Efficient distributed object framework for data warehousing

Noha Adly, Yousry Taha and Arsany S. Sawiros
Computer Science & Automatic Control Dept., Alexandria University, Alexandria, Egypt

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.

هذا البحث يقدم إطار لعمل مستودعات البيانات. هذا الإطار مصمم بنظام الكيانات الموزعة، ولهذا فهو يتمتع بمزايا عديدة كالقابلية للاتساع وسهولة العمل المشترك بين الكيانات المختلفة إلى جانب القدرة على الربط بين البيئات ومصادر البيانات غير المتجانسة. الإطار المعروض يتبنى طريقة بسيطة تراكمية وذات كفاءة عالية لتحديث المستودعات. هذه الطريقة تستند إلى الحقيقة القائلة بأن الأجزاء المختلفة لمستودع البيانات قد تكون لها شروط حداثة تختلف من جزء لآخر. هذه الحقيقة يمكن استخدامها لتحسين أداء عملية التحديث خاصة في المستودعات الضخمة، وذلك بتركيب التعديلات التي تم استقبالها من مصادر البيانات فقط عندما يكون ذلك ضروريا وفقا لشروط الحداثة ويتم التركيب فقط للأجزاء التي لها هذه الشروط التي تحتم التحديث مع تأجيل التركيب في باقي الأجزاء. طريقة التحديث المعروضة تضمن درجة التجانس القوى لكل جزء من المستودع على حدة، ولا تتطلب الوصول لحالة سكون في تعديلات مصادر البيانات من أجل إتمام عملية التحديث.

Keywords: Data warehousing, View maintenance, Freshness constraints, Distributed objects, View directed acyclic graph

1. Introduction

Data Warehousing (DW) is used for reducing the load of on-line transactional systems by extracting and storing the data needed for analytical purposes [1] such as On-Line Analytical Processing (OLAP) and Data Mining (DM) which have become essential for Decision Support Systems (DSS).

As suggested in [2], The general architecture of a Data Warehouse Management System DWMS consists of several components, which are distributed among several sites. The central site hosts the DW as well as the component responsible for data integration called the *DW Integrator.* At the source sites, there are two main components, a *source monitor* to detect and report all source updates and a *wrapper* that is in contact with the DW Integrator via a communication network; the task of the wrapper is to provide a query interface to the integrator. The integrator sends queries to the data source and the associated wrapper translates the query to the source query language to extract the information and send it back to the integrator after, probably, performing some transformations on the data.

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.

The *View Maintenance* problem has several dimensions and alternatives that have been investigated in literature such as [3]. 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 installing 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 updates buffer; so, a single update can be installed in some views and left in the update buffer for later installation in other views. When an update *U* will be installed in a view *V*? depends on the freshness constraints defined on *V*. We call this approach *per-view maintenance*. This constraint-based deferred maintenance saves a large maintenance overhead while in the same time obeys the freshness requirements of users.

Defining different freshness constraints for different views realizes the concept of unequally 'important' views suggested in [4].

In addition to the major problems of view maintenance, data warehousing have several other problems such as monitoring legacy systems, data transformation and integration and the huge amounts of data stored. This paper considers only the architecture and the view maintenance. The other topics are deeply discussed in other papers and the approaches are applicable to our framework.

The DWMS architecture proposed in this paper is based on distributed objects such as the CORBA standard [5,6]. The employment of distributed object technology with the proposed architecture offers several benefits [7,8] such as:

1- *Plug-and-Play modularity:* Modules or objects can be easily replaced or inserted without the need for any modifications in the system.

2- *Scalability:* The system can scale gracefully by distributing the maintenance work among more objects and more machines.

3- *Heterogeneous sources:* The DW sources can have heterogeneous platforms. Wrapper objects of different sources will have the same interface and different implementations according to the sources.

4- *Communication heterogeneity:* the communication heterogeneity is solved by the location and protocol transparency provided by the Object Request Broker ORB as a client can request and get a service from an object without knowing the object's location or communication protocol.

5- *Interoperability:* the different system components can easily inter-operate to perform the required tasks.

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 database. Each view is defined in terms of driver objects; the latter could be source relations as well as other data warehouse views. For each DW view, let *V* = <*D*, *F*> be the view's descriptor where:

D: *Dependency predicate,* describes the relationship between the DW view and its drivers. 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 to fulfil the DW user's requirements.

The dependencies among the different objects, including both source relations and DW views, can be modelled by a View directed Acyclic Graph (VDAG). The VDAG can be composed of one or more separable components. A VDAG represents the information sources and DW views as its nodes and their dependencies as its arcs. Source relations appear in a VDAG as leaves. All the internal nodes in a VDAG are DW materialized views. We refer to all the descendants of a view *V* as the *drivers* of *V*. Let *N* be the set of all nodes in the VDAG and let *E* be the set of all edges.

As mentioned above, A DW view can have freshness constraints defined by the user. Hence, the view can be in a state that is not up-to-date, yet it is accepted by the user; we call this the *tolerated state* [9]. According to that model, A DW view *V* can be in one of three states:

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

2. *Tolerated:* The data in the view isn't up to date but the tolerance is accepted by the users.

3. *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. Hence, whenever a restoration occurs the view is updated according to the most recent source state i.e. the view moves to the fresh state. Fig. 1 shows the state transition diagram of a DW view

The freshness constraints defined on a DW view *V* can be classified into three main classes:

1. *Time constraints:* Consider the maximum allowed time period between successive refreshments to *V*. Or the maximum allowed time period between a refresh of *V* and an uninstalled update in the drivers of *V*.
2. *Version constraints:* Consider the maximum allowed number of uninstalled updates received from the drivers of *V*.
3. *Value constraints*: Consider some stored or calculated value from the sources or the DW.

Obviously, it is assumed that checking the state of a view is cheaper (in terms of time, space, communication, processing, etc) than restoring the view to the fresh state.

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.

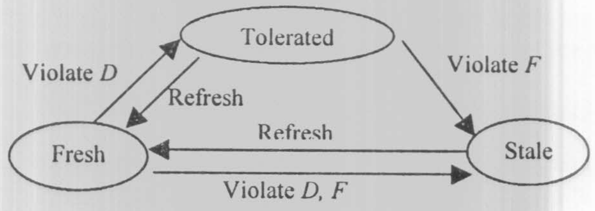

Fig. 1. State transition diagram.

3. The DWMS objects and modules

This section describes the proposed DWMS architecture where the different tasks are distributed among several objects and modules at several sites.

Fig. 2 shows the objects and the modules and the interaction among them. Each object has an interface that is a set of public services. This interface is used by the other objects to request the services that the object is responsible for. This object interaction gives the architecture flexibility and ease of maintainability since the implementation of any object can be changed without affecting the others as long as the interface is fixed. In the following, the different objects and modules will be briefly discussed.

3.1. Configuration object

This object provides an interface that enables configuring the DW environment. It also encapsulates the DW metadata that describe the whole environment. Configuring the (DW) environment includes adding, deleting, and modifying DW views. Modifying a view can affect the *D* and/or the *F* predicates of it.

The metadata that is encapsulated in the Configuration Object (CO) includes information about the DW schema and source schemas. The CO has interface to answer any query about this metadata.

This encapsulation provides flexibility in the Metadata-Standard used. It also simplifies and unifies the process of querying metadata by the different objects.

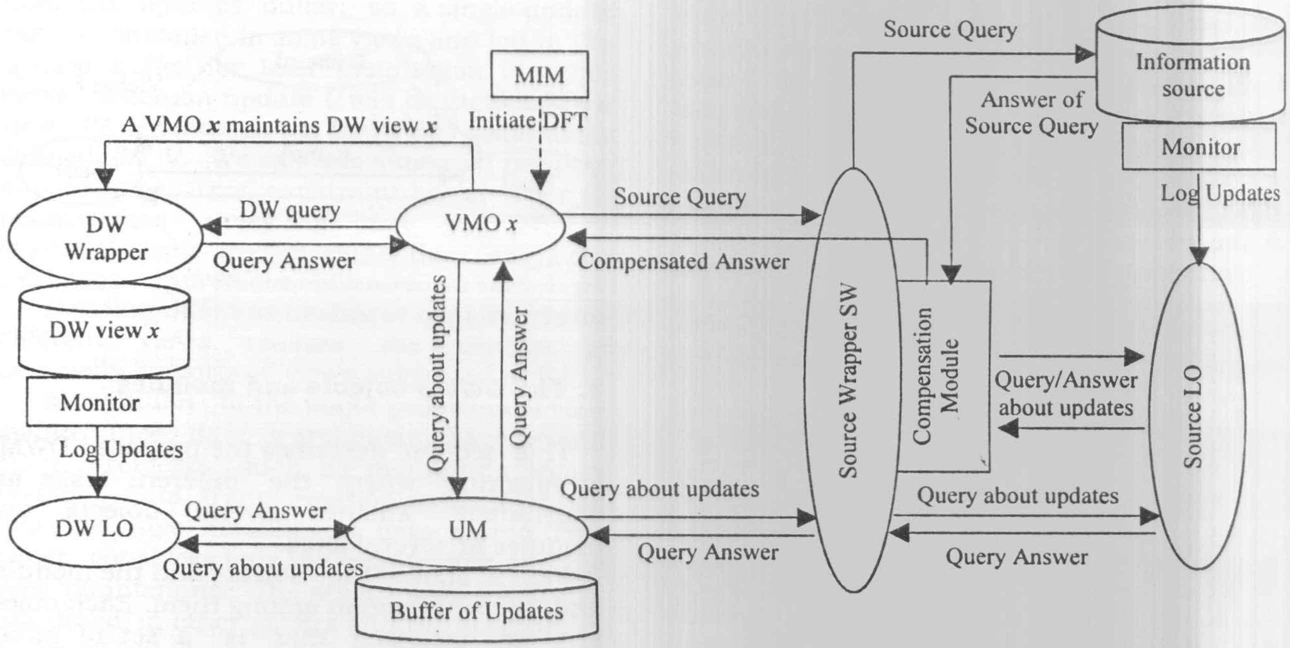

Fig. 2. The architecture objects and modules.

3.2. The main integrator module

This is the main module of the DW integrator. Periodically, the Main Integrator Module (MIM) initiates a *view maintenance transaction* [4] during which all 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 initiates the VDAG traversal at the root node(s) of a VDAG. If a VDAG has more than one root then the traversal initiation can be done asynchronously at the different roots i.e. after initiating the traversal at a root, the MIM doesn't have to wait for the DFT return before initiating it at another root. This will produce parallelism in the view maintenance processing of the VDAG and hence the maintenance time is reduced. We note that if two traversal paths intersect at a node (as a result of the parallelism) then one of them must stall waiting the DFT return of the other or it can go in another valid path according to the DFT rules.

3.3. View maintenance objects

Every view in the VDAG has an associated 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 services: *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 fig. 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 DW view.

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 ADD or DELETE. We assume that a MODIFY is logged as a DELETE followed by an ADD. 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.

The need for source monitors is obvious. The task of the DW monitors is to detect and log the view updates that occur during the view maintenance operation. These updates are needed by the VMOs of other views to propagate the updates according to the inter-view dependencies.

The source monitors may be simple if the sources are able to report any updates in them or they can be complex if the sources are legacy sources. On the other hand, the DW monitors will use the capabilities of the DBMS on which the DW is built.

3.5. Log objects

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

SELECT *count of updates*
WHERE *source="a specified source" AND timestamp BETWEEN "a specified range"*
Another query can be:
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

When a VMO needs any information about updates, it queries the Update Manager (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 answer is a set of updates 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, appropriately, 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. The time difference between requests of the same updates can be due to the different positions of requester VMOs in the VDAG (the DFT dictates that lower level VMOs will request before the higher level ones). The time difference can also be due to difference in the *F* predicates of the different VMOs; two VMOs sharing a common driver may need to install updates received from that driver at different times according to their *F* predicates.

The above scenario 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 update buffer.

How can the UM know what VMOs will request what updates? And how can it keep track of what VMOs has already taken what updates? This can be accomplished by keeping a bitmap field with every update. The bitmap has a bit for every VMO. When an update *U* is received from a LO, it's buffered and its bitmap is initialized by (1's) in every bit that corresponds to a VMO that has the source of *U* as one of its children in the VDAG. When *U* is delivered to some VMO, the corresponding bit in the bitmap of *U* is reset to (0). When the bitmap is all zero, *U* is deleted from the update buffer.

3.7. Source wrappers

The main task of a Source Wrapper (SW) is to translate source queries issued by VMOs against the source relations to a query language that the source can 'understand'. The translated query is then processed by the source and the result is returned to the SW. This result is then compensated by a compensation module, can be integrated with SW, in order to remove the component of the result that is due to concurrent updates (this will be discussed in section 4). Some transformations may be applied to the compensated result. The compensated result is then returned to the requester VMO. The SW can do another task; it can serve as a middleware between the UM and the source LO; this enables accomplishing some transformations in answers of queries about source updates.

3.8. DW wrapper

The task of this module is to shield the other modules of the system from particulars of the DBMS of the DW [8]. Hence, different DBMSs can be used without affecting the other modules; 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* [1]. 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 the concurrent updates in the source LO from their associated time-stamps. An update is considered concurrent if its time-stamp exceeds the time at which the DFT of view maintenance 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 begins. 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 timestamps 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.

This algorithm simulates the following: once the DFT starts, a snapshot of all the sources is taken and VMOs will 'see' only this snapshot excluding any updates that take place after this snapshot. During the DFT maintenance transaction, updates can take place without interrupting the maintenance operation 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.

Note that the time of DFT start can be broadcast to all source wrappers rather than being included with each query. But, this would require the assumption that the communication network delivers messages from a certain source to a certain destination in a FIFO manner (the time value must be guaranteed to reach every source before the first query to that source reaches the source). By including the DFT start time value in every query, the FIFO assumption is relaxed since no other part of the algorithm requires it.

According to our per-view maintenance, the installation of updates in a view may be deferred even if the updates are received from the sources. As described above, 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, 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 scheme allows two views to reflect source states at two different points of time and guarantee s strong consistency for each view individually but not for all views together. This is illustrated in the following scenario: An update *U1* is installed in view *V1* and not installed in view *V2*; then a new later update *U2* arrives and it is also installed in *V1* but not in *V2*. After that, *U1* is installed in *V2*; hence this later installation of *U1* took place after the installation of *U2* in *V1* although *U1* took place before *U2*; this violates the condition of global strong consistency but keeps individual strong consistency for each view.

5. Conclusions and future work

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

A prototype that realizes some of the ideas proposed in this paper has been implemented using OOP [10]. It's intended to extend that prototype by moving it into a CORBA environment and implementing the proposed view maintenance algorithm (the prototype has assumed self maintainability) and measuring the performance of maintaining the DW.

It was shown that the UM keeps track of what updates has been processed by what

VMOs. This information can be used to recover the system from crashes during view maintenance; the topic of recoverability needs more attention especially when we study the effect of possible network failures on the view maintenance strategy.

Also, the topic of distributing VMOs on mobile machines is under consideration.

References

[1] D. Agrawal, A. El Abbadi, K. Singh and T. Yurek. "Efficient View Maintenance at Data Warehouses", SIGMOD (1997).

[2] J. Widom. "Research Problems in Data Warehousing", CIKM (1995).

[3] A. Gupta and S. Mumick. "Maintenance of Materialized views Problems, Techniques and Applications". IEEE Data Engineering Bulletin (1995).

[4] A. Labrinidis and N. Roussopoulos. "Reduction of materialized view staleness using online updates". TR3878, DCS, University. of Maryland at College Park, Feb. 1998. [5] Object Management Group, "The Common Object Request Broker: Architecture and Specifications", OMG Document Number 91.12.1, December (1991).

[6] A. Dogac, C. Dengi and M. T. Öszu. "Distributed Object Computing Platforms", Communications of the ACM, Vol. 41 (9), (1998).

[7] E. Kilic, G. Ozhan, C. Dengi, N. Kesim, P. Koksal and A. Dogac, "Experiences in Using CORBA for Multidatabase Implementation", in Proceedings of the 6th International Conference in database and Expert Systems Applications, London, September (1995).

[8] J. L. Wiener, H. Gupta, W. J. Labio, Y. Zhuge, H. Garcia-Molina and 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).

[9] Y. Taha, A. Helal and K. Ahmed. "A Stochastic Consistency Model for Data Warehousing", In Proceedings of the AIS, Indianapolis, USA (1997).

[10] A. Sawiros, H. Khalil, H. Diaa El-deen, M. Alsebaeyie, R. Saleh and Y. Bassily. "A prototype for a Data Warehouse Management System", B.Sc. graduation project, Department of Computer Science and Automatic Control, Faculty of Engineering, Alexandria University, July (2000).

Received July 30, 2001
Accepted October 30, 2001