

AN EFFICIENT DATA WAREHOUSING FRAMEWORK

Yousry Taha, Arsany S. Sawiros, Noha Adly

Department of Computer Science, Faculty of Engineering, Alexandria University

Alexandria, 21544

Egypt

taha@ccis.ksu.edu.sa, a.sawiros@menanet.net, noha@dataxprs.com.eg

ABSTRACT

This paper introduces a data-warehousing framework that is designed in the context of distributed objects, thus it has the benefits of scalability, interoperability, and support for heterogeneous environments. The proposed framework adopts an efficient incremental lightweight view maintenance technique that is motivated by the fact that different views in a Data Warehouse (DW) can have different freshness requirements. This fact can be used to enhance the view maintenance performance in huge DWs by installing the source updates only when the freshness constraints are violated and only for the DW views that have the violated constraints. The proposed view maintenance strategy guarantees strong consistency for each view and doesn't require the DW sources to be quiescent in order to complete the view maintenance.

1. INTRODUCTION

Data Warehousing is used for reducing the load of on-line transactional systems by extracting and storing the data needed for analytical purposes [3] such as On-Line Analytical Processing (OLAP) and Data Mining (DM) which have become essential for Decision Support Systems (DSS).

The general architecture of a Data Warehouse Management System DWMS is introduced in [4]. The basic task, for which the different components cooperate, is to maintain the views of the DW to reflect the updates that take place in the sources. Several decisions have to be taken to specify a view maintenance strategy. Among those decisions, the one we call *the maintenance perspective*: the maintenance can be done from two different points of view. (i) The point-of-view of every source update, this is the traditional perspective. In this approach, an arriving source update is processed once by in-

stalling it in all the relevant DW views. We call this: *per-update maintenance*. (ii) The point-of-view of every DW view. This is the approach used in our framework. In this approach, every DW view is refreshed by picking and installing the relevant updates from the buffer of updates ; so, a single update can be installed in some views and left in the buffer of updates for later installation in other views. When will an update U be installed in a view V ? depends on the freshness constraints defined on V . We call this approach *per-view maintenance*. This per-view constraint-based deferred maintenance saves a large maintenance overhead by installing the source updates only when the freshness constraints are violated and only for the DW views that have the violated constraints. Defining different freshness constraints for different views realizes the concept of unequally 'important' views suggested in [1].

The framework proposed provides (i) A distributed-object-based architecture and (ii) A per-view constraint-based view maintenance strategy. The employment of distributed object technology with the proposed architecture offers several benefits [2,5] such as: plug-and-play modularity, scalability, interoperability, and support for heterogeneous sources.

The rest of this paper is organized as follows: section 2 describes the data warehouse model. Section 3 describes the architecture objects and modules and the interaction among them. Section 4 describes the view maintenance technique. Section 5 concludes the paper and suggests some future work.

2. THE DATA WAREHOUSE MODEL

The DW is composed of a set of materialized views. The views are relations in a relational data-

base. Each view is defined in terms of other objects; the latter could be source relations as well as other data warehouse views. For each DW view, let $V = \langle D, F \rangle$ be the view's descriptor where:

D: Dependency predicate. This is the definition of the view in terms of data sources or other views.

F: Freshness predicate Describes the freshness constraints imposed on the DW view.

The dependencies among the different objects, including both source relations and DW views, can be modeled by a View Directed Acyclic Graph (VDAG). The VDAG can be composed of one or more separable components. A VDAG represents the source relations and DW views as its nodes and their interdependencies as its edges. Source relations appear in a VDAG as leaves. All the internal nodes in a VDAG are DW materialized views. Let N be the set of all nodes in the VDAG and let E be the set of all edges.

A DW view V can be in one of three states [6]:

Fresh: The data in the view reflects precisely the data in the source databases; hence it's up to date.

Tolerated: The data in the view isn't up to date. But users accept that.

Stale: The data in the view must be refreshed to reflect the new data in the source databases.

After the initial loading, the view is in the fresh state. If D is violated due to source updates but F is not violated, then the view moves into the tolerated state. If both D and F are violated then the view moves into the stale state. When refreshment occurs according to D , the view moves back into the fresh state.

Sometimes, we have to refresh a view while it's still in the tolerated state i.e. before it moves to the stale state. The reason for that premature refreshment is clarified by the following scenario: let $\{V1, V2\} \subset N$ and $(V1 \rightarrow V2) \in E$ (i.e. $V1, V2$ are two views in the DW and $V1$ depends on $V2$). If $V1$ is found in the stale state then it must be refreshed. To refresh it, $V2$ must also be refreshed even if it is in the tolerated state. Restoring the view from the stale state to the tolerated state is not considered, because the overhead needed for that restoration is almost equal to the overhead needed to restore the view to the fresh state. Figure 1 shows the state transition diagram of a DW view.

Freshness constraints can be bounds on time interval during which the view is not fresh, bounds on the number of uninstalled updates in the view or

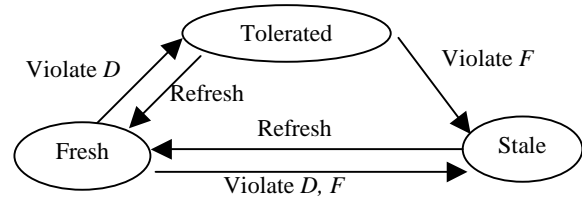


Figure 1: State transition diagram

bounds on values of some items in data sources or in other DW views.

The following facts are assumed in the model: any source site can contain any number of source relations. The communication network is not assumed to be FIFO. Transactions working on relations on the same source database are allowed while global transactions, those covering several source databases, are not.

3. THE DWMS OBJECTS AND MODULES

Figure 2 shows the objects, the modules and the interaction among them. Each object has an interface that is a set of public services used by the other objects to request some services from the object. The following subsections describe each object or module and the services it offers.

3.1. Configuration Object (CO)

The CO encapsulates the DW metadata and provides an interface for configuring the DW and for answering queries about the metadata.

3.2. The Main Integrator Module (MIM)

This is the main module of the DW integrator. Periodically, the MIM initiates a *view maintenance transaction* [1] during which the VDAG is traversed. When a view V is visited during this traversal, the view's freshness predicate F is checked. If the constraints are violated then a refreshment procedure of the view is issued to restore the view to the fresh state by applying the rules defined in the dependency predicate D of the view.

The VDAG nodes are visited in Depth First Traversal DFT to ensure that all the children of a node are visited before the node itself is visited. This is needed because checking and/or refreshing a view V

will make use of (1) The states of the children of V and (2) Updates, if any, that have taken place in those children. Hence, those states and updates must be determined before checking and/or refreshing V i.e. children of V must be visited before V is visited. The MIM can initiate the DFT at several root nodes of the VDAG; this introduces parallelism and speed-up in the view maintenance transaction.

3.3. View Maintenance Objects (VMOs)

Every view in the VDAG is associated with an object in the distributed object model. This object, called a View Maintenance Object (VMO) is responsible for storing information about the view; this information includes both the D and F predicates. The VMO is also responsible for checking the state of its view and refreshing it i.e. for maintaining the view, hence the name ‘View Maintenance Object’. All the VMOs are organized in a VDAG as every VMO stores a list of its children. The DFT initiated by the MIM actually traverses the VMOs; the traversal is initiated by the MIM and then recursively completed by the VMOs themselves. Note that the time period between successive DFT maintenance transactions should be guaranteed to be greater than the needed maintenance time of the whole DW.

The interface of a VMO consists of three ser-

vices: *VISIT*: To initiate the DFT at the VMO. *CHECK_STATE*: To check the state of the view. *REFRESH*: to restore the view to the fresh state. Both *CHECK_STATE* and *REFRESH* can do three types of queries (see figure 2): (1) *Query About Updates*: Request the needed information about updates from the Update Manager UM object. (2) *Source Query*: They also can query the source relations via the source wrappers. (3) *DW queries*: Issued against the DW views. For source queries, a compensation strategy is needed to overcome the problem of concurrent updates; this is described in section 4.

When a VMO is visited, during the DFT, the view’s state is checked using *CHECK_STATE* and if the state is *stale* the view is refreshed by *REFRESH*. If *REFRESH* of some VMO X detects that any VMO Y that is a child of X is tolerated or stale then it calls *REFRESH* of Y . This is necessary because refreshing a view implies refreshing all its descendants.

We note that the implementation of *VISIT* is the same for all VMOs while the other two services are implemented differently for the different VMOs. It is possible to automate the generation of the implementation part of these two services from a high-level declarative specification of D and F given by the DW administrator to the CO for every view

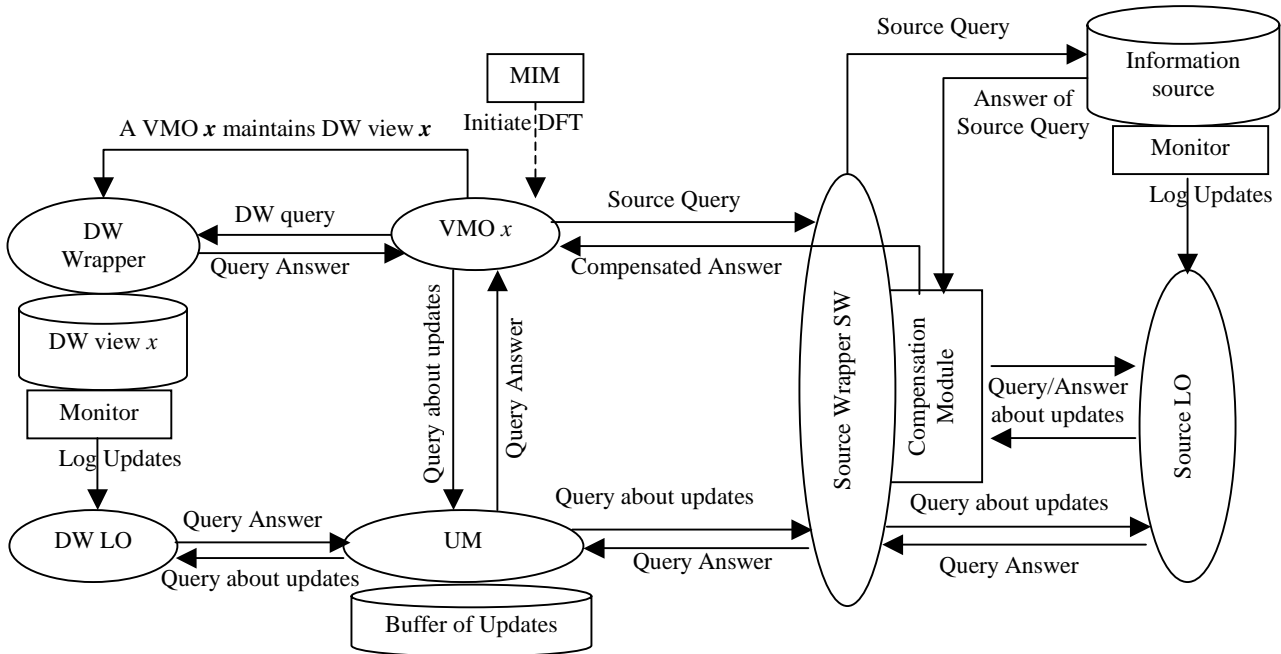


Figure 2: The architecture objects and modules

3.4. Monitors

The main task of a monitor is to keep an eye on the updates in a source or a data warehouse view and record these updates in a log object LO. The updates may be INSERT or DELETE. We assume that a MODIFY is logged as a DELETE followed by an INSERT. With each detected update, the monitor logs a time-stamp (update occurrence time) and the source or view at which the update has occurred. This information is used by the view maintenance strategy as will be described later. Source monitors detect and log source updates while the DW monitors detect and log the view updates that occur during the view maintenance transaction; these updates are needed by the VMOs of other views to propagate the updates according to the inter-view dependencies.

3.5. Log Objects (LO)

A LO is responsible for: (1) Storing the updates reported by monitors. (2) Answering queries about these updates. VMOs query LOs via the update manager UM as will be described. For example, a query about updates can be

```
SELECT count of updates
WHERE source="a specified source" AND
      timestamp BETWEEN "a specified
      range"
```

Another query can be a *Get_Updates* query:

```
SELECT all updates
WHERE source="a specified source" AND
      timestamp BETWEEN "a specified
      range"
```

When a LO processes a VMO query like the latter one, it returns the updates that satisfy the specified criteria and deletes the returned updates from its log. A Source LO stores the updates reported by the source monitor. The DW LO stores the updates reported by the DW monitors.

3.6. Update Manager (UM)

When a VMO needs any information about updates, it queries the UM which, in turn, queries the DW LO or the appropriate source LO via the source wrapper. When the UM gets the query answer, it

returns the answer to the requester VMO. If the query is a *Get_Updates* query then the LO will delete the returned updates from its log and the requester VMO will receive the updates from the UM and process the updates by installing them in its view. We note that there may be other VMOs that will need the same updates to maintain their views. Those other VMOs will request the updates later. This clarifies the need for an update manager that buffers the received updates and keeps track of what VMOs has taken what updates in order to correctly answer any query about updates from VMOs. A correct query answer is the answer that includes an update if and only if the update: (1) hasn't been delivered to the requesting VMO in a previous query. And (2) satisfies the query criteria. When the UM detects that a certain buffered update has already been delivered to all VMOs that need it, the UM deletes the update from the buffer of updates.

To accomplish that, a bitmap field is kept with every update. The bitmap has a bit for every VMO; each bit indicates whether the update should be delivered to the associated VMO in a later query about updates. The bitmap is updated after every query and when its bits are all zeros, the update is deleted from the buffer of updates.

3.7. Source Wrappers (SW)

A source wrapper translates source queries issued by VMOs to a query language that the source can 'understand'. The result of the translated query is then compensated by a compensation module (which can be integrated with SW) in order to remove the component of the result that is due to concurrent updates (see section 4). The compensated result is then returned to the requester VMO. The SW can also serve as a middleware between the UM and the source LO.

3.8. DW Wrapper

This object shields the other objects from particulars of the DBMS of the DW [5]. Hence, different DBMSs can be used without affecting the other objects; the implementation of the DW wrapper would differ while its interface is fixed.

4. VIEW MAINTENANCE

The view maintenance is initiated periodically by the MIM. VMOs request updates from the UM which, in turn, request them from the appropriate LOs. The VMOs use the received updates to incrementally maintain the views. Self-maintainability is not assumed i.e. the received updates may be insufficient for VMOs to maintain the views. VMOs may need to, further, query sources in order to complete the view maintenance. The answer of these source queries may contain components that are due to updates that haven't been sent to the DW yet; this introduces the problem of *concurrent updates* [3]. The problem is overcome by using a remote (at source sites) compensation operation that removes the effect of concurrent updates from the query answers.

The compensation module recognizes updates in the source LO as being *concurrent* if its time-stamp exceeds the time at which the DFT was initiated; this time value is passed from the MIM to VMOs on initiating the DFT and it's propagated down the VDAG and included with every source query or query about updates issued by any VMO. Hence, no answer of a source query or a query about updates will contain the effect of any update that occurred after the DFT had begun. Since the answer of any query about updates will not include updates that occurred after DFT initiation, these updates will not be removed from source LOs after answering any *Get_Updates* query (see subsection 3.5); this will enable the compensation module to remove the effect of these updates from the answers of source queries. To do that, the compensation module queries the source LO about those updates having time-stamps exceeding the DFT initiation time; the source LO answers this query, obviously, without removing the returned updates as they must be included in answers to VMOs' queries about updates in prospective maintenance transactions.

During the DFT maintenance transaction, updates can take place without interrupting the maintenance transaction at all (since updates are requested by VMOs rather than sent by sources); this relaxes the need of quiescence in sources for the completion of the view maintenance. According to our constraint-based per-view maintenance, the installation of updates in a view may be deferred even if the updates have been already received from the sources. This is controlled by the *F* predicate of the

view. This *controlled batching* of updates can enhance the maintenance performance. Since updates may be batched before installation in some views, the algorithm does not provide complete consistency; but strong consistency is guaranteed for each view, individually, since the successive states of any view correspond to successive states of sources and in the same order. Note that this view maintenance strategy allows two views to reflect source states at two different points of time and strong consistency is not guaranteed for all views together but only for each view individually.

5. CONCLUSIONS AND FUTURE WORK

This paper has proposed an efficient data warehousing framework based on a distributed object architecture and an incremental periodical view maintenance strategy that guarantees strong consistency for each view and doesn't require the sources to be quiescent. The efficiency is achieved by deferring the expensive view maintenance of some views whenever this deferring doesn't violate some freshness constraints defined by the users for every view.

It was shown that the UM keeps track of what updates has been processed by what VMOs, Future research work can employ this information to recover the system from crashes during view maintenance. Also, the topic of distributing VMOs on mobile machines can be studied in further research.

REFERENCES

- [1] A. Labrinidis and N. Roussopoulos. "Reduction of materialized view staleness using online updates". TR3878, DCS, University of Maryland at College Park, Feb. 1998.
- [2] Asuman Dogac, Cevdet Dengi and M. Tamer Özsu. "Distributed Object Computing Platforms". Communications of the ACM. September 1998/ Vol. 41 No.9
- [3] Divyakant Agrawal, Amr El Abbadi, Ambuj K. Singh and Tolga Yurek. "Efficient View Maintenance at Data Warehouses". SIGMOD 1997.
- [4] J. Widom. "Research Problems in Data Warehousing". CIKM 1995.

- [5] J. L. Wiener, H. Gupta, W. J. Labio, Y. Zhuge, H. Garcia-Molina, J. Widom. "A System Prototype for Warehouse View Maintenance." In *Proceedings of the ACM Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, June 7, 1996.
- [6] Yousry Taha, Abdelsalam Helal and Khalil M. Ahmed. "A Stochastic Consistency Model for Data Warehousing". In *Proceedings of the AIS*, Indianapolis, USA 1997.