. As for another example, if the document with one of its element modified is later deleted, then the earlier element modification need not be reflected to the materialization of the view.

Secondly, a scheme for efficient logging of document insertion or element modification needs to be devised. This is to avoid the log records of very large size when inserting a large document or modifying a large element. The logged data values of elements are redundantly stored in the corresponding base documents. As such, some referencing mechanism where a log record points to its relevant portion of the base document, is desirable. The penalty for that is on the process of checking the update's relevance to a view, which inevitably requires access to the base documents.

Thirdly, the views with unrestricted filtering condition need to be dealt with. In this paper, given a log record, its update's relevance to a view can be determined without access to the base documents at all. Besides, Modify-I is the only relevance type requiring access to the base documents for generating view refresh information. However, as mentioned in Section 3.3.2, with the view whose condition is more general than that described in Section 3.1, base document access is inevitable both for checking update/view relevance and for generating view refresh information.

Fourthly, the finer unit of updates to the XML documents needs to be considered. In this paper, the granularity for a update we considered is either the entire document for insertion and deletion or the element for modification. We need to extend the result of this paper to incorporate more complicated model of updates like the element insertion and deletion, which results in structural change of the XML documents.

Finally, performance analysis needs to be conducted to derive the equation on the metadata in the XML store whose value is a priori known or can be estimated so that the XML query optimizer can always choose the winner of incremental refresh and recomputation given a request of the XML view that is maintained as a materialized one.

6 References

Abiteboul, S. (1999): On views and XML. *Proc. ACM Symp. on Principles of Database System*, 1-9.

Abiteboul, S. et al. (1996): The Lorel query language for semistructured data. *J. of Digital Libraries* **1**(1).

Abiteboul, S. et al. (1998): Incremental maintenance for materialized views over semistructured data. *Proc. Int'l Conf. on VLDB*, 38-49.

Abiteboul, S. et al. (1999): Active views for electronic commerce. *Proc. Int'l Conf. on VLDB*, 138-149.

Boag, S. et al. (2002): *XQuery 1.0: an XML query language*. http://www.w3.org/TR/xquery/.

Bosak, J. (1999): *The plays of Shakespeare*. http://www.ibiblio.org/bosak/.

Buneman, P. et al. (1995): Programming constructs for unstructured data. *Proc. DBPL*.

Carey, M. et al. (2000): XPERANTO: publishing object-relational data as XML. *Proc. Workshop on the Web and Databases*.

Cattell, R. et al. (1994): *The object database standard: ODMG-93*. Morgan Kaufmann.

Chen, L. and Rundensteiner, E. (2000): Aggregate path index for incremental Web view maintenance. *Proc. 2nd Int'l Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*.

Chen, L. and Rundensteiner, E. (2002): ACE-XQ: a cache-aware XQuery answering system. *Proc. Workshop on the Web and Databases*.

Chen, L. et al. (2002): XCache - a semantic caching system for XML queries. Proc. *ACM SIGMOD Int'l Conf. on Management of Data*, Demonstration paper.

Cluet, S. et al. (2001): Views in a large scale XML repository. *Proc. Int'l Conf. on VLDB*, 271-280.

Deutsch, A. et al. (1999): Storing semistructured data with STORED. *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, 431-442.

Fernandez, M. et al. (2000): SilkRoute: trading between relations and XML. *Proc. the 9th WWW Conf.*, 723-746.

Fernandez, M. et al. (2001): Efficient evaluation of XML middle-ware queries. *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, 103-114.

Florescu, D. and Kossmann, D. (1999a): Storing and querying XML data using an RDBMS. *IEEE Data Eng. Bulletin* **22**(3):27-34.

Florescu, D. and Kossmann, D. (1999b): A performance evaluation of alternative mapping schemes for storing XML data in a relational database. *Tech. Rep., INRIA, France*.

Gupta, A. and Mumick, I. (ed.) (1999): *Materialized views: techniques, implementations, and applications*. MIT Press.

Hristidis, V. and Petropoulos, M. (2002): Semantic caching of XML databases. *Proc. Workshop on the Web and Databases*.

McHugh, J. et al. (1997): Lore: a database management system for semistructured data. *ACM SIGMOD Record* **26**(3):54-66.

Papakonstantinou, Y. et al. (1995): Object exchange across heterogeneous information sources. *Proc. Int'l Conf. on Data Engineering*, 251-260.

Quan, L. et al. (2000): Argos: efficient refresh in an XQL-based Web caching system. *Proc. Workshop on the Web and Databases*, 23-28.

Robie, J. et al. (1998): *XML query language (XQL)*. http://www.w3.org/TandS/QL/QL98/pp/xql.html.

Roussopoulos, N. (1991): An incremental access method for ViewCache: concept, algorithms, and cost analysis. *ACM Trans. on Database Systems* **16**(3):535-563.

Roussopoulos, N. and Kang, H. (1986): Principles and techniques in the design of ADMS±. *IEEE Computer* **19**(12):19-25.

Shanmugasundaram, J. et al. (1999): Relational databases for querying XML documents: limitations and opportunities. *Proc. Int'l Conf. on VLDB*, 302-314.

Shanmugasundaram, J. et al. (2000): Efficiently publishing relational data as XML documents. *Proc. Int'l Conf. on VLDB*, 65-76.

Shanmugasundaram, J. et al. (2001): Querying XML views of relational data. *Proc. Int'l Conf. on VLDB*, 261-270.

Suciu, D. (1996): Query decomposition and view maintenance for query languages for unstructured data. *Proc. Int'l Conf. on VLDB*, 227-238.

Tatarinov, I. et al. (2001): Updating XML. *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, 413-424.

Tian, F. et al. (2002): The design and performance evaluation of alternative XML storage strategies. *ACM SIGMOD Record* **31**(1):5-10.

Xyleme, L. (2001): A dynamic warehouse for XML data of the Web. *IEEE Data Eng. Bulletin* **24**(2):40-47.

Zhuge, Y. and Garcia-Molina, H. (1998): Graph structured views and their incremental maintenance. *Proc. Int'l Conf. on Data Engineering*, 116-125.