

XML View Materialization with Deferred Incremental Refresh: the Case of a Restricted Class of Views*

DAE HYUN HWANG AND HYUNCHUL KANG

School of Computer Science and Engineering

Chung-Ang University

Seoul, 156-756, Korea

E-mail: dhhwang@dblab.cse.cau.ac.kr

E-mail: hckang@cau.ac.kr

A view mechanism can provide a user with an appropriate portion of a database through data filtering and aggregation. Views are often materialized for query performance improvement, and in that case, their consistency needs to be maintained against 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 on efficient storing and querying of XML documents with the DBMS. In this paper, we investigate the materialization of XML views and their incremental refresh for the case of a restricted class of views. The object-relational DBMS is employed to store XML documents and their materialized views, and the update log is used for deferred view refresh. Algorithms for checking a logged update's relevance to a view and for generating the optimized SQL statements to refresh the materialized view stored in an object-relational database through scanning of the update log are described. Experimental results show that our approach can be very effective in providing views of a large-scale XML warehouse on the Web.

Keywords: XML, materialized view, deferred incremental view refresh, Web, semi-structured data

1. INTRODUCTION

In database systems, a view is used to represent a portion of a database defined by a query expression. The view concept has been applied in useful and effective mechanisms for 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 concept is information filtering and aggregation [28], functionality which is becoming even more crucial for information processing in today's Web-based computing environment, where vast amount of heterogeneous information are added every day.

Views are often materialized for query performance, thus requiring that their consistency be maintained against updates of the underlying data [18]. Consistency maintenance can be done either by recomputing the view from the source data or by incremen-

Received February 13, 2003; revised July 30 & October 13, 2004; accepted January 12, 2005.

Communicated by Ming-Syan Chen.

* This Work was supported by the Basic Research Program of the Korea Science and Engineering Foundation, grant No. R01-2003-000-10395-0.

tally refreshing the outdated materialized view. The latter can be done either immediately after the source update occurs or in a deferred manner.

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 dealing with XML data [1, 4, 14, 25]. However, the problem of *XML view materialization* has rarely been addressed. In this paper, we investigate this issue with respect to fast retrieval of XML views.

View materialization in relational databases is well understood [18]. Would the same technology be useful and efficient in the XML context? Currently, the database research community is making great efforts to establish XML query processing technology. So far, these efforts have yielded several state-of-the-art techniques, which include Lore's navigation [22], XRel's path table approach [37], structural joins with XML numbering schemes [5, 8, 12, 20, 21, 38], and mixed mode processing [19]. These techniques are all traditional in that, they generate query results from scratch.

Materialization and incremental maintenance of XML views, when generated by any of the aforementioned XML query processing algorithms, will be core techniques in *XML warehouses*, which will soon be commonplace on the Web. In fact, there already is a commercial XML warehouse on the Web [36], which was spun off the Xyleme project of INRIA, France [35]. It is certain that more and more data exchanged on the Web will be in XML and will possibly conform to some DTDs or XML Schemas used to describe it. Proliferation of XML documents on the Web will necessitate XML warehousing, where view materialization is often required for good query performance. The space overhead for materializing XML views could be fully justified because the gross performance gain through view materialization could be enormous on the Web, where the frequency of popular view access for Web applications might be on the Internet scale.

There are three major issues in XML view materialization: (1) checking the relevance between an update and a view and generating view refresh information, which consists of data and operations for relevant updates; (2) the storage of XML materialized views in connection with that of XML sources; and (3) scalability. The concepts involved in the first issue are well understood with regard to relational databases. For XML, however, the solution must be much more complicated than that for the relational case because of the hierarchical and nested structure of XML data. The second issue is practically important and closely influences the design of solutions for the first problem. The two major types of operations performed on a materialized view are retrieving the whole view to provide the requested view and fixing just a small portion of it for incremental refresh. There are performance tradeoffs in efficiently executing these two types of operations. The scalability requirement is mandatory because XML views are mostly retrieved for Web applications. There are two issues here. First, since XML views are defined against an XML source on the Web, the volume of a view's source could be huge. Secondly, the number of views materialized could also be huge, perhaps on the Internet scale. These three issues are interrelated to each other.

To the best of our knowledge, [27] is the only work on XML view materialization in the literature. Among the three issues described above, only the first issue was dealt with in [27]. In this paper, we investigate XML view materialization and address all three issues for the case of a restricted class of views. The restrictions imposed on a view are listed in section 2.1, and their removal as a future work is described in section 5.

There are three aspects to our work: First of all, we consider the case where the source XML documents as well as their materialized views are stored in *a relational DBMS (RDBMS) or an object-relational DBMS (ORDBMS)*.¹ Since the traditional RDBMSs and modern ORDBMSs are widely used, storing and querying XML documents with them is of practical importance and has attracted much attention [15, 16, 29, 31, 32, 37]. With a DBMS employed as XML store, storing and incremental refresh of XML materialized views can also leverage the well-established DBMS technology, and a wide range of techniques proposed for XML data management with a DBMS, such as XML-relational mapping, can be utilized for XML view materialization, making our proposed scheme seamlessly integratable with state-of-the-art XML data management technology.

Secondly, we consider XML views against *a huge set of XML documents* in an XML warehouse on the Web, rather than against a single document. This means that an XML view is the result of *document filtering* as well as of *element retrieval* from each filtered document. This type of view is practically useful because a large number of XML documents can usually conform to some schema (e.g., DTD) in an XML warehouse. Since XML emerged as the de facto standard for data exchange on the Web, many communities have been defining schemas for their data in DTDs, XML Schemas, or other specifications to provide standards for representing and exchanging their data. For example, in the business community, the XML schemas defined by various industry groups and organizations are registered in the XML.org Registry of OASIS and serve as community resources for the fast-growing body of XML specifications developed for vertical industries and horizontal applications [34].

Thirdly, we employ *deferred incremental refresh* of XML materialized views, which requires the *logging of updates* of source XML documents. The deferred policy is preferable to its immediate counterpart for supporting a number of XML materialized views on the Web. Immediate update propagation will not scale as the number of views maintained as materialized ones increases, since this might be on the Internet scale. With deferred update propagation, the overhead incurred in XML update processing to support materialized views is just update logging, which is negligible because the time needed for update logging is dependent on the complexity of the update operation itself, not on the number of materialized views supported. Another important advantage of the deferred refresh policy is that it enables the *optimized* incremental refresh of a materialized view. When the interrelationships among logged updates are identified, some can cancel each other out, merged to form one, and/or just be ignored. As for the space overhead incurred in update logging, it is regarded as small because the logged data is shared for all materialized views, the number of which can be enormous. Besides, it will be fully amortized by the performance gain through view materialization.

The rest of this paper is organized as follows: Section 2 describes our scheme for XML view materialization with deferred incremental refresh. Section 3 reports experimental results for performance. Section 4 surveys the related works and compares them with ours. Finally, section 5 gives concluding remarks and describes future works.

¹ The ORDBMS is preferred to the RDBMS in our work because it is desirable to employ the table columns of *structured* and *collection* types, and the *MERGE* statement, which are specified in SQL99 and provided by off-the-shelf commercial ORDBMSs.

2. XML VIEW MATERIALIZATION WITH DEFERRED INCREMENTAL REFRESH

In this section, we describe our scheme for XML view materialization with deferred incremental refresh. The storage structures, techniques, and algorithms involved in our scheme include: (1) the object-relational database schema used to store not just the base XML documents, but also the materialized views derived from them, and some *auxiliary* information necessary for supporting view materialization; (2) the logging of updates of XML sources; and (3) the algorithm for the deferred incremental refresh of an XML materialized view, the core of which consists of (i) relevance checking between a logged update and a view, and (ii) the generation of optimized SQL statements used to refresh materialized view stored in an object-relational database through scanning of the update log.

2.1 Overview

The storage area of an XML warehouse built on top of an ORDBMS consists of two parts: the *base document area* and the *materialized view area* (see Fig. 1). The former is managed by the base document manager, and the latter by the view manager. This separation makes it easy to migrate to the environment of the Web, where XML materialized views reside at sites other than the one where their source is stored.

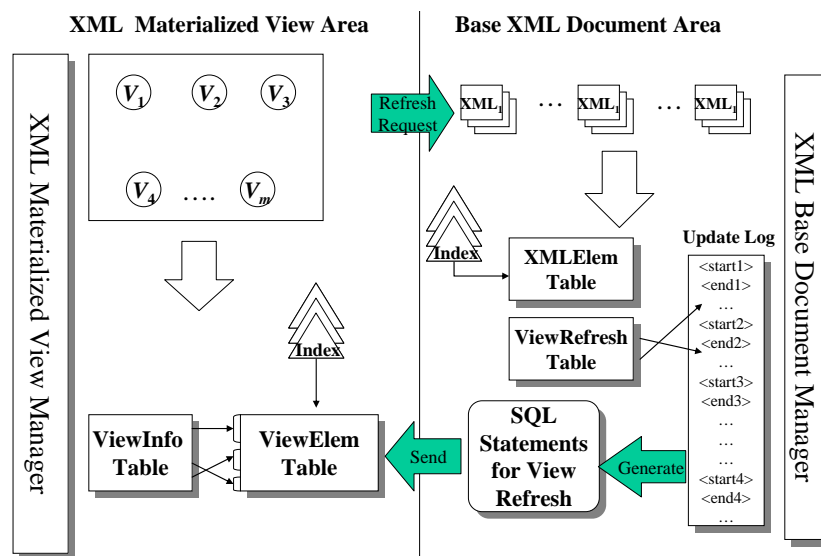


Fig. 1. Management of XML materialized views.

In the base document area, the base XML documents inserted into the XML warehouse are stored with proper indexing for fast access. In the view area, the materialized

views and some auxiliary information, such as their definitions are stored. Indexing is also provided for fast retrieval of materialized views.

An XML view of a collection of XML documents in the base document area is defined by an XML query language expression. Although the W3C recommendations of XML query languages, such as XQuery and XPath, do not specify how the results of query expressions are to be returned, we assume that an XML view is also an *XML document* because it would mostly be exported for Web applications. Since an XML view is derived from a collection of base XML documents, it is the result of document filtering as well as of element retrieval of filtered documents. In the definition of a view, thus, there needs to be some predicate condition given for filtering source documents. The XML elements on which this filtering condition is specified are called the *condition elements*. A list of elements to be retrieved for the view for each filtered document also needs to be given. They are called *target elements*.

As in previous works on storing XML in the DBMS [15, 16, 29, 31, 32, 37], when base documents are stored in the ORDBMS, they are parsed and decomposed into XML elements and subelements, which are mapped to tuples. As such, XML view materialization amounts to *caching* the tuples of the base documents that are needed to construct the view. Hence, when an XML materialized view is returned, retrieving its tuples followed by *tagging* them in XML is needed. When XML materialized views are stored this way, both returning the whole view and fixing a small portion of it for incremental refresh can be efficiently supported. In addition, the *mapping* between the data objects in the materialized views and their counterparts in the source can be easily done because both the source and materialized views are stored in the same structures. An alternative approach is to store XML materialized views as plain text files. The problem with this approach is that incremental refresh of views might be quite inefficient because of the compact storage format, though returning them would be optimal. A separate auxiliary information structure would also be needed to maintain the aforementioned mapping.

When some update is performed on the base documents in the base document area, information about the update is logged in the update log. The time when the deferred refresh is done is assumed in this paper to be when the view is requested. Such last minute (laziest) refresh obviously slows down delivery of the view compared to the case where an up-to-date view is made ready through several alternative refresh policies, such as periodic, on-demand, automatically triggered depending on the size of the update log, off-line, and so on. But the major problem with these more alternative approaches is that they can put a formidable burden on the system when many views are maintained as materialized ones, thus slowing down all the activities of the system, including the delivery of already up-to-date views. (Since we deal here with XML views on the Web, the number of views maintained as materialized ones can be on the Internet scale.) To come up with the best policy for each materialized view, the tradeoffs between the various refresh policies, from the immediate one to the last minute deferred one, need to be carefully examined, while considering access frequency and source update frequency as well. In this paper, we consider the aforementioned last minute refresh policy for the following reasons: First of all, it is scalable as the number of materialized views increases. Other alternatives will not scale enough. Secondly, even if any of or any combination of the other alternatives is fully or partially adopted, a view cannot be guaranteed to be up-to-date when it is requested. In this case, our last minute refresh is needed anyway. (For

details about the consistency of refreshed materialized views, refer to section 2.4.5.) Thirdly, it is simple to implement.

The scenario for the retrieval of XML view V , which is maintained as a materialized one, is as follows (see Fig. 1): When V is requested, the view manager asks the base document manager to send it the optimized SQL statements needed to refresh. Then, the document manager (1) freezes (i.e., blocks updates against) V 's source, (2) identifies the scope of the update log to be processed, (3) examines all the update log records in that scope to figure out which updates performed on the base documents are relevant to V , (4) generates the optimized SQL statements from among the relevant update log records, and (5) sends them to the view manager. Then, the view manager refreshes the tuples of V , retrieves them, tags them in XML, and then returns the up-to-date V .

The restrictions imposed on the XML view that we will deal with through section 4 of this paper are as follows: In defining an XML view, there is only one condition element involved, for which there can be at most one instance per base document, and the target elements are all leaves where the tag names of conditions or target elements are unique within a base document. As for updates to the source XML documents that can affect the views, there are only three types: document insertion, document deletion, and leaf element modification, where the text (PCDATA) is modified. These restrictions are mostly imposed to ease the exposition and removal of them is described in section 5.

2.2 Storage Structures for XML Documents and Materialized Views

Many XML-relational mappings have been proposed in the literature [15, 16, 29, 31, 32, 37]. Among them, we adopt the edge-inlining approach investigated in [16, 17] for storing base XML documents because with it, we can store XML documents without the need for DTDs. A base XML document is decomposed into elements and stored in an *XMLElem table*, whose record corresponds to an element of the document. An *XMLElem table* consists of DID, EID, ParentEID, Ename, and Content (see Fig. 2). DID stores the identifier of each XML document, EID stores the identifier of each element, ParentEID stores the identifier of each element's parent element, Ename stores the tag name of each element, and Content stores the text of each leaf element. An XML document is usually modeled as a node-labeled ordered tree, and each XML element, be it a leaf or non-leaf, corresponds to a node of that tree. EID assignment is done in a monotonically increasing way from the root element to its subelements through *preorder* traversal of the XML tree. EID assignment is used in most of the popular XML numbering schemes for XML query processing with structural joins [5, 8, 12, 20, 21, 38]. Fig. 2 shows an example of an *XMLElem table* storing three XML documents on papers whose DTD is shown in Fig. 3.

Meanwhile, the *ViewInfo table* and *ViewElem table* are employed to store the definition of the views and their materialization, respectively. Each record of the *ViewInfo table* corresponds to a materialized view and stores the view identifier (ViewID), a symbolic name of the view's source (ViewSrc), and the view definition (ViewDef), which has three components: (1) filtering condition P , (2) condition element (CE), and (3) a set of target elements (TE). In order to efficiently represent the view definition, the *ViewDef* column can be of a *structured* type, having a *collection* type as one of its members. Both type constructors are provided in the ORDBMS. These are stored in a *ViewDef*

DID	EID	ParentEID	Ename	Content
1	1	0	paper	-
1	2	1	Title	Evaluation of a Storage Manager for Retrieval and Update of XML Data
1	3	1	author	A. Smith and M. Jones
1	4	1	abstract	XML has emerged as a standard for Web documents ...
1	5	1	keyword	XML, Web, storage manager, performance evaluation, ...
1	6	1	section	1. Introduction
1	7	6	paragraph	The web was the most influential in advance of the internet ...
1	8	6	paragraph	The reason why Web has become the core of ...
1	9	6	paragraph	In most applications today, the Web-based user interface is ...
1	10	1	section	2. Types of SML Documents
1	11	10	paragraph	The XML documents are either with the DTD or without ...
1	12	10	paragraph	The valid XML documents are the ones that are ...
2	1	0	paper	-
2	2	1	title	A Snapshot Differential Refresh Algorithm
2	3	1	author	B. Lindsay <i>et al.</i>
2	4	1	abstract	This article presents an algorithm to refresh the contents of database ...
2	5	1	keyword	Database snapshot, differential refresh, ...
2	6	1	section	1. Introduction
2	7	6	paragraph	A DBMS provides a mechanism for maintaining, access, and updating ...
2	8	6	paragraph	The notion of a database snapshot was introduced in [ADIBA80] ...
2	9	1	section	2. Snapshot Refresh Objectives
2	10	9	paragraph	Snapshot refresh should make the snapshot reflect the current, ...
2	11	1	section	3. Alternative Refresh Methods
2	12	11	paragraph	Several alternatives are available for implementing snapshot refresh ...
2	13	11	paragraph	Another alternative is to buffer the changes to the base table and ...
3	1	0	paper	-
3	2	1	title	Document Links and View Update in XML Repository
3	3	1	author	U. Fox and S. King
3	4	1	abstract	Due to the proliferation of SML documents on the Web ...
3	5	1	keyword	XML, Web database, extended link, ...
3	6	1	section	1. Introduction
3	7	6	paragraph	The difference between the conventional HTML links ...
3	8	6	paragraph	The virtual document can be implemented on the Web ...
3	9	6	paragraph	There are so many heterogeneous types of information on the Web ...
3	10	1	section	2. Related Work
3	11	10	paragraph	An XML document can represent the structure of ...
3	12	10	paragraph	The links of XML can use Xlink [6] and Xpointer [7] ...

Fig. 2. XMLElem table.

```

<!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)>

```

Fig. 3. DTD on papers.

column of a structured type with P, CE, and TE as its members, where TE is of a collection type. Fig. 4 shows an example of a ViewInfo table with one record for the view whose identifier is V_1 . It is used to export the “title,” “author,” and “abstract” elements of the paper documents whose “title” contains the word “Refresh”.

As described in section 2.1, an XML view is represented as an XML document. Fig. 5 (a) shows the template of an XML materialized view document adopted in this paper. Each element “qdocu,” 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 ,” $i = 1, \dots, n$, are the target elements of the view retrieved from that particular base document. Element “qdocu” is used to delimit the target elements retrieved from different base documents and can be eliminated when the view is returned if there is no need of distinguish among the target elements in terms of their source documents. Fig. 5 (b) shows the XML materialized view document for view V_1 .

ViewID	ViewSrc	ViewDef		
		P	CE	TE
V_1	\$(Paper)	contains(“Refresh”)	title	{title, author, abstract}

Fig. 4. ViewInfo table.

```

<view>
  <qdocu>
    < $t_1$ > ... </ $t_1$ >
    ⋮
    < $t_n$ > ... </ $t_n$ >
  </qdocu>
  ⋮
  <qdocu>
    < $t_1$ > ... </ $t_1$ >
    ⋮
    < $t_n$ > ... </ $t_n$ >
  </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 .

Fig. 5. XML materialized view document and ViewElem table.

ViewID	DID	EID	ParentEID	Ename	Content
V_1	2	2	1	title	A Snapshot Differential Refresh Algorithm
V_1	2	3	1	author	B. Lindsay <i>et al.</i>
V_1	2	4	1	abstract	This article presents an algorithm to refresh the ...

(c) ViewElem table.

Fig. 5. (Cont'd) XML materialized view document and ViewElem table.

When an XML view is materialized, its tuples are retrieved from the XMLElem table and then cached in the ViewElem table. As such, the schema of the ViewElem table is the same as that of the XMLElem table except that the ViewID column, the identifier of the view, is augmented. ViewID is a foreign key referencing the ViewID column of the ViewInfo table. Fig. 5 (c) shows the ViewElem table that stores the elements of view V_1 .

To return a view in XML, whether it is retrieved from the XMLElem table or the ViewElem table, its tuples need to be tagged in XML. Our XML tagging scheme, which transforms the tuples into XML, is very effective in the sense that the tuple stream of the SQL result set need not be buffered at all. As the tuple stream is produced, each tuple is fed into the tagging process, which consumes it and throws it away before the process needs to consume the next one in the stream. When the last tuple of the stream is processed, generation of the view in XML is completed. Such *pipelined XML tagging* is possible because (1) EIDs within a document are assigned in the preorder traversal of the XML tree as explained above, (2) each tuple stores the EID of an XML element along with its ParentEID, and (3) the tuple stream is produced as *sorted* on DID and then on EID within a document using an ORDER BY clause of SQL (i.e., ORDER BY Did, Eid). Such ordering of tuples makes it possible for any XML subtree t of an XML document to be generated in a pipelined way with a *stack* of \langle Ename, EID \rangle pairs as the tuple stream that corresponds to the XML elements and subelements belong to t is produced. A core part of the tagging process is comparing the ParentEID of the current tuple with the EID of the stack top. If the ParentEID of the current tuple is equal to the EID of the stack top, an open tag is written with the Ename of the current tuple and the \langle Ename, EID \rangle pair of the current tuple is pushed to the stack. Otherwise, the stack is popped up, and the close tag is written with the Ename popped.

2.3 Logging of Updates to Base XML Documents

Each update performed on the base XML documents is recorded in the update log chronologically to facilitate deferred incremental refresh of materialized views. The information logged for update u is limited to the information regarding u . Although this logging step facilitates incremental refresh of XML materialized views, no information which is relevant to materialized views is logged. View-conscious logging is definitely more useful for refreshing materialized views, but it is not scalable as the number of materialized views increases. The data structure of the *update log record* is shown in Fig. 6. A log record uses a block structure, starting with the \langle StartUpdateLog \rangle field and ending with the \langle EndUpdateLog \rangle field, to log a related series of updates as an atomic action.

```

<StartUpdateLog, LSN, DID, ObjType, OpType>
<EID, ParentEID, Ename, Content>[,<EID, ParentEID, Ename, Content>, ...]
<EndUpdateLog>

```

Fig. 6. Data structure of an update log record.

When an XML document is inserted, for example, this amounts to a sequence of element insertions, and they are logged as an atomic update. For an update, LSN is the log sequence number assigned for each log record. Given two update log records u and v , $u.LSN < v.LSN$ if and only if the update of u precedes that of v . 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 update. For element modification, it takes "MODIFY." For the insertion of a new document, it takes "INSERT," whereas for the deletion of an existing document, it takes "DELETE." The quartet <EID, ParentEID, Ename, Content> records the information of the updated element and can be either skipped or can appear once or more times in a log record, depending on the values of ObjType and OpType. EID, ParentEID, and Ename are self-explanatory. Content records the value of the modified or inserted element when OpType = "MODIFY" or "INSERT," respectively. When OpType = "DELETE" it is set to NULL.

2.4 Deferred Incremental Refresh of an XML Materialized View

The process of deferred incremental refresh of an XML materialized view consists of (1) checking the relevance between the logged updates and the view while scanning the update log, (2) generating the optimized SQL statements with the information obtained as a result of processing the log, and finally (3) executing them to refresh the ViewElem table.

2.4.1 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 the *ViewRefresh table*, which stores the auxiliary information necessary for incremental refresh of materialized views and resides in the base XML document area of the XML warehouse (see Fig. 1). Fig. 7 depicts the structure of the ViewRefresh table and its relationship with the update log. Each record of the ViewRefresh table is for a materialized view, storing its identifier (ViewID), the pointer to the first log record at 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 the 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 at the log record whose first byte is stored at the location with FirstLROffset and ends when the location that was the end of the log at the time when the log was open for view refresh is reached. When the log scan for view refresh is completed,

ViewRefresh Table

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

Update Log

```

...
<StartUpdateLog, 1, 1, ELEMENT, MODIFY> ← offset 150
<2, 1, title, 'Performance Evaluation ...'>
<EndUpdateLog>
...
<StartUpdateLog, 3, 1, ELEMENT, MODIFY > ← offset 200
<4, 1, abstract, 'A Web document can be ...'>
<EndUpdateLog>
...
<StartUpdateLog, 10, 5, ELEMENT, MODIFY> ← offset 1000
<25, 7, color, 'yellow'>
<EndUpdateLog>
(End of Update Log) ← offset 1800

```

Fig. 7. ViewRefresh table and update log.

the view's FirstLROffset is set to the *current* end of the log. (The reason why it is set to the current end instead of the location where the log scan ended is explained in section 2.4.5.) In Fig. 7, for example, when V_1 is to be refreshed, the log records from the location with offset 150 to the end (i.e., the location with offset 1800 if that location was the end of the log at the time when the log was open for the refresh of V_1) are scanned, and after that, FirstLROffset of V_1 in the ViewRefresh table is modified to point to the current end of the log (e.g., 1800 if no further update was logged during the log scan). When a materialized view is created, the FirstLROffset column value of its record in the ViewRefresh table is also initialized to point to the current end of the log.

Eventually, the update log can get large. Garbage collection for the update log can be done simply by referring to all the FirstLROffset values in the ViewRefresh table. First of all, we need to figure out which log records can be eliminated from the log. They are those records stored at the offset that is less than the minimum of all the FirstLROffset values in the 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 minimum offset value. In Fig. 7, for example, since FirstLROffset of V_1 , which is 150, is the smallest, those records before offset 150 can be deleted. If that is done, the FirstLROffset values of V_1 , V_2 , V_3 , and V_4 will be adjusted to 0, 50, 850, and 1650, respectively.

With the above method alone, the problem of a huge log can still exist. Suppose some view V_5 which was materialized before has not been accessed for quite some time. Then, its FirstLROffset value could be much less than those of other views, and it is necessary to keep all the log records from the one at that offset on to refresh V_5 . This situation could cause too huge a log. To solve this problem, we need to consider not just the minimum of all the FirstLROffset values in the ViewRefresh table but the second minimum, the third minimum, and so on also when we discard the old log records. If the second or the third minimum is chosen, some views such as V_5 , can no longer be refreshed, which means that their materialization is given up. Since incremental refresh of such views would take longer than their recomputation from scratch and it is not useful to keep such infrequently accessed views materialized, this practice is not regarded as a loss. Assuming that a feasible size, L_0 , of the update log in bytes is given as a system parameter, garbage collection of the update log works as follows: Suppose the size of the current update log is L in bytes ($L > L_0$). All the FirstLROffset values in ViewRefresh table are sorted. If the minimum of them is greater than $L - L_0$, then it is chosen. Otherwise, the k -th minimum is chosen where $(k - 1)$ -st minimum $< L - L_0 < k$ -th minimum. All the log records stored at the offset that is less than the chosen minimum are deleted. Then, each FirstLROffset value in the ViewRefresh table is decremented by the chosen minimum offset value. Those tuples whose adjusted FirstLROffset values are negative are deleted from the ViewRefresh table along with their counterparts in the ViewInfo table.

2.4.2 Relevance types

While the update log is scanned, for each update log record, the corresponding update is checked if it is relevant to the view to be refreshed. In this checking process, the view's definition (i.e., ViewDef in the ViewInfo table) and DIDList in the ViewRefresh table of the view are referred to. ViewDef is needed to refer to condition P , condition element CE , and the set of target elements, TE , of the view. Given update log record U and view V , there are five types of relevance: MODIFY-M, MODIFY-I, MODIFY-D, INSERT, and DELETE, which are described in the following, where $U.x$ and $V.y$ denote the field x of U and some information y on V , respectively.

Type MODIFY-M:

If ($U.OpType = \text{'MODIFY'} \text{ AND } U.Ename \in V.TE \text{ AND } U.DID \in V.DIDList \text{ AND } (U.Ename \neq V.CE \text{ OR } V.P(U.Content))$), where $P(x)$ returns TRUE if x satisfies predicate condition P and returns FALSE otherwise, then this 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 ($U.OpType = \text{'MODIFY'}$), and that the document is still qualified for V despite the modification ($U.Ename \neq V.CE \text{ OR } V.P(U.Content)$). As such, it is necessary to reflect the same modification in the corresponding tuple of the ViewElem table.

Type MODIFY-I:

If ($U.OpType = \text{'MODIFY'} \text{ AND } U.Ename = V.CE \text{ AND } U.DID \notin V.DIDList \text{ AND } V.P(U.Content)$), then this 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 modifica-

tion ($U.OpType = \text{'MODIFY'}$) of an element which is the condition element of V ($U.Ename = V.CE$). As such, it is necessary to insert the tuples for V 's target elements of the modified document into the ViewElem table. The XMLElem table needs to be accessed through the index on DID to retrieve them unless V 's condition element is the only target element. For this type, $U.DID$ needs to be inserted into $V.DIDList$.

Type MODIFY-D:

If ($U.OpType = \text{'MODIFY'}$ AND $U.Ename = V.CE$ AND $U.DID \in V.DIDList$ AND (NOT $V.P(U.Content)$)), then this implies that the modified document which was qualified for V ($U.DID \in V.DIDList$) is now not qualified (NOT $V.P(U.Content)$) due to the modification ($U.OpType = \text{'MODIFY'}$) of its condition element ($U.Ename = V.CE$). As such, it is necessary to delete all the tuples for V 's target elements of the modified document from the ViewElem table. For this type, $U.DID$ needs to be deleted from $V.DIDList$.

Type INSERT:

If ($U.OpType = \text{'INSERT'}$ 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 quartet $\langle EID, ParentEID, Ename, Content \rangle$ in U whose Ename equals $V.CE$, then this implies that a new document was inserted ($U.OpType = \text{'INSERT'}$) which is qualified for V ($V.P(U.Content$, where $U.Ename = V.CE$)). As such, it is necessary to insert the tuples for V 's target elements of the inserted document into the ViewElem table. For this type, $U.DID$ needs to be inserted into $V.DIDList$.

Type DELETE:

If ($U.OpType = \text{'DELETE'}$ AND $U.DID \in V.DIDList$), then this implies that the document which was qualified for V ($U.DID \in V.DIDList$) was deleted ($U.OpType = \text{'DELETE'}$). As such, all the tuples for V 's target elements of the deleted document need to be deleted from the ViewElem table. For this type, $U.DID$ needs to be deleted from $V.DIDList$.

2.4.3 Generation of optimized SQL statements for view refresh

By examining the update log records and identifying the ones relevant to the view, we can generate the SQL statements needed to incrementally refresh the ViewElem table. Since we adopt the deferred view refresh policy, *optimization* in generating the SQL statements is possible in two ways.

First, the interrelated updates accumulated in the *same* document can be *logically* merged. For example, if a newly inserted document is deleted later, then neither the insertion nor the deletion needs to be reflected in the ViewElem table. There are 5 cases of such optimization:

- (i) **I-D** (a type INSERT or MODIFY-I update followed by a type DELETE update): neither update is reflected as if nothing happened.
- (ii) **M-D** (one or more type MODIFY-M updates followed by a type DELETE update):

- all the type MODIFY-M updates are ignored, and only the type DELETE update is reflected.
- (iii) **M-M** (a series of two or more type MODIFY-M updates on the same target element): only the last one is reflected.
 - (iv) **I-M** (a type INSERT update followed by one or more type MODIFY-M updates on the same target element): when the insertion is reflected, for the target element in question, the last modified value is used instead of the original one.
 - (v) **MI-MM** (a type MODIFY-I update followed by one or more type MODIFY-M updates): all the type MODIFY-M updates are ignored, and only the type MODIFY-I update is reflected.

All the optimizations are intuitively easy to understand except MI-MM optimization. The reason for MI-MM optimization is as follows. To reflect the type MODIFY-I update, the tuples for the target elements of the view out of the modified document are retrieved from the XMLElem table and inserted into the ViewElem table. The type MODIFY-M updates were done against the XMLElem table; therefore, they already affected on the retrieved tuples. As such, all the type MODIFY-M updates can be ignored.

In the second type of optimization, the same types of SQL statements can be *physically* merged. For example, all the SQL statements of the form “DELETE FROM ViewElem WHERE DID = x ” for reflecting type DELETE updates in the ViewElem table can be merged into one: “DELETE FROM ViewElem WHERE DID = x_1 OR ... OR DID = x_n .” After all, the number of SQL statements generated is at most three. One is to reflect type MODIFY-I updates, another one is for type INSERT and MODIFY-M updates, and the remaining one is for type DELETE and MODIFY-D updates.

The generated SQL statements are *optimal* in the sense that the total number of tuple insertions, deletions, and modifications in the ViewElem table is exactly the minimum required in the minimum number of SQL statements.

Fig. 8 shows the C-like pseudo code of algorithm *Gen_SQL4ViewRefresh* which generates the optimized SQL statements out of the logged updates. The input parameters for *Gen_SQL4ViewRefresh* are the identifier (ViewID), the source (ViewSrc), and the definition (ViewDef) of the view to be refreshed. It returns the generated SQL statements. Two types of data structures are needed to maintain the data necessary for generating the SQL statements and obtained while the update log is processed. One is *TNT*, which is a table used to buffer the new tuples to be inserted into the ViewElem table obtained from the log records of type INSERT or MODIFY-M updates. The schema of TNT is the same as that of the ViewElem table except that the ViewID column is replaced by the LSN column. The other is *Sx*, which is a set of identifiers of the documents involved in type x updates, where x represents all the relevance types. Let us denote *Sx*'s as *Si*, *Sd*, *Smi*, *Smd*, and *Smm* for types INSERT, DELETE, MODIFY-I, MODIFY-D, and MODIFY-M, respectively.

Gen_SQL4ViewRefresh works as follows: After initializing TNT and *Sx*'s as empty (*init_refreshinfo()*) and retrieving the FirstLROffset as well as the DIDList values from the ViewRefresh table, it freezes (i.e., blocks updates against) the view's source (*freeze()*). Then, it opens the update log (*open_updatelog()*), getting EOL as the offset to the current end of the log. It checks if there is any update in the log to examine. If none exists, it terminates by returning SQL statements which are null. Otherwise, it conducts a

```

Gen_SQL4ViewRefresh(ViewID, ViewSrc, ViewDef)
{
  init_refreshinfo();           /* initialization of TNT & Si, Sd, Smm, Smi, Smd as empty */
  LROffset = ViewRefresh[ViewID].FirstLROffset /* retrieval of FirstLROffset */
  DIDList = ViewRefresh[ViewID].DIDList;      /* DIDList from ViewRefresh table */
  freeze(ViewSrc);                          /* block updates against view's source */
  EOL = open_update_log(LROffset);           /* open update log for scan */
                                           /* EOL: offset to current end of log */
  IF (LROffset = EOL) RETURN (NULL);        /* update does not exist returns NULL
                                           SQL statement */

  DIDListUpdated = False;
  DO {
    LROffset = scan_update_log(LROffset, & u); /* retrieval of log record u */
    rel_type = check_relevance(ViewDef, DIDList, u)
                                           /* checking relevance between update and view */
    IF (rel_type != NULL) {                /* if relevant */
      CASE(rel_type) {
        WHEN(INSERT): Si = Si  $\cup$  {u.DID};
          gen_I_tuple(u);                 /* generate tuples for target elements out of u and
                                           tore them in TNT */
        WHEN(MODIFY-I): Smi = Smi  $\cup$  {u.DID};
        WHEN(MODIFY-M):
          IF (u.DID  $\in$  Smi) break;
          Smm = Smm  $\cup$  {u.DID};
          gen_MM_tuple(u);                /* generate modified tuple out of u and store it
                                           in TNT */
        WHEN(DELETE):
          IF (u.DID  $\in$  Smm) Smm = Smm - {u.DID};
          IF (u.DID  $\in$  Si) THEN Si = Si - {u.DID};
          ELSE IF (u.DID  $\in$  Smi) THEN Smi = Smi - {u.DID};
          ELSE Sd = Sd  $\cup$  {u.DID};
        WHEN(MODIFY-D):
          IF (u.DID  $\in$  Smm) Smm = Smm - {u.DID};
          IF (u.DID  $\in$  Si) THEN Si = Si - {u.DID};
          ELSE IF (u.DID  $\in$  Smi) THEN Smi = Smi - {u.DID};
          ELSE Smd = Smd  $\cup$  {u.DID};
      } /* end of CASE */
      IF (rel_type != MODIFY-M) DIDListUpdated = TRUE;
    }
  } WHILE (LROffset != EOL);              /* there exist more updates to examine */
  ViewRefresh[ViewID].FirstLROffset = end_of_log(); /* set to current end of log */
  IF (DIDListUpdated) ViewRefresh[ViewID].DIDList;
  RETURN (gen_sql(TNT, Si, Sd, Smi, Smd)); /* generate & return optimized SQL
                                           statements */
}

```

Fig. 8. Algorithm for generating the optimized SQL statements for view refresh.

log scan (`scan_updatelog()`) from the log record pointed to by `FirstLROffset` to the location whose offset is EOL. For each log record, it performs relevance checking (`check_relevance()`). The variable u is used as a parameter in both `scan_updatelog()` and `check_relevance()` to store the update log record that is currently being examined. The function `check_relevance()` returns rel_type , the relevance type value, 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. Note that `check_relevance()` also updates `DIDList` when it encounters a relevant update of all the relevance types except MODIFY-M. If the update is relevant, `Gen_SQL4ViewRefresh` updates TNT and Sx 's appropriately, depending on rel_type . When the location of the log whose offset is EOL is reached, it ends the log scan, thus modifying the `FirstLROffset` and the `DIDList` columns of the view's record in the `ViewRefresh` table. (The former is set to the current end of the log (`end_of_log()`), and the latter takes all the changes that were made during the log scan, if any.) Then, it generates and returns the optimized SQL statements with TNT and Sx 's (`gen_sql()`).

Update of TNT and Sx 's

For a log record u of the update relevant to view V , TNT and Sx 's are updated as follows: If u is for type a INSERT update, $u.DID$ is added to Si and then, for each target element of V , a tuple of TNT is generated out of u and inserted into TNT (`gen_I_tuple()`). Note that the schema of TNT is the same as that of the `ViewElem` table except that `ViewID` of the latter is replaced by LSN in TNT. As such, the inserted tuples store $u.LSN$ instead of the identifier of the view. These LSNs are needed later for M-M and/or I-M optimization. The tuples prepared in TNT will be inserted into the `ViewElem` table by means of an SQL statement.

If u is for type a MODIFY-I update, $u.DID$ is added to Smi . The tuples for the target elements of V out of the modified document need not be prepared now because the `XMLElem` table is to be accessed later to retrieve them by means of an SQL statement.

If u is for type a MODIFY-M update, it is first checked if the modified document is the one which has already undergone a type MODIFY-I update (i.e., $u.DID \in Smi$). If this is the case, then nothing is done because of the MI-MM optimization. Otherwise, $u.DID$ is added to Smm and then a tuple of TNT for the modification is generated out of u and inserted into TNT (`gen_MM_tuple()`).

If u is for a type DELETE or MODIFY-D update, it is first checked if the deleted or modified document is the one which has already undergone a type MODIFY-M update (i.e., $u.DID \in Smm$). If this is the case, then $u.DID$ is deleted from Smm for M-D optimization. After that, it is also checked if the deleted or modified document is the one which has already undergone a type INSERT or MODIFY-I update (i.e., $u.DID \in Si$ or Smi). If this is not the case, then $u.DID$ is added to Sd or Smd . Otherwise, $u.DID$ is deleted from Si or Smi for I-D optimization. Note that we also need to remove the tuples from TNT which had already been prepared for insertion into the `ViewElem` table when the earlier log record of type INSERT or MODIFY-M update was examined. However, TNT is intact at this moment. Deletion of such invalid tuples will be handled later.

Generation of SQL Statements

The function `gen_sql()` first sorts all the tuples in TNT on DID, then on EID, and

then on LSN, removing all the invalid tuples from TNT. The valid tuples are those whose DID belongs to $S_i \cup (S_{mm} - (S_d \cup S_{md}))$. Among the valid ones, if some of them have the same DID and the same EID, then this means that the same tuple of XMLElem will be modified more than once possibly after it is inserted with a type INSERT update. For those tuples of TNT, all except the one with the largest LSN are removed for I-M and/or M-M optimization.

Now, `gen_sql()` is ready to generate the three SQL statements. The first one is for type MODIFY-I updates and is generated as “*INSERT INTO ViewElem X,*” where *X* is the SQL expression needed to retrieve from the XMLElem table all the tuples for the target elements of *V* whose DID column belongs to *S_{mi}*.

The second one is for type INSERT and MODIFY-M updates and is generated as “*MERGE INTO ViewElem USING Temp4TNT X,*” where *Temp4TNT* is the temporary table linked to the source file storing the tuples of TNT. The MERGE statement is specified in the SQL99 standard and provided by off-the-shelf commercial ORDBMSs, such as Oracle, DB2, and so on. It performs the conventional SQL INSERT and UPDATE at the same time. For each tuple, say *t1* of Temp4TNT, if it matches one in ViewElem, say *t2*, in terms of the key fields (i.e., DID and EID), then *t2* is replaced by *t1*. Otherwise, *t1* is inserted into ViewElem. *X* is used to give this instruction and to link Temp4TNT to the source file.

The third one is for type DELETE and MODIFY-D updates and is generated as “*DELETE FROM ViewElem WHERE P,*” where the predicate *P* is a disjunction of the terms $DID = x$ such that $x \in (S_d \cup S_{md})$.

2.4.4 Example

In this subsection, we will give an example of deferred incremental refresh of an XML materialized view. Suppose we are given the ViewRefresh table and update log depicted in Fig. 9 when view V_1 in Fig. 4 is requested.

To return V_1 , its tuples in ViewElem table need to be refreshed first. Algorithm `Gen_SQL4ViewRefresh` searches the ViewRefresh table for V_1 's record, setting `LRoffset` to 150 and `DIDList` to {2}. Then, it opens the update log and starts the log scan from the record at offset 150. For each log record, it checks if the update at hand is relevant to V_1 , and if it is, then the data structures TNT and S_x 's are changed as described in section 2.4.3. Initially, TNT and S_x 's are empty. The first log record is:

```

<StartUpdateLog, 1, 1, ELEMENT, MODIFY>
<2, 1, title, 'Performance Evaluation of an XML Storage System'>
<EndUpdateLog>
```

It is for logging the modification of “title” element of the document whose DID equals 1. Since “title” element is the condition element (CE) of V_1 , V_1 's condition needs to be checked with the modified value (“Performance Evaluation of an XML Storage System”). It does not satisfy the condition. Besides, $1 \notin \text{DIDList}$. Thus, this modification is not relevant to V_1 .

ViewRefresh Table

ViewID	FirstLROffset	DIDList
V_1	150	{2}

Update Log

```

<StartUpdateLog, 1, 1, ELEMENT, MODIFY> ← offset 150
<2, 1, title, 'Performance Evaluation of an XML Storage System'>
<EndUpdateLog>
<StartUpdateLog, 2, 4, DOCUMENT, INSERT>
<1, 0, paper, '-'>
<2, 1, title, 'XML View and Its Refresh'>
<3, 1, author, 'J. Fisher'>
<4, 1, abstract, 'XML views can be materialized ...'>
<5, 1, keyword, 'XML, materialized view, incremental refresh ...'>
<6, 1, section, '1. Introduction'>
<7, 6, paragraph, 'For more cost-effective retrieval of XML documents ...'>
<8, 6, paragraph, 'The views and the objects ...'>
<EndUpdateLog>
<StartUpdateLog, 3, 1, ELEMENT, MODIFY>
<5, 1, keyword, 'XML, storage system, ...'>
<EndUpdateLog>
<StartUpdateLog, 4, 2, ELEMENT, MODIFY>
<5, 1, keyword, 'database snapshot, differential file, ...'>
<EndUpdateLog>
<StartUpdateLog, 5, 2, ELEMENT, MODIFY>
<3, 1, author, 'B. G. Lindsay et al.'>
<EndUpdateLog>
<StartUpdateLog, 6, 3, ELEMENT, MODIFY>
<3, 1, author, 'J. Fisher et al.'>
<EndUpdateLog>
<StartUpdateLog, 7, 2, ELEMENT, MODIFY>
<2, 1, title, 'A Snapshot Differential Maintenance Algorithm'>
<EndUpdateLog>
<StartUpdateLog, 8, 3, ELEMENT, MODIFY>
<2, 1, title, 'Document Links and View Refresh in XML Repository'>
<EndUpdateLog>
<StartUpdateLog, 9, 4, DOCUMENT, DELETE>
<EndUpdateLog>
(End of Update Log) ← offset 1800

```

Fig. 9. Example of ViewRefresh table and update log.

The second log record is:

```
<StartUpdateLog, 2, 4, DOCUMENT, INSERT>
<1, 0, paper, '-'>
<2, 1, title, 'XML View and Its Refresh'>
<3, 1, author, 'J. Fisher'>
<4, 1, abstract, 'XML views can be materialized ...'>
<5, 1, keyword, 'XML, materialized view, incremental refresh, ...'>
<6, 1, section, '1. Introduction'>
<7, 6, paragraph, 'For more cost-effective retrieval of XML documents ...'>
<8, 6, paragraph, 'The views and the objects ...'>
<EndUpdateLog>
```

It is for the insertion of a new document whose DID equals 4. Since V_1 's CE is "title," its value ("XML View and Its Refresh") is checked against the condition of V_1 . It satisfies the condition. In other words, the new document is qualified for V_1 . The relevance type here is "INSERT" as described in section 2.4.2. As such, V_1 's target elements (TE), which are "title," "author," and "abstract," need to be retrieved and inserted into the materialization of V_1 . From the target elements' corresponding quartets in the log record (i.e., <2, 1, title, "XML View and Its Refresh">, <3, 1, author, "J. Fisher">, and <4, 1, abstract, "XML views can be materialized ...">), three tuples (the first through the third one in Fig. 10) are generated and inserted into TNT. 4, the value of DID, is added to S_i , which becomes {4}. Also, 4 is added to DIDList, which then becomes {2, 4}.

LSN	DID	EID	ParentEID	Ename	Content
2	4	2	1	title	XML View and Its Refresh
2	4	3	1	author	J. Fisher
2	4	4	1	abstract	XML views can be materialized ...
5	2	3	1	author	B. G. Lindsay <i>et al.</i>

Fig. 10. TNT before calling *gen_sql()* of Fig. 8.

The third log record is:

```
<StartUpdateLog, 3, 1, ELEMENT, MODIFY>
<5, 1, keyword, 'XML, storage system, ...'>
<EndUpdateLog>
```

It is for the modification of the "keyword" element of the document whose DID equals 1. Since $1 \notin \text{DIDList}$ and "keyword" \neq CE, this modification is not relevant to V_1 .

The fourth log record is:

```
<StartUpdateLog, 4, 2, ELEMENT, MODIFY>
<5, 1, keyword, 'database snapshot, differential file, ...'>
<EndUpdateLog>
```

It is for the modification of the “keyword” element of the document whose DID equals 2. Although $2 \in \text{DIDList}$, this modification is not relevant to V_1 because “keyword” $\notin \text{TE}$.

The fifth log record is:

```
<StartUpdateLog, 5, 2, ELEMENT, MODIFY>
<3, 1, author, 'B. G. Lindsay et al.'>
<EndUpdateLog>
```

It is for the modification of the “author” element of the document whose DID equals 2. Since $2 \in \text{DIDList}$, “author” $\in \text{TE}$, and “author” $\neq \text{CE}$, it is necessary to reflect this modification in the materialization of V_1 . The relevance type here is “MODIFY-M.” Since $2 \notin \text{Smi}$, a record is added to TNT, which is the fourth one in Fig. 10. 2, the value of DID, is added to Smm , which becomes $\{2\}$.

The sixth log record is:

```
<StartUpdateLog, 6, 3, ELEMENT, MODIFY>
<3, 1, author, 'J. Fisher et al.'>
<EndUpdateLog>
```

It is for the modification of the “author” element of the document whose DID equals 3. Although “author” $\in \text{TE}$, this modification is not relevant to V_1 because $3 \notin \text{DIDList}$.

The seventh log record is:

```
<StartUpdateLog, 7, 2, ELEMENT, MODIFY>
<2, 1, title, 'A Snapshot Differential Maintenance ...'>
<EndUpdateLog>
```

It is for the modification of the “title” element of the document whose DID equals 2. Since $2 \in \text{DIDList}$ and “title” = CE, the modified value (“A Snapshot Differential Maintenance Algorithm”) needs to be checked against the condition of V_1 to see if this document is still qualified for V_1 . Now it does not satisfy V_1 's condition. Thus, it is necessary to delete V_1 's target elements of this document from the materialization of V_1 . The relevance type here is “MODIFY-D.” Since $2 \in \text{Smm}$, Smm now becomes $\{\}$, and since $2 \notin \text{Si}$ or Smi , Smd now becomes $\{2\}$. Also, 2 is deleted from DIDList , which then becomes $\{4\}$.

The eighth log record is:

```
<StartUpdateLog, 9, 3, ELEMENT, MODIFY>
<2, 1, title, 'Document Link and View Refresh in XML Repository'>
<EndUpdateLog>
```

It is for the modification of the “title” element of the document whose DID equals 3. Since “title” = CE, the modified value (“Document Link and View Refresh in XML Repository”) needs to be checked against the condition of V_1 . Since it satisfies the condition but the modified document was not qualified for V_1 before ($3 \notin \text{DIDList}$), all of V_1 's target elements of this document need to be inserted into the materialization of V_1 . The relevance type here is “MODIFY-I.” 3, the value of DID is added to S_{mi} , which becomes $\{3\}$. Also, 3 is added to DIDList, which then becomes $\{3, 4\}$.

Finally, the ninth log record is:

```
<StartUpdateLog, 9, 4, DOCUMENT, DELETE>
<EndUpdateLog>
```

It is logged for the deletion of the document whose DID equals 4. Since that document was the one qualified for V_1 ($4 \in \text{DIDList}$), all of V_1 's target elements need to be deleted from the materialization of V_1 . The relevance type here is “DELETE.” Since $4 \in S_i$, S_d remains the same, that is, $\{ \}$. Instead, 4 is deleted from S_i , which then becomes $\{ \}$. Also, 4 is deleted from DIDList, which then becomes $\{3\}$.

Fig. 10 shows TNT generated thus far. Meanwhile, $S_i = S_d = S_{mm} = \{ \}$, $S_{mi} = \{3\}$, and $S_{md} = \{2\}$. These are given to the function `gen_sql()`. Since $S_i \cup (S_{mm} - (S_d \cup S_{md})) = \{ \}$, none of the records in TNT need to be reflected in the ViewElem table. That is, the final TNT is empty. As such, only two SQL statements are generated. One is an INSERT statement used to insert into ViewElem table the tuples representing the target elements of V_1 , which are retrieved from the XMLElem table with the DID column equal to 3, the only element of S_{mi} . The other is a DELETE statement used to delete from the ViewElem table all the tuples whose DID is 2, the only element of $(S_d \cup S_{md})$.

Fig. 11 shows the XMLElem table after all the updates logged in the update log of Fig. 9 have been done, whereas Fig. 12 shows the ViewRefresh table and ViewElem table after V_1 is refreshed against the update log shown in of Fig. 9. Examining the XMLElem table in Fig. 11 and ViewElem table of Fig. 12, we can confirm that V_1 's deferred incremental refresh was correctly conducted.

2.4.5 Consistency of materialized views

In this subsection, we will explain why a materialized view refreshed with algorithm `Gen_SQL4ViewRefresh` of Fig. 8 is up-to-date. The consistency criteria for materialized views in this paper are *serializability*. It can be enforced by means of *two phase locking*. The process of returning requested materialized view V , including its incremental refresh, in a deferred way is a *transaction*. Let us call it V 's delivery transaction and denote it

DI D	EID	ParentEID	Ename	Content
1	1	0	paper	-
1	2	1	Title	Performance Evaluation of an XML Storage System
1	3	1	author	A. Smith and M. Jones
1	4	1	abstract	XML has emerged as a standard for Web documents ...
1	5	1	keyword	XML, storage system, ...
1	6	1	section	1. Introduction
1	7	6	paragraph	The Web was the most influential in advance of the internet ...
1	8	6	paragraph	The reason why Web has become the core of ...
1	9	6	paragraph	In most applications today, the Web-based user interface is ...
1	10	1	section	2. Types of XML Documents
1	11	10	paragraph	The XML documents are either with the DTD or without ...
1	12	10	paragraph	The valid XML documents are the ones that are ...
2	1	0	paper	-
2	2	1	title	A Snapshot Differential Maintenance Algorithm
2	3	1	author	B. G. Lindsay <i>et al.</i>
2	4	1	abstract	This article presents an algorithm to refresh the contents of database ...
2	5	1	keyword	database snapshot, differential file, ...
2	6	1	section	1. Introduction
2	7	6	paragraph	A DBMS provides a mechanism for maintaining, access, and updating ...
2	8	6	paragraph	The notion of a database snapshot was introduced in [ADIBA80] ...
2	9	1	section	2. Snapshot Refresh Objectives
2	10	9	paragraph	Snapshot refresh should make the snapshot reflect the current, ...
2	11	1	section	3. Alternative Refresh Methods
2	12	11	paragraph	Several alternatives are available for implementing snapshot refresh ...
2	13	11	paragraph	Another alternative is to buffer the changes to the base table and ...
3	1	0	paper	-
3	2	1	title	Document Links and View Refresh in XML Repository
3	3	1	author	J. Fisher <i>et al.</i>
3	4	1	abstract	Due to the proliferation of XML documents on the Web ...
3	5	1	keyword	XML, Web database, extended link, ...
3	6	1	section	1. Introduction
3	7	6	paragraph	The different between the conventional HTML links ...
3	8	6	paragraph	The virtual document can be implemented on the Web ...
3	9	6	paragraph	There are so many heterogeneous types of information on the Web ...
3	10	1	section	2. Related Work
3	11	10	paragraph	An XML document can represent the structure of ...
3	12	10	paragraph	The links of XML can use Xlink [6] and Xpointer [7] ...

Fig. 11. Updated XMLElem table.

as $d_T(V)$. It is performed to acquire locks (1) on the tuple of ViewRefresh table that corresponds to V in exclusive mode, (2) on the V 's source in shared mode, and (3) on V 's tuples in the ViewElem table in exclusive mode. The first and third exclusive locks are used to read FirstLROffset of V and update it after V 's refresh is finished, and to incrementally refresh V 's tuples in the ViewElem table, respectively. Since they are exclusive, at most one instance of $d_T(V)$ can be active at a time. The second lock is used to freeze (i.e., block updates against) V 's source while V is being refreshed and also to read V 's source to reflect MODIFY-I type updates in V .

ViewRefresh Table

ViewID	FirstLROffset	DIDList
V_1	1800	{3}

ViewElem Table

ViewID	DID	EID	ParentEID	Ename	Content
V_1	3	2	1	title	Document Links and View ...
V_1	3	3	1	author	J. Fisher <i>et al.</i>
V_1	3	4	1	abstract	Due to the proliferation of ...

Fig. 12. ViewRefresh table and ViewElem table after refresh of V_1 .

Once the first two locks are acquired, the update log is opened for scanning, and the scope of the update log used to examine V 's refresh is first determined. In Fig. 8, it extends from the log record pointed to by FirstLROffset of V to the current end of the log at the time when the log is open. The latter is with offset EOL. Let us assume that the last one of the previous $d_T(V)$'s had correctly updated FirstLROffset of V . Then, whether the refreshed V is consistent or not depends on whether the location with offset EOL is the right location at which to end the log scan or not. Given the correct FirstLROffset, to guarantee consistency of refreshed V , up to all those updates that were logged before the shared lock on V 's source is acquired should be examined. Such updates are part of the transactions which precede $d_T(V)$ in serializability order. Let x be the offset to the location of the log where logging of all those updates ends. Then, $EOL \geq x$ for the following two reasons: (1) Since all those updates are logged before their transactions release the exclusive lock on V 's source, when $d_T(V)$ opens the update log after it acquires the shared lock on the V 's source, they all are already in the log. (2) During the processing of the update log for V 's refresh, further updates on the XML source can take place and be logged, thus continually advancing the end of the log. If $EOL = x$, then obviously there will be no problem with the consistency of refreshed V . What if $EOL > x$? The answer is that there will be no problem in this case, either. Even if the log records beyond the location with offset x are included in the log scan, it will not do any harm because they are not the ones logged for the updates against V 's source, which has already been locked in shared mode; thus, they will turn out to be irrelevant to V .

The final thing to clarify is whether the update of FirstLROffset of V after the log scan in Fig. 8 is finished is correct. If it is not done correctly, then the consistency of V returned by the next $d_T(V)$'s will not be guaranteed. It is obviously correct to update it to EOL. In Fig. 8, however, it is set to the current end of the log at the time when the log scan is finished. Let y be the offset to that end. Since $y \geq EOL$ because of reason (2) above, it is better to advance FirstLROffset to y than to EOL as long as this does not cause a problem. As described for the updates logged in the offset range $[x, EOL)$ above, the ones in $[EOL, y)$ can never be relevant to V . Thus, they can be safely skipped for the next $d_T(V)$'s.

3. PERFORMANCE EVALUATION

Our proposed approach as described in the previous section was implemented in Java with the SAX parser [7] on top of Oracle 9i with the JDBC connection, resulting in a prototype XML storage system that supports view materialization running on a Windows 2000 Server. In this section, the results of performance experiments performed with the implemented system will be reported.

3.1 Overview

The *order* documents in XML of the TPC-W benchmark [33] were used in the experiments while assuming that a large number of them were stored in an XML warehouse supporting, say e-Commerce applications. An order document stores information about orders such as the customer, date, payment, and ordered items. Each order document used in the experiments was about 2KB long, excluding all the white spaces, and consisted of about 45 elements. An XML view against a collection of order documents was defined similarly to the one used as the running example in the previous section: They each had one condition element and three target elements. Table 1 shows the performance parameters, their descriptions, and the setting for the experiments. The experiments were conducted on a Pentium IV 2GHz PC with 512 MB of main memory and an E-IDE 80GB disk, running Windows 2000 Server. The experimental results were obtained by averaging 5 measurements for each experiment.

Table 1. Performance parameters.

Parameter	Description	Setting
D	The number of base XML documents	5K, 10K, 15K, 20K, 25K, 30K
U	The proportion of base document updates	0.01, 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

The major goal of our experiments was to figure out under what condition incremental refresh of the materialized view outperforms *view recomputation* whereby the XML views are regenerated from scratch by some XML query processing algorithm every time they are requested. The most influential performance parameter in this regard

is the number of logged updates to be examined for deferred incremental refresh. As such, in our experiments, we assumed that the number of base XML documents remained the same along with the numbers of document insertions and deletions. We also assumed that the size of the retrieved view was the same all the time. These assumptions were made so that the view recomputation time would remain virtually the same despite the updates done to the base documents, whereas the time with incremental refresh would increase as more updates were done. To achieve this goal, the numbers of document insertions and of deletions were kept the same (i.e., $I = D$), and the number of insertions relevant to the view and that of deletions relevant to the view were also kept the same (i.e., $I \times R = D \times R$). The number of element modifications of the MODIFY-I relevance type and that of the MODIFY-D type were also kept the same (i.e., $MI = MD$). One more assumption in the experiments was that the relevance ratio was the same as the view selectivity (i.e., $R = S$).

Another aspect of our experiments was the control of update accumulation the same document. As described in section 2.4.3, the interrelationships among the updates to the same document are considered while processing the update log to efficiently refresh the materialized view. In the experiments, four disjoint sets of documents were managed. With respect to a materialized XML view, there were four categories of source documents in the XML warehouse. The first one was for documents qualified for the view with their target elements already in the view. The second one was for documents qualified for the view but inserted after the view was materialized. The third and fourth ones were the opposite counterparts of the first and the second, respectively. When any type of XML update occurred, be it relevant to the view or not, we chose the document involved *randomly*. In this way, the identifier of a deleted document was not reused when some document was inserted into the warehouse.

3.2 View Retrieval Time

The time for XML view retrieval with incremental refresh of the materialized view and that with view recomputation were measured for the purpose of comparison. Figs. 13 and 14 compare the view retrieval times obtained with these two methods as the proportion of base document updates (U) increased. The number of base documents (D) was set to 10,000. We note that as U increased, the time with view recomputation did not change, whereas that with incremental refresh increased. As Fig. 13 shows, incremental refresh outperformed view recomputation as long as U was less than about 25% ($S = 0.2$) and about 23% ($S = 0.3$). This *upper limit* on the percentage of updates for incremental refresh to be more effective than view recomputation increased as shown in Fig. 14 to about 47% ($S = 0.2$) and about 38% ($S = 0.3$).

Figs. 15 and 16 compare the view retrieval times as the number of base documents (D) increased. U was set to 20% for incremental refresh. The size of the view increased as D increased. As such, we note that both the times with view recomputation and incremental refresh increased as D increased. However, we can observe that incremental refresh outperformed view recomputation all the time and was less sensitive to the increase of the size of the view's source.

These results imply that XML view materialization can be very effective in providing views against a large-scale XML warehouse on the Web.

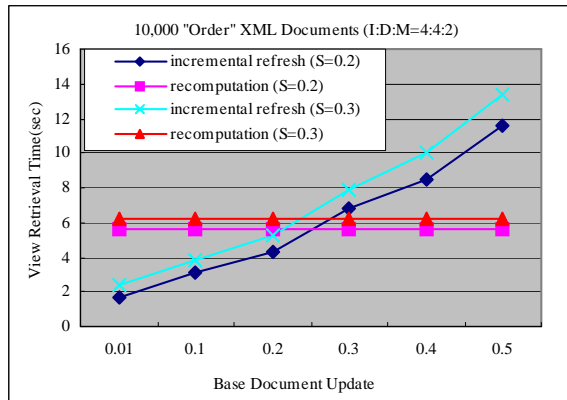


Fig. 13. View retrieval time w.r.t varying proportion of base document updates.

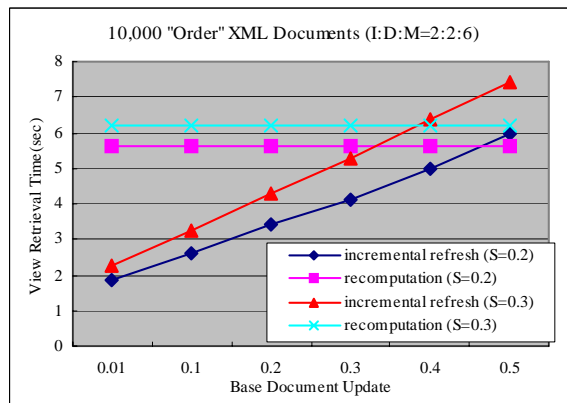


Fig. 14. View retrieval time w.r.t varying proportion of base document updates.

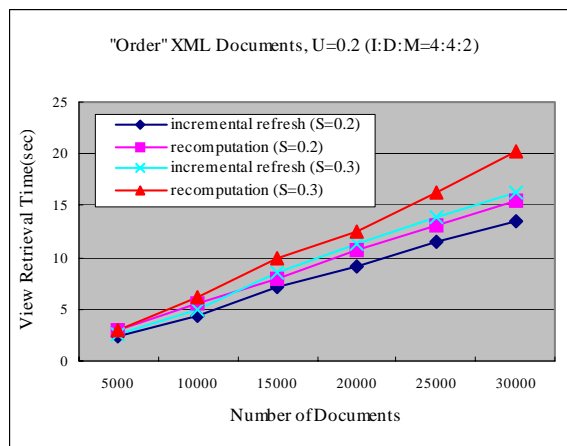


Fig. 15. View retrieval time w.r.t varying number of base documents.

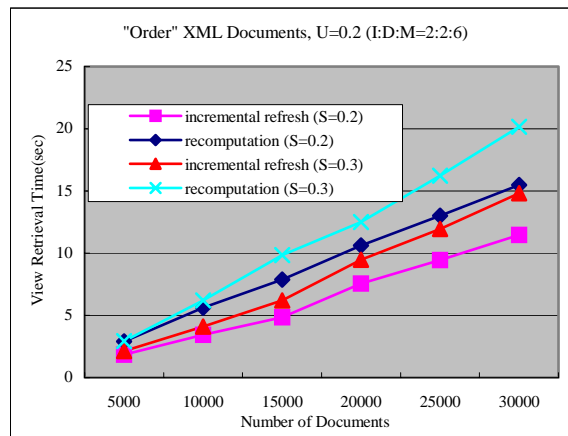


Fig. 16. View retrieval time w.r.t varying number of base documents.

3.3 Other Measurements

There are three tasks involved in view retrieval with incremental refresh of the materialized view: (1) scanning the update log, (2) refreshing the ViewElem table by processing the log, and (3) retrieving the view's tuples from the ViewElem table and tagging them in XML. Fig. 17 compares the time it took for each of these tasks as the proportion of base document updates (U) increased. D was set to 10,000, S was set to 0.2, and the ratio $I:D:M$ was set to 4:4:2. We note that the most time consuming parts were the first and second tasks, which correspond to incremental refresh of the materialized view. We also note that the disk I/O for log scanning took much time, especially when U increased.

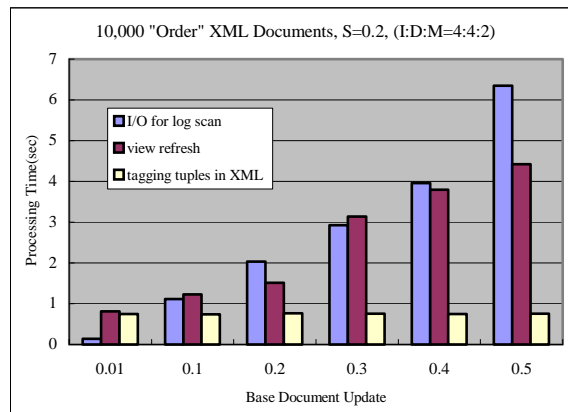


Fig. 17. Time for each task in view retrieval through incremental refresh of materialized view.

The aforementioned second task in view retrieval in turn consists of the following four subtasks: (1) reading DIDList from the ViewRefresh table, (2) the checking rele-

vance between the logged updates and the view, and generating SQL statements, (3) restoring the updated DIDList back to the ViewRefresh table, and finally (4) executing the generated SQL statements against the ViewElem table. Fig. 18 compares the time it took for each subtask and shows that the time for executing the SQL statements was dominant, whereas the other overheads were negligible.

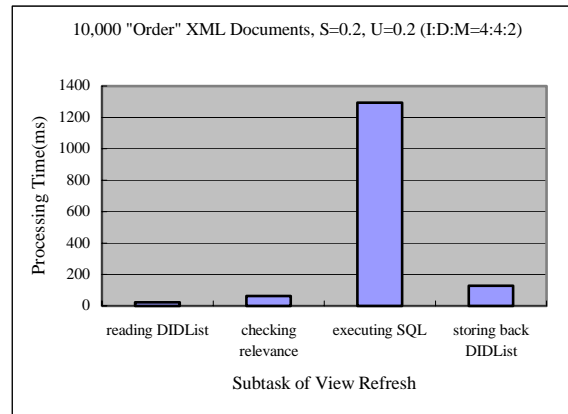


Fig. 18. Time for each task in view refresh.

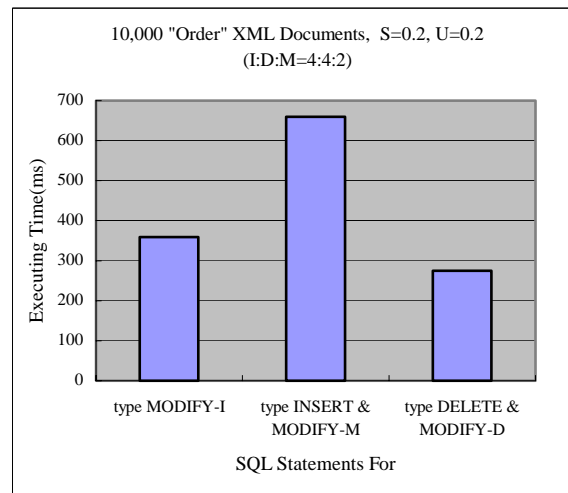


Fig. 19. Time for SQL execution in view refresh.

As described in section 2.4.3, at most three SQL statements are generated to refresh the ViewElem table. Fig. 19 compares the time it took to execute each SQL statement. The MERGE statement for type INSERT and MODIFY-M updates took the most time. That is because of the additional disk I/O required to move the new tuples to be inserted

into the ViewElem table stored in the data structure TNT to a temporary file linked to the MERGE statement. The INSERT statement for type MODIFY-I updates took more than the DELETE statement for type DELETE and MODIFY-D updates. That is because the former accesses the XMLElem table as well as the ViewElem table.

Finally, Fig. 20 shows the logging overhead incurred to support materialized views when performing XML updates to the XMLElem table. The time it took for document insertion, deletion, or modification when there was no need to support materialized views was normalized to 1. As shown, the time it took to append a log record at the end of the update log was quite negligible.

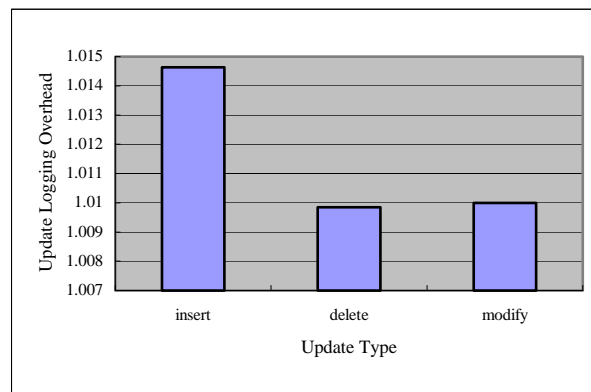


Fig. 20. Update logging overhead.

4. RELATED WORK

The problem of incremental refresh of materialized views has received much attention in studies on relational database systems [18]. The same problem for XML data, however, has rarely been addressed. To the best of our knowledge, [27] is the only work on XML view materialization in the literature. It proposed a technique for checking the relevance between an XML update and an XML view defined in a subset of XQL. For an XML materialized view over an XML source document which is represented as a DOM tree, an auxiliary information structure called the aggregate path index (APIX) [10] is employed to check the update's relevance to the view. The APIX consists of all the identifiers of the DOM tree nodes which correspond to the XML elements involved in the XQL path expression for defining the view. Its data structure is a hierarchical one representing the parent-child relationship among the nodes whose identifiers are in the APIX. The APIX is generated when the view is initially computed and maintained against the subsequent updates on the XML source. For the update relevant to the view, not just the APIX but the materialized view are fixed. For the latter, the maintenance statements which possibly access the XML source are generated.

The problems with the technique in [27] are that it requires too much space for APIXs, which are basically the auxiliary information for maintaining the materialized views, and that it is not a scalable solution. The size of the APIX depends on the view's

source document instance because it stores the identifiers of all the nodes relevant to the view occurring in the source document. As such, its space requirement might be big depending on the view definition and the source document instance. Secondly, when there are n source documents, n APIXs need to be maintained, including the APIXs for the source documents not qualified for the view because an APIX is needed for each source document. In the XML warehouse environment on the Web, n might be very large. If there are m materialized views, the total number of APIXs needed will be $m \times n$. Thirdly, whenever an XML update occurs in a source document, m APIXs that correspond to the updated document are accessed and changed appropriately in order to refresh m materialized views. If m is large, this is obviously not feasible, and thus, the deferred update propagation for each APIX should be employed. However, just the naïve repetition of the proposed APIX maintenance along with the materialized view refresh for a set of logged updates will be inefficient because the APIX as well as the XML source will be redundantly accessed several times, in proportion to the number of logged updates. Meanwhile, the issue of the storage format of materialized views in connection with the storage format of the XML source was not addressed in [27].

Unlike the technique in [27], our scheme is scalable. With deferred update propagation, where the optimized SQL statements used to refresh the materialized view are generated, our approach can support a number of materialized views over a huge collection of XML documents. The way XML materialized views are stored in our scheme is also viable. Storing them in the ViewElem table by caching their corresponding source tuples from the XMLElem table, returning a view and fixing a small portion of it for incremental refresh are efficiently supported. Mapping between the source data objects and those in materialized views is naturally accomplished.

A few studies on view materialization in *semistructured* databases were conducted in the late 90s. In [30], incremental refresh of materialized views over a semistructured database of rooted trees with labelled edges was investigated. The views considered were defined in UnQL [9]. The updates considered are the insertion of a tree into another one as a subtree of one of its nodes, and the replacement of a subtree with a new tree. When an update of the data source occurs, this information is sent 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.

In [39], incremental refresh of materialized views over a graph-structured database was investigated. An example of such a database is linked Web pages, and any database that can be modelled as a set of objects (nodes) with pointers (edges) is applicable. The views considered are the ones defined in an extended OQL, and the materialized view is represented as a set of objects satisfying the view's condition without links (i.e., edges) among them. The updates to the data source considered include edge insertion between two objects, edge deletion, and modification of the atomic object's value. 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 into or deleted from the materialized view.

In [3], incremental refresh of the materialized views over semistructured data in the Object Exchange Model [24] was investigated. The views considered were the ones defined in an extended version of Lorel [2], the query language of Lore [23]. The representation of a materialized view is the same to that in [39] except that the edges among ob-

jects are included. The model of update to the data source is the same as that in [39], and view refresh is also done similarly. With an 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 in the materialized view.

These previous works are limited for a few reasons. First, the technique proposed in [30] does not support the join view nor value modification of the data source. The techniques proposed in [39] and [3] do not consider the insertion/deletion of objects for updating. The view considered in [39] is just a set of objects without consideration of the structural relationships among them, and thus, not applicable to the XML view exported in XML for Web applications. Secondly, like the technique proposed in [27], they are not scalable solutions. There should be some efficient mechanism that can employ the deferred update propagation policy with the proposed view refresh techniques. Finally, because of the semistructured database context in which these works were developed, the proposed techniques are infeasible for implementation with an RDBMS or an ORDBMS. Rather, they lend themselves to view materialization in native XML databases.

5. CONCLUSIONS AND FUTURE WORKS

In this paper, we have investigated the XML view, its materialization, and deferred incremental refresh for the case of a restricted class of views. The three major issues in XML view materialization, which are consistency, storage structure, and scalability, have been dealt with together. The contributions of this paper are as follows:

1. An *object-relational* database schema employing collection and structured type constructors for storing materialized views derived from source XML documents and other auxiliary information for view refresh has been designed. Storing XML materialized views in a DBMS table by caching their corresponding source tuples, returning a view, and fixing a small portion of it for incremental refresh can be efficiently supported.
2. An algorithm that scans logged updates, checks their relevance to the view to be refreshed, identifies the interrelationships among them to generate *optimized SQL99* statements for deferred incremental refresh of the materialized view has been developed.
3. A *scalable* solution for XML view materialization has been proposed whereby the volume of the view's source can be huge and the number of materialized views can also be huge, perhaps on the Internet scale, with incurring negligible overhead in performing updates to the XML source.
4. A prototype XML storage system supporting view materialization has been implemented, and the results of performance experiments have been reported.

There are several avenues for future work. First, a scheme for efficient logging of document/element insertions or element modifications needs to be devised. This is necessary to avoid log records that are very large size when inserting a large document/element or modifying a large element. In fact, as shown in Fig. 17, the time needed for log scanning in view refresh is relatively large when document insertions are the ma-

for updates. Note that document insertions may be the most dominant type of update in the XML warehouse environment. The logged data values of elements are redundantly stored in corresponding base documents. As such, some referencing mechanism where a log record points to its relevant portion of the base document is desirable. However, such an update logging scheme would necessitate more accesses to the base documents during log processing for view refresh. The time and space tradeoffs here need to be carefully examined.

Secondly, performance analysis needs to be conducted on the view retrieval times with incremental refresh of the materialized view and with recomputation of the view in order to derive equations for the metadata maintained in the XML warehouse, whose values are a priori known or can be estimated. For the case of incremental refresh, the number of logged updates to be scanned might be the most important metadata. With such equations, given a request for an XML view that is maintained as a materialized one, the XML query optimizer can always choose the winner of incremental refresh and recomputation.

Finally and most importantly, the restrictions imposed on the views listed in section 2.1 need to be removed. They are that: (1) the tag names of condition and target elements are unique within a base document, (2) the target elements are leaves, (3) there is just one condition element with at most one instance of it per base document, (4) the views' source updates are done only with document insertion/deletions and modifications of each leaf element's text. In what follows, we will briefly explain how the system described in section 2 can be extended to remove all these restrictions.

The first restriction can be removed by simply using the *element path* instead of the tag name when designating a particular element. To remove the second restriction, we need to treat the modification of any subelement of a non-leaf target as a type MODIFY-M update. To remove the third restriction, we first need to consider the case where there is still one condition element but possibly more than one instance of it per base document. To support this case, the DIDList column of ViewRefresh table needs to be extended to a set of (DID, c) pairs, where c is the *counter* recording the number of the condition element instances in that particular document which satisfy the view's filtering condition. Also for element modification, the *before image* of the modified element as well as the after one needs to be logged.

For an INSERT type update, its DID is added to DIDList and its counter c initialized to k , where k is the number of condition element instances satisfying the view's condition ($k \geq 1$). For a MODIFY-I type update, the DID of the modified document is added to the DIDList and its counter c initialized to 1, and it is treated as a normal MODIFY-I update when refreshing the view. For the modification of a condition element where the new value satisfies the view's condition while the old does not, if the DID of the modified document is already in DIDList, then its counter c is incremented by 1, and the modification is ignored in further processing for view refresh. For a MODIFY-D type update, c is decremented by 1, and if c is still greater than or equal to 1, then the modification is ignored when refreshing the view. Only when c becomes 0 is the modification treated as a normal MODIFY-D update. For a DELETE type update, its (DID, c) pair is removed from DIDList as usual regardless of the value of c .

Now we can deal with an arbitrary document filtering condition to completely remove the third restriction. Suppose it is given in the *disjunctive normal form* as " T_1

OR ... OR T_n ," where T_i is " $t_{i,1}$ AND ... AND t_{i,m_i} " ($m_i \geq 1$ and $i = 1, \dots, n$). Each term $t_{i,j}$ is for one condition element, and different terms may be on the same condition element. To support this arbitrary condition, the CE column is used to store a list of 3-tuples (Epath, i , j) which corresponds to term $t_{i,j}$ of the view's filtering condition, where Epath is the path to the corresponding condition element within the source document. The total number of 3-tuples in the CE column is $\sum_{i=1}^n m_i$. In addition, the same number of predi-

cates are stored in the P column to be matched with its corresponding 3-tuple of the CE column. The DIDList is a set $\{(DID, \text{flag}, C_1, \dots, C_n) \mid \text{flag} \in \{\text{TRUE}, \text{FALSE}\}, \text{ and } C_i \text{ is the counter vector } (c_{i,1}, \dots, c_{i,m_i}), i = 1, \dots, n\}$. Let us call each element of DIDList a *DID entry*. The flag is used to denote whether the corresponding base document is qualified for the view (i.e., TRUE) or not (i.e., FALSE). Each counter $c_{i,j}$ in counter vector C_i is equal to 0 if the corresponding condition element does not satisfy its predicate in the P column of the ViewInfo table. Otherwise, it is greater than or equal to 1.

XML updates to the view's source are performed to make changes to these counters. There are three basic rules for maintaining DIDList: (1) If $c_{i,1} = \dots = c_{i,m_i} = 0$, then C_i is not represented in its DID entry. (2) If there is at least one C_i whose m_i counters are all greater than or equal to 1, then the flag is set to TRUE. Otherwise, it is set to FALSE. (3) If none of the C_i 's is represented, then their DID entry is removed from DIDList.

For a logged insertion of base document D, D is first evaluated against the view's document filtering condition represented in the CE/P columns of ViewInfo table. D's DID entry is added to DIDList and its counter vectors initialized unless the above rule (3) applies, and the flag is set according to rule (2). If the flag is TRUE, the insertion is treated as a normal INSERT type update when refreshing the view. Otherwise, it is ignored in further processing for view refresh. For a DELETE type update, its DID entry is removed from DIDList as usual regardless of the flag and the counter vectors.

For a logged modification of condition element E in base document D, there can be many cases depending on (i) whether D's DID entry is in DIDList or not, (ii) whether the flag is TRUE or FALSE if the DID entry is in DIDList, and (iii) whether the before and after images of the modification satisfy E's predicate. Among them, only 5 cases result in changes to DIDList. One of them is when D's DID entry is not in DIDList and the new value of E satisfies E's predicate. In this case, D's DID entry is added to DIDList with just *one* counter vector represented such that only *one* counter is set to 1 and all others are set to 0, if any (i.e., $(0, \dots, 0, 1, 0, \dots, 0)$). The flag is set to TRUE if the represented counter vector consists of just one counter, which is 1. Otherwise, it is set to FALSE. The remaining 4 cases are when D's DID entry is already in DIDList. Depending on the flag and the before/after images of E, E's counter is either incremented or decremented by 1, and then the aforementioned three rules of DIDList maintenance are applied. Finally, E's modification that changes the flag from FALSE to TRUE is treated as a normal MODIFY-I type update when refreshing the view. E's modification that changes the flag from TRUE to FALSE or that removes D's DID entry from DIDList is treated as a normal MODIFY-D type update when refreshing the view. Others are ignored in further processing for view refresh.

For a logged modification of element E, which is a target element, or of its subelement in base document D regardless of whether E is also a condition element or not, if

D's DID entry is in DIDList and the flag is TRUE before and after the modification, then it is treated as a normal MODIFY-M type update when refreshing the view.

To remove the fourth restriction, we need to support *leaf* element insertions and deletions since leaf element updates are the basic building blocks for arbitrary XML updates. One thing to note is that with element insertions and deletions, *structural changes* occur in the XML document, and renumbering of the EIDs assigned in the preorder traversal of the XML tree for the document as described in section 2.2 might be inevitable. As such, the XML numbering scheme which avoids renumbering of the EIDs altogether or as much as possible against any XML updates is desirable. This scheme is called *update robust* [6, 11, 13, 21, 26]. Since the issue of update robustness is orthogonal to the work in this paper, we just assume the use of any of the above referenced schemes except [13], which is not compatible with the EID assignment in this paper.

For a logged insertion of leaf element E in base document D, if E is a condition element, then it is regarded as a *modification*, where its before image is the one that does not satisfy E's predicate, possibly resulting in changes to DIDList as described above. As such, it may be either irrelevant to the view or ignored in further processing for view refresh, or treated as a normal MODIFY-I type update when refreshing the view. If E is a target element or its subelement, regardless of whether E is also a condition element or not, if D's DID entry is in DIDList, and if the flag is TRUE before and after the modification, then it is treated as a MODIFY-M type update. Although there is no E's corresponding *old* tuple in the ViewElem table that needs to be modified (i.e., replaced by the tuple for E), the MERGE SQL statement is generated to insert E's tuple when it reflects all the INSERT and MODIFY-M type updates into the ViewElem table (refer to section 2.4.3).

For a logged deletion of leaf element E from base document D, if E is a condition element and D's DID entry is in DIDList, then it is regarded as a *modification*, where its after image is the one that does not satisfy E's predicate, possibly resulting in changes to DIDList as described above. As such, it can be either irrelevant to the view or ignored in further processing for view refresh, or treated as a normal MODIFY-D type update when refreshing the view. If E is a target element or its subelement, regardless of whether E is also a condition element or not, if D's DID entry is in DIDList, and if the flag is TRUE before and after the modification, then only E's corresponding tuple in ViewElem needs to be deleted. We need a new relevance type called *E-DELETE* here because none of the existing ones is suitable for this type of refresh.

REFERENCES

1. S. Abiteboul, "On views and XML," in *Proceedings of ACM Symposium on Principles of Database System*, 1999, pp. 1-9.
2. S. Abiteboul *et al.*, "The Lorel query language for semistructured data," *Journal of Digital Libraries*, Vol. 1, 1996, pp. 68-88.
3. S. Abiteboul *et al.*, "Incremental maintenance for materialized views over semistructured data," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 1998, pp. 38-49.
4. S. Abiteboul *et al.*, "Active views for electronic commerce," in *Proceeding of Inter-*

- national Conference on Very Large Data Bases (VLDB)*, 1999, pp. 138-149.
5. S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, D. Srivastava, and Y. Wu, "Structural joins: a primitive for efficient XML query pattern matching," in *Proceedings of International Conference on Data Engineering*, 2002, pp. 141-152.
 6. T. Amagasa *et al.*, "QRS: a robust numbering scheme for XML documents," in *Proceedings of International Conference on Data Engineering*, 2003, pp. 705-707.
 7. Apache XML, <http://xml.apache.org/xerces-j/>.
 8. N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: optimal XML pattern matching," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2002, pp. 310-321.
 9. P. Buneman *et al.*, "Programming constructs for unstructured data," in *Proceedings of International Workshop on Database Programming Languages (DBPL)*, 1995.
 10. L. Chen and E. Rundensteiner, "Aggregate path index for incremental web view maintenance," in *Proceedings of 2nd International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*, 2000, pp. 231-238.
 11. S. Chien *et al.*, "Storing and querying multiversion XML documents using durable node numbers," in *Proceedings of 2nd International Conference on Web Information Systems Engineering*, 2001, pp. 232-241.
 12. S. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo, "Efficient structural joins on indexed XML documents," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2002, pp. 263-274.
 13. E. Cohen *et al.*, "Labeling dynamic XML trees," in *Proceedings of ACM International Symposium on Principles of Database Systems (PODS)*, 2002, pp. 271-281.
 14. S. Cluet *et al.*, "Views in a large scale XML repository," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2001, pp. 271-280.
 15. A. Deutsch *et al.*, "Storing semistructured data with STORED," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1999, pp. 431-442.
 16. D. Florescu and D. Kossmann, "Storing and querying XML data using an RDBMS," *IEEE Data Engineering Bulletin*, Vol. 22, 1999, pp. 27-34.
 17. D. Florescu and D. Kossmann, "A performance evaluation of alternative mapping schemes for storing XML data in a relational database," Technical Report, No. 3680, INRIA, France, 1999.
 18. A. Gupta and I. Mumick, *Materialized Views: Techniques, Implementations, and Applications*, MIT Press, 1999.
 19. A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. Rao, F. Tian, S. Viglas, Y. Wang, J. Naughton, and D. DeWitt, "Mixed mode XML query processing," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2003, pp. 225-236.
 20. H. Jiang, W. Wang, H. Lu, and J. Yu, "Holistic twig joins on indexed XML documents," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2003, pp. 273-284.
 21. Q. Li and B. Moon, "Indexing and querying XML data for regular path expressions," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2001, pp. 361-370.
 22. J. McHugh and J. Widom, "Query optimization for XML," in *Proceedings of Inter-*

- national Conference on Very Large Data Bases (VLDB)*, 1999, pp. 315-326.
23. J. McHugh *et al.*, "Lore: a database management system for semistructured data," *SIGMOD Record*, Vol. 26, 1997, pp. 54-66.
 24. Y. Papakonstantinou *et al.*, "Object exchange across heterogeneous information sources," in *Proceedings of International Conference on Data Engineering*, 1995, pp. 251-260.
 25. Y. Papakonstantinou and V. Vianu, "DTD inference for views of XML data," in *Proceedings of 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 2000, pp. 35-46.
 26. J. Park and H. Kang, "Handling updates for cache-answerability of XML queries on the web," in *Proceedings of British National Conference on Databases*, 2004, pp. 124-135.
 27. L. Quan *et al.*, "Argos: efficient refresh in an XQL-based web caching system," in *Proceedings of Workshop on the Web and Databases*, 2000, pp. 23-28.
 28. N. Roussopoulos, "Materialized views and data warehouses," in *Proceedings of 4th Workshop on Knowledge Representation Meets Databases (KRDB)*, 1997, pp. 12.1-12.6.
 29. J. Shanmugasundaram *et al.*, "Relational databases for querying XML documents: limitations and opportunities," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 1999, pp. 302-314.
 30. D. Suciu, "Query decomposition and view maintenance for query languages for unstructured data," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 1996, pp. 227-238.
 31. I. Tatarinov *et al.*, "Storing and querying ordered XML using a relational database system," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2002, pp. 204-215.
 32. F. Tian *et al.*, "The design and performance evaluation of alternative XML storage strategies," *ACM SIGMOD Record*, Vol. 31, 2002, pp. 5-10.
 33. TPC-W, *Transaction Processing Performance Council*, <http://www.tpc.org/tpcw/>.
 34. XML.org, <http://www.xml.org>.
 35. L. Xyleme, "A dynamic warehouse for XML data of the web," *IEEE Data Engineering Bulletin*, Vol. 24, 2001, pp. 40-47.
 36. Xyleme, <http://www.xyleme.com>.
 37. M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura, "XRel: a path-based approach to storage and retrieval of XML documents using relational databases," *ACM Transactions on Internet Technology*, Vol. 1, 2001, pp. 110-141.
 38. A. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman, "On supporting containment queries in relational database management systems," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2001, pp. 425-436.
 39. Y. Zhuge and H. Garcia-Molina, "Graph structured views and their incremental maintenance," in *Proceedings of International Conference on Data Engineering*, 1998, pp. 116-125.



Dae Hyun Hwang received the B.E. and M.S. degrees in Computer Science from Chung-Ang University, Seoul, Korea in 1992 and 1995, respectively. He is currently a Ph.D. student in Computer Science and Engineering at Chung-Ang University. Before he started the Ph.D. program in 2002, he had worked as a software research and development engineer in Korea Electronic Power Data Network, Inc., Seoul, Korea from 1995 to 2000, and in News Design, Inc., Ansan, Korea from 2000 to 2002. His research interests include XML query processing, XML update, and Web database. In particular, he focuses on materialized views for XML warehouse on the Web.



Hyunchul Kang received the B.E. degree in Computer Engineering from Seoul National University, Seoul, Korea in 1983, and received the M.S. and Ph.D. degrees in Computer Science from the University of Maryland, College Park, U.S.A. in 1985 and 1987, respectively. In 1988, he joined the School of Computer Science and Engineering, Chung-Ang University, Seoul, Korea where he is currently a Professor. His current research interests include materialized view, XML and Web data management, stream data management, and mobile and embedded data management.