

Incremental Maintenance of Object-Oriented Views in a Warehousing Environment

Ching-Ming Chao

*Department of Computer and Information Science
Soochow University
Taipei, Taiwan 111, R.O.C.
E-mail: chao@cis.scu.edu.tw*

Abstract

Data warehousing is an approach to data integration in which integrated information is stored in a data warehouse for direct querying and analysis. To provide fast access, a data warehouse stores materialized views defined over data from its data sources. As a result, a data warehouse needs to be maintained to keep its contents consistent with the contents of its data sources. Incremental maintenance is generally regarded as a more efficient way to maintain materialized views in a data warehouse; however, it is also technically more difficult. Most of the previous work on incremental maintenance of materialized views is confined to the relational database. Unfortunately, existing approaches to incremental maintenance of relational views cannot be directly applied to object-oriented views. Although incremental maintenance of object-oriented views has been investigated in the past few years, the proposed view maintenance algorithms only work in a centralized environment but not in a data-warehousing environment. In this paper, we study the problem of incremental maintenance of object-oriented views in a data-warehousing environment with multiple distributed data sources. In particular, we address two primary issues. First, we identify which kinds of updates to which classes in the data source may cause changes to an object-oriented view. Second, we propose an incremental view maintenance algorithm that is capable of maintaining an object-oriented view defined over multiple distributed data sources. A prototype system has been implemented and the result of our preliminary performance evaluation demonstrates that our incremental view maintenance algorithm is efficient.

Key Words: Data Warehousing, Object-Oriented Data Warehouses, Materialized Views, Incremental View Maintenance

1. Introduction

Data warehousing is an approach to data integration in which integrated information is stored in a data warehouse for direct querying and analysis by the user [5]. The key idea of the data warehousing approach that differs from traditional data integration approaches is that relevant data is extracted from data sources and integrated information is stored in the data warehouse in advance of

querying and analysis. With the data warehousing approach, queries can be answered and data analysis can be performed more quickly and efficiently since integrated information is directly available at the data warehouse.

A *data warehouse* is a repository of integrated information, which usually integrates data from multiple distributed, autonomous, and possibly heterogeneous data sources. In order to provide

the user with fast access capability, a data warehouse stores *materialized views* defined over data from its data sources. As the data of its data sources is updated, a data warehouse may need to be maintained in order to keep its contents consistent with the contents of its data sources. Because a data warehouse effectively stores integrated information as materialized views, maintaining a data warehouse can be regarded as maintaining the materialized views in the data warehouse.

To maintain a materialized view, there is a choice between recomputing the view from scratch and maintaining the view incrementally. Recomputing a view from scratch is technically simpler, but is more time and resource consuming. To maintain a view incrementally, one has to compute the change to the view on the basis of the update to its source data and then apply the computed change to the view. *Incremental view maintenance* is generally less expensive when the size of updates to the source data is small compared to the size of the source data. However, it is technically more difficult.

Much research has been done on the problem of incremental maintenance of materialized views. However, most of the previous work is confined to the relational database, e.g., [1,4,6,8,10,11,12]. Recently, the concept of an *object-oriented data warehouse* has been proposed as a better means to data integration [7,9]. An object-oriented data warehouse is a data warehouse in which integrated information is stored as *object-oriented materialized views*. Because the object-oriented database has many unique features that are absent from the relational database, such as object identity, complex attributes, inter-object reference, class inheritance, etc., existing approaches to incremental maintenance of relational views cannot be directly used to maintain object-oriented views. Although in the past few years incremental maintenance of object-oriented views has been investigated, e.g., [2,3]. However, the proposed view maintenance algorithms only work in a centralized environment but not in a data-warehousing environment in which the data sources are distributed and separated from the data warehouse.

In this paper, we study the problem of incremental maintenance of object-oriented views in a data-warehousing environment with multiple distributed data sources. In particular, we address two primary issues. The first issue is to identify during view compilation time which kinds of updates to which classes in the data source may cause

changes to an object-oriented view. Such updates are called the *potential updates* to the view. We identify six categories of potential updates to an object-oriented view. The second issue is to incrementally maintain an object-oriented view in response to its potential updates. We propose an incremental view maintenance algorithm that is capable of maintaining an object-oriented view defined over multiple distributed data sources. We have implemented a prototype system for incremental maintenance of object-oriented views in a data-warehousing environment. A preliminary performance evaluation has been conducted, which demonstrates that our incremental view maintenance algorithm is efficient because it outperforms recomputation in the majority of cases.

The remainder of this paper is organized as follows. In Section 2 we describe some background information and assumptions that will be used for the rest of this paper. In Section 3 we identify the potential updates to an object-oriented view. In Section 4 we present the incremental view maintenance algorithm. In Section 5 we illustrate some of the results of our preliminary performance evaluation. Section 6 concludes this paper and suggests some directions for future research.

2. Background and Assumption

Figure 1 depicts the architecture adopted by this paper for incremental maintenance of object-oriented data warehouses. The data warehouse integrates data from multiple distributed, autonomous, and possibly heterogeneous data sources. Each data source has a wrapper associated with it. The *wrapper* is responsible for detecting updates occurred at the data source and notifying the integrator those updates, handling queries from the integrator, and sending answers of queries to the integrator. The data warehouse has an integrator associated with it. The *integrator* is responsible for computing the change to a view caused by updates to data sources and applying the computed change to the view. As receiving updates from data sources, the integrator may issue necessary queries to the wrappers of relevant data sources to compute changes to affected views.

In a distributed environment, communication among systems is through sending messages. During the process of warehouse maintenance, there are primarily three types of messages between the integrator and wrappers: a wrapper sends an update message to the integrator, the integrator sends a

query message to a wrapper, and a wrapper sends an answer message to the integrator. These messages are sent through a communications network. We assume that the communications network is reliable and that messages sent from the same source to the same destination are delivered in the same order as they were sent. However, we place no restrictions on the order in which messages sent from different wrappers to the integrator are delivered.

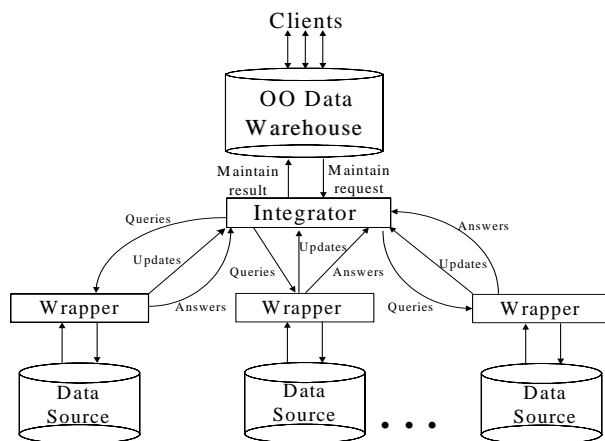


Figure 1. An architecture for object-oriented data warehouse maintenance

We assume that materialized views stored in the object-oriented data warehouse are defined on object-oriented classes. In the case that some data sources are not object-oriented databases, their wrappers are responsible for performing the necessary transformation. Our approach to incremental maintenance of object-oriented data warehouses is general and is not restricted to any specific object data model. Therefore, we adopt a generic object model and language to describe the ideas and examples. A simplified university data warehouse will be used as the running example throughout this paper. This data warehouse integrates data from two data sources, source X and source Y. Figure 2 shows the definition of base classes of the data warehouse in which the classes Staff and Department are stored in source Y and the other classes are stored in source X.

We study incremental maintenance of object-oriented views that are defined on one or more classes and whose definition allows path expressions to appear in the SELECT and WHERE clauses. Figure 3 shows the definition of two materialized views V1 and V2 in the data warehouse.

3. Potential Updates

Identifying the potential updates to a view is necessary for view maintenance, because one has to know which updates to which classes may cause changes to a view to be able to maintain the view. To improve maintenance efficiency, we identify the potential updates to a view during view compilation time instead of run time.

```

class Person
{Name: string, Age: integer, Children: set (Person)};
class Student inherits Person
{Major: Department, Year: integer, Courses: set (Course)};
class Staff inherits Person
{Dept: Department, Salary: integer};
class Graduate inherits Student
{Advisor: Staff, Thesis: string};
class Course
{Name: string, Code: string, Credit: integer};
class Department
{Name: string, Head: Staff};

```

Figure 2. Definition of base classes

```

view V1 (SN: string, CN: set (string), HN: string, HA: integer)
select Student.Name, Student.Courses.Name, Student.Major.Head.Name, Student.Major.Head.Age
from Student
where Student.Year = 4
and "BCC" in Student.Courses.Name ;
view V2 (SN: string, FN: string)
select Student.Name, Staff.Name
from Student, Staff
where Student.Major = Staff.Dept ;

```

Figure 3. Definition of materialized views

For the purpose of identifying potential updates, we distinguish four roles that a class may play for a view. A class is a *defining class* of a view if the class appears in the FROM clause of the view definition. For example, Student is the only defining class of the view V1. A class is a *referenced class* of a view if the class is referenced from a defining class within some path expression of the view definition. For example, Course, Department, and Staff are the referenced classes of V1. A class is an *inheriting class* of a view if the class directly or indirectly inherits a defining or referenced class of the view. For example, Graduate is an inheriting class of V1. Note that a class may play more than one of the three roles mentioned above for a view. A class is an *irrelevant class* of a view if the class does not play any of the three previous roles for the view. For example, Person is an irrelevant class of V1.

Updates to an irrelevant class of a view cannot

cause any change to the view. To demonstrate this argument, we enumerate various situations in which a class is regarded as an irrelevant class of a view. First, a class that is inherited by a defining or referenced class of a view is an irrelevant class of the view. For example, Person is inherited by Student (a defining class) and Staff (a referenced class), and therefore is an irrelevant class of V1. Updates to objects of Person that are not also objects of any of its subclasses cannot cause any change to V1. Second, a class that is referenced by a defining class in the class composition hierarchy but not within any path expression of a view is an irrelevant class of the view. For example, Person is referenced by Student in the class composition hierarchy but not within any path expression of V1 and is therefore an irrelevant class of V1. Finally, a class that is not related in any way to a view is an irrelevant class of the view. Updates to such kind of classes obviously cannot cause any change to the view.

Updates to a defining, referenced, or inheriting class of a view may cause changes to the view, but only for certain kinds of updates. Again for the purpose of identifying potential updates, we distinguish five kinds of updates to a class: insertion, deletion, modification of SELECT attributes (i.e., attributes that appear only in the SELECT clause of the view), modification of WHERE attributes (i.e., attributes that appear in the WHERE clause of the view), and modification of other attributes (i.e., attributes that do not appear in the definition of the view). We will discuss the effects of these five kinds of updates to those three kinds of classes on a view in turn.

First, we discuss the effects of updates to a defining class on a view. Inserting an object to a defining class will cause insertion of one or more objects to a view if the WHERE condition evaluates to true on the inserted object. For example, inserting an object to Student will cause insertion of an object to V1 if the inserted Student object makes the WHERE condition of V1 evaluate to true. Deleting an object from a defining class will cause all objects derived from the deleted object, if any, to be deleted from a view. For example, deleting a Student object will cause all objects derived from the deleted Student object, if any, to be deleted from V2. Modifying a SELECT attribute of an object of a defining class will cause one or more attributes of all objects of a view that are derived from the modified object to be modified. For example, modifying the attribute Major of a Student object will cause the attributes HN and HA of the V1 object derived from the modified Student object to be modified. Modifying a WHERE attribute of

an object of a defining class may cause insertion to, deletion from, or modification of a view. For example, changing the value of the attribute Year of a Student object from 3 to 4 may cause an object to be inserted to V1. Modifying other attributes of a defining class cannot cause any change to a view.

Then we discuss the effects of updates to a referenced class on a view. Inserting an object to a referenced class does not by itself cause any change to a view. However, it may cause modification of the defining class of the referenced class, which may in turn cause changes to a view. The same applies to deleting an object from a referenced class. Therefore, we do not consider the insertion to and deletion from a referenced class as the potential updates to a view. Modifying a SELECT attribute of an object of a referenced class will cause one or more attributes of all objects of a view that are derived from the objects of the defining class that reference the modified object to be modified. For example, modifying the attribute Head of a Department object will cause the attributes HN and HA of V1 objects derived from Student objects that reference the modified Department object to be modified. Modifying a WHERE attribute of an object of a referenced class may cause insertion to, deletion from, or modification of a view. For example, changing the value of the attribute Name of a Course object from "BCC" to "IIT" may cause objects to be deleted from V1. Modifying other attributes of a referenced class cannot cause any change to a view. Finally, we discuss the effects of updates to an inheriting class on a view. The effect of updating an inheriting class on a view is the same as that of updating the defining or referenced class that is inherited by the inheriting class, because an object of an inheriting class is also an object of the inherited class. For example, inserting an object to the class Graduate produces the same result to V1 as inserting an object to the class Student.

In summary, we identify the following six categories of potential updates to an object-oriented view.

1. **Ins**: Insertion to a defining class or an inheriting class that inherits a defining class
2. **Del**: Deletion from a defining class or an inheriting class that inherits a defining class
3. **MDS**: Modification of SELECT attributes of a defining class or an inheriting class that inherits a defining class
4. **MDW**: Modification of WHERE attributes of a defining class or an inheriting class that inherits a defining class
5. **MRS**: Modification of SELECT attributes of a

referenced class or an inheriting class that inherits a referenced class

6. **MRW**: Modification of WHERE attributes of a reference class or an inheriting class that inherits a referenced class

For example, the six categories of potential updates to the view V1 are listed below.

1. **Ins** Student **Ins** Graduate
2. **Del** Student **Del** Graduate
3. **MDS** Student.Name **MDS** Student.Major
MDS Graduate.Name **MDS** Graduate.Major
4. **MDW** Student.Year **MDW** Student.Courses
MDW Graduate.Year **MDW** Graduate.Courses
5. **MRS** Department.Head
MRS Staff.Name **MRS** Staff.Age
6. **MRW** Course.Name

4. View Maintenance Algorithm

In this section, we present an algorithm that can incrementally maintain an object-oriented view defined over multiple distributed data sources. Our algorithm adopts the *immediate update* mode, in which a view is maintained immediately after each update to its source data. With regard to the correctness of a view maintenance algorithm, [12] defined four levels of consistency for warehouse views. Our algorithm achieves *strong consistency*, in which each warehouse state reflects a set of valid source states and the order of the warehouse states matches the order of the source actions.

Our algorithm has several salient features that can improve maintenance efficiency. First, only the potential updates to a view can cause the algorithm to compute the change to the view. This avoids a waste of resources on updates that cannot affect any view. Second, a source modification that only causes modification of a view rather than insertion to or deletion from a view is treated directly. In contrast, most of the previous algorithms treat such a modification as a deletion followed by an insertion. Finally, our algorithm uses two *auxiliary views* to assist in the maintenance of a warehouse view, which can avoid access to source data as much as possible.

As a materialized view V is defined in the data warehouse, two auxiliary views AV1_for_V and AV2_for_V are also defined there. AV1_for_V is materialized but AV2_for_V is not materialized. AV1_for_V has attributes for the OID of objects in the defining classes of V that derive an object of V as well as an attribute for the OID of that object of V. AV2_for_V is almost identical to V except that it in-

cludes additional attributes for the OID of objects that derive an object of V, if V does not have those attributes. The derivation of the definition of auxiliary views from the definition of a given view is a simple syntactic mapping. The definitions of auxiliary views for V1 and V2 are shown in Figure 4 and Figure 5, respectively, in which the generated strings in the definition of auxiliary views are shown in italics. AV1_for_V is used to find the objects of V and AV1_for_V that are to be deleted or the objects of V that are to be modified without access to source data. AV2_for_V is used to compute the objects to be inserted to V and AV1_for_V.

```

view AV1_for_V1 (SO: Student, VO: V1)
select Student, V1
from Student, V1
where Student.Year = 4
and "BCC" in Student.Courses.Name
and V1.SN = Student.Name
and V1.CN = Student.Courses.Name
and V1.HN = Student.Major.Head.Name
and V1.HA = Student.Major.Head.Age ;
view AV2_for_V1 (SO: Student, SN: string,
CN: set (string), HN: string, HA: integer)
select Student, Student.Name,
Student.Courses.Name,
Student.Major.Head.Name,
Student.Major.Head.Age
from Student
where Student.Year = 4
and "BCC" in Student.Courses.Name ;

```

Figure 4. Definition of auxiliary views for V1

```

view AV1_for_V2 (SO: Student, FO: Staff, VO: V2)
select Student, Staff, V2
from Student, Staff, V2
where Student.Major = Staff.Dept
and V2.SN = Student.Name
and V2.FN = Staff.Name ;
view AV2_for_V2 (SO: Student, FO: Staff,
SN: string, FN: string)
select Student, Staff, Student.Name, Staff.Name
from Student, Staff
where Student.Major = Staff.Dept ;

```

Figure 5. Definition of auxiliary views for V2

Algorithm Incremental View Maintenance

At each data source:

- Upon detection of an update U, send U to the warehouse.
- Upon receipt of a query Q, evaluate Q in terms of the current source state and send the result to the warehouse.

At the data warehouse:

- Initialize WV and WAV1 with the current state of V and AV1, respectively.

- Upon receipt of an update U_i that is a potential update to V :
- ① If U_i is insert ($C_i, \Delta C_i$)
 - Set $PU(U_i) = \emptyset$;
 - *Source_evaluate* ($AV2[\Delta C_i]$);
 - Upon completion of evaluating ΔV and $\Delta AV1$, $\forall U_j \in PU(U_i)$,
 - ❶ If U_j is delete ($C_j, \nabla C_j$)
 - ⇒ Find $\nabla \Delta V$ and $\nabla \Delta AV1$ by searching ∇C_j in $\Delta AV1$;
 - ⇒ Execute delete ($\Delta V, \nabla \Delta V$) and delete ($\Delta AV1, \nabla \Delta AV1$);
 - ❷ If U_j is modify ($C_j, \diamond C_j, MA(C_j)$)
 - ⇒ Find $\diamond \Delta V$ by searching $\diamond C_j$ in $\Delta AV1$;
 - ⇒ Determine the affected attributes in ΔV , $AA(\Delta V)$, from $MA(C_j)$;
 - ⇒ Compute the new value of $AA(\Delta V)$ possibly by querying data sources;
 - ⇒ Execute modify ($\Delta V, \diamond \Delta V, AA(\Delta V)$);
 - Execute insert ($WV, \Delta V$) and insert ($WAV1, \Delta AV1$) without inserting duplicate objects;
 - ② If U_i is delete ($C_i, \nabla C_i$)
 - $\forall U_k \in UUS$ and U_k is an insertion, add U_i to $PU(U_k)$;
 - Find ∇V and $\nabla AV1$ by searching ∇C_i in $WAV1$;
 - Execute delete ($WV, \nabla V$) and delete ($WAV1, \nabla AV1$);
 - ③ If U_i is modify ($C_i, \diamond C_i, MA(C_i)$)
 - $\forall U_k \in UUS$ and U_k is an insertion, add U_i to $PU(U_k)$;
 - Find $\diamond V$ by searching $\diamond C_i$ in $WAV1$;
 - Determine the affected attributes in V , $AA(V)$, from $MA(C_i)$;
 - Compute the new value of $AA(V)$ possibly by querying data sources;
 - Execute modify ($WV, \diamond V, AA(V)$);
- When $UUS = \emptyset$, replace V and $AV1$ with WV and $WAV1$, respectively.

Figure 6. View maintenance algorithm (partial)

To simplify the presentation, we describe a partial version of our algorithm that handles the first three categories of potential updates. Actually, the complete algorithm can handle all six categories of potential updates. Expanding the partial algorithm to the complete algorithm is straightforward. The fourth category of potential updates is handled by treating it as a deletion followed by an insertion. The fifth and sixth categories of potential updates are handled similarly to the third and fourth categories of potential updates, respectively, with some extension. Figure 6 shows the partial algorithm.

First, we define the notations used in the algorithm. V is the view to be maintained. $AV1$ and $AV2$ are the auxiliary views of V . WV and $WAV1$

are the working copies of V and $AV1$, respectively. If C denotes a base class, the notations ΔC , ∇C , and $\diamond C$ denote the objects inserted to C , deleted from C , and modified in C , respectively. If C denotes a materialized view, the notations ΔC , ∇C , and $\diamond C$ denote the objects to be inserted to C , deleted from C , and modified in C , respectively. Let C be a base class or a materialized view. The notation insert ($C, \Delta C$) denotes an update that inserts objects ΔC to C . The notation delete ($C, \nabla C$) denotes an update that deletes objects ∇C from C . The notation modify ($C, \diamond C, A$) denotes an update that modifies attributes A of objects $\diamond C$ in C .

UUS is the *unfinished update set*, which contains source updates that have been received at the data warehouse but their effects on V have not been incorporated into WV . Let U be an insertion, $PU(U)$ denotes the set of *pending updates* of U , which contains source updates that are received at the data warehouse while U is still in UUS . $PU(U)$ contains only deletions and modifications, but not insertions. $AV2[\Delta C_i]$ denotes the view defining expression of $AV2$ in which C_i is replaced with ΔC_i . *Source_evaluate* ($AV2[\Delta C_i]$) denotes the evaluation of the query $AV2[\Delta C_i]$. It first checks if the query can be answered with only data available at the data warehouse. If not, it sends queries to data sources to compute the result of the query $AV2[\Delta C_i]$. After the query $AV2[\Delta C_i]$ is evaluated, ΔV and $\Delta AV1$ can be computed.

Now we explain how the algorithm works. Upon detection of a source update, a data source sends the update to the data warehouse. Upon receipt of a query from the data warehouse, a data source evaluates the query in terms of its current state and sends the result of the query to the data warehouse. Initially, WV and $WAV1$ have the same value as V and $AV1$, respectively. Upon receipt of a source update U that is a potential update to V , the data warehouse computes the changes to V and $AV1$ caused by U . These changes are not applied to V and $AV1$ immediately; instead, they are applied to WV and $WAV1$. This prevents V and $AV1$ from being in an inconsistent state. The data warehouse updates V and $AV1$ by assigning the value of WV and $WAV1$ to V and $AV1$, respectively, only when it is assured that doing so will bring V and $AV1$ to a consistent state. This occurs when UUS is empty; that is, when the effect of every received update on V has been computed and applied to WV .

When a deletion (the second category of potential update) or a modification (the third category of potential update) is received, the data warehouse finds the objects to be deleted from V

finds the objects to be deleted from V or modified in V locally without querying data sources. When an insertion (the first category of potential update) is received, the data warehouse generates a query $AV2 [\Delta C_i]$ to compute the objects to be inserted to V , ΔV , and the objects to be inserted to $AV1$, $\Delta AV1$. To evaluate the query, the data warehouse may need to query data sources. While the query is being evaluated by data sources, concurrent updates may occur at data sources. This causes two problems. The first problem is that applying ΔV and $\Delta AV1$ to V and $AV1$, respectively, may bring V and $AV1$ to an inconsistent state. This problem can be solved by the mechanism involving WV , $WAV1$, and UUS , as described in the previous paragraph. In addition, duplicate objects must not be inserted while inserting ΔV and $\Delta AV1$ to WV and $WAV1$, respectively. The second problem is that the processing of concurrent deletions or modifications at the data warehouse may miss those objects returned by the query. To remedy the problem, the data warehouse keeps a set $PU (U)$ of deletions or modifications for each insertion U . As a deletion or modification is received at the data warehouse, it is added to $PU (U)$ for each insertion U that is in UUS . After ΔV and $\Delta AV1$ have been computed with $AV2 [\Delta C_i]$ for an insertion U , each deletion or modification in $PU (U)$ is processed again to delete or modify objects of ΔV and $\Delta AV1$ that were missed. Then the updated ΔV and $\Delta AV1$ are inserted to WV and $WAV1$, respectively.

Example: Maintaining $V2$ in response to three consecutive updates

Assume initially there are five objects in the class Student whose attribute Major has the value "CS" and two objects in the class Staff whose attribute Dept has the value "CS". Two updates $U1$ and $U2$ occur at source X . One update $U3$ occurs at source Y . $U1$ inserts an object whose attribute Major has the value "CS" to the class Student. $U2$ deletes the object inserted by $U1$. $U3$ inserts an object whose attribute Dept has the value "CS" to the class Staff. Using our algorithm, the following events for maintaining $V2$ may occur at the data warehouse.

1. The warehouse receives $U1$ from source X , initializes $PU (U1)$ to be empty, and sends a query $Q1$ to source Y to compute the changes to $V2$ and $AV1_for_V2$ caused by $U1$.
2. The warehouse receives $U2$ from source X and adds $U2$ to $PU (U1)$. Because the answer of $Q1$ has not been returned, nothing is deleted from both $WV2$ and $WAV1_for_V2$ in response to $U2$.

3. The warehouse receives $U3$ from source Y , initializes $PU (U3)$ to be empty, and sends a query $Q3$ to source X to compute the changes to $V2$ and $AV1_for_V2$ caused by $U3$.
4. The warehouse receives the answer of $Q1$ from source Y with three objects in both $\Delta V2$ and $\Delta AV1_for_V2$. Because $U2$ is in $PU (U1)$, the processing of $U2$ causes both $\Delta V2$ and $\Delta AV1_for_V2$ to become empty; therefore, nothing is inserted to both $WV2$ and $WAV1_for_V2$.
5. The warehouse receives the answer of $Q3$ from source X and inserts five objects to both $WV2$ and $WAV1_for_V2$.
6. Because UUS is now empty, the warehouse replaces $V2$ and $AV1_for_V2$ with $WV2$ and $WAV1_for_V2$, respectively.

The state of $V2$ is consistent with the source state after $U1$, $U2$, and $U3$ have occurred.

5. Performance Evaluation

We have implemented a prototype system for incremental maintenance of object-oriented views in a data-warehousing environment. In the prototype system, databases are built on the Object-Store object-oriented database management system and programs are written in the C++ object-oriented programming language. The hardware platform of the prototype system is several personal computers with the following hardware components: Intel Pentium II processor (400 MHz), 256KB cache, 128MB RAM, and 6.4GB SCSI hard disk. These personal computers are connected with a local area network. A preliminary performance evaluation has been carried out using the university data warehouse whose schema is shown in Figures 2 and 3.

In our preliminary performance evaluation, we compared the execution time between incremental maintenance (IM) and recomputation (RC) of an object-oriented materialized view. Figure 7 shows the comparison of the execution time between IM and RC of the view $V1$ in response to 10 consecutive updates of various kinds to the class Student. Figure 8 shows the comparison of the execution time between IM and RC of the view $V2$ in response to 10 consecutive updates of various kinds to the class Staff. In the performance evaluation, the size of the database is varied but only the number of objects in the class Student is shown in the figures.

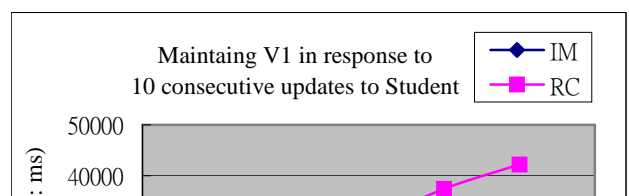


Figure 7. Execution time for maintaining V1

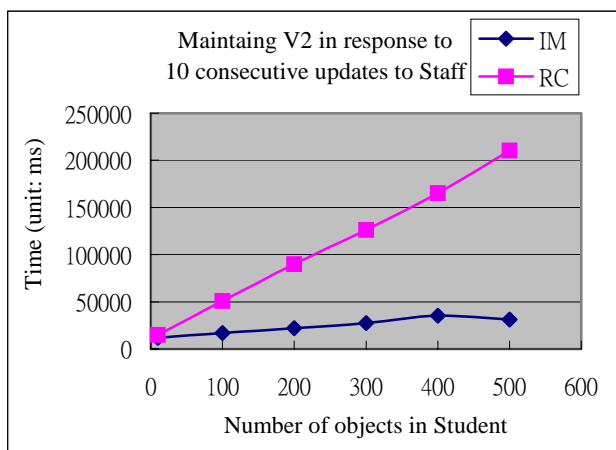


Figure 8. Execution time for maintaining V2

From our experiments, we observed the following two phenomena. First, the execution time for recomputation is proportional to the size of the database. On the other hand, the execution time for incremental maintenance is quite steady and does not depend on the size of the database. Second, the execution time for recomputation is longer than the execution time for incremental maintenance in the majority of cases. Because the size of a data warehouse tends to be very large, we can say that our incremental view maintenance algorithm outperforms recomputation and is efficient.

During the performance evaluation, we also compared the execution results between incremental maintenance and recomputation of a view. Recomputing a view is presumed to produce the correct result [11]. Because our incremental view maintenance algorithm always produces the same result as recomputing the view, we suppose that our incremental view maintenance algorithm should be correct.

6. Conclusion and Future Work

In this paper, we addressed two primary issues on the problem of incremental maintenance of object-oriented views in data warehouses. First, we identified six categories of potential updates to an object-oriented view. Second, we proposed an incremental view maintenance algorithm that is capable of maintaining an object-oriented view defined over multiple distributed data sources. Our empirical study demonstrates that our incremental view maintenance algorithm is efficient and should be correct.

In the future, we plan to study two important problems related to the problem of incremental maintenance of object-oriented views in data warehouses. First, we will study the problem of self-maintenance of object-oriented views in data warehouses. Self-maintenance of a materialized view means maintaining the view without accessing the source data, which can improve the maintenance efficiency. Second, we will study the problem of incremental maintenance of web warehouses. A web warehouse integrates data from various web sites on the Internet. The unique features of web warehouses pose more challenges to the problem of incremental maintenance of materialized views.

References

- [1] Agrawal, D., El Abbadi, A., Singh, A., and Yurek, T., "Efficient View Maintenance at Data Warehouses," in Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, Tucson, Arizona, U.S.A., pp. 417-427 (1997).
- [2] Alhadj, R. and Polat, F., "Incremental View Maintenance in Object-Oriented Databases," ACM Data Base for Advances in Information Systems, Vol. 39, pp. 52-64 (1998).
- [3] Ali, M. A., Fernandes, A. A. A., and Paton, N. W., "Incremental Maintenance of Materialized OQL Views," in Proceedings of 3rd ACM International Workshop on Data Warehousing and OLAP, Washington D.C., U.S.A., pp. 41-48 (2000).
- [4] Blakeley, J. A., Larson, P. A., and Tompa, F. W., "Efficiently Updating Materialized Views," in Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington D.C., U.S.A., pp. 61-71 (1986).
- [5] Inmon, W. H., *Building the Data Warehouse*, John Wiley and Sons, New York

- (1992).
- [6] Gupta, A., Mumick, I. S. and Subrahmanian, V.S., "Maintaining Views Incrementally," in Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., U.S.A., pp. 157-166 (1993).
 - [7] Miller, L. L., Honavar, V., Wong, J. and Nilakanta, S., "Object-Oriented Data Warehouse for Information Fusion from Heterogeneous Distributed Data and Knowledge Sources," in Proceedings of the 1998 IEEE Information Technology Conference: Information Environment for the Future, Syracuse, NY, U.S.A., pp. 27-30 (1998).
 - [8] Mumick, I. S., Quass, D. and Mumick, B. S., "Maintenance of Data Cubes and Summary Tables in a Warehouse," in Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, Tucson, Arizona, U.S.A., pp. 100-111 (1997).
 - [9] Thomann, J. and Wells, D., "Real World Objects in the Data Warehouse: the Vision," *Journal of Data Warehousing*, Vol. 2, pp. 62-65 (1997).
 - [10] Wiener, J. L., Gupta, H., Labio, W. J., Zhuge, Y., Garcia-Molina, H. and Widom, J., "A System Prototype for Warehouse View Maintenance," in Proceedings of the ACM Workshop on Materialized Views: Techniques and Applications, Montreal, Canada, pp. 26-33 (1996).
 - [11] Zhuge, Y., Garcia-Molina, H., Hammer, J. and Widom, J., "View Maintenance in a Warehousing Environment," in Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, CA, U.S.A., pp. 316-327 (1995).
 - [12] Zhuge, Y., Garcia-Molina, H. and Wiener, J. L., "The Strobe Algorithms for Multi-Source Warehouse Consistency," in Proceedings of the International Conference on Parallel and Distributed Information Systems, Miami Beach, FL, U.S.A., pp. 146-157 (1996).

Manuscript Received: May 22, 2002

Revision Received: Sept. 18, 2002

and Accepted: Nov. 12, 2002