# An Approach to Incremental Maintenance of Object-Oriented Views

Ching-Ming Chao
Department of Computer and Information Science
Soochow University, Taipei, Taiwan
chao@cis.scu.edu.tw

## ABSTRACT

The problem of incremental maintenance of materialized views has regained much attention due to the advent of data warehousing technology. So far, most of the work on this problem has been confined to relational settings. In this paper, we propose an approach to incremental maintenance of materialized views in object-oriented databases. In particular, we focus on two primary issues. The first issue is to determine the potential updates to a view. We distinguish six categories of potential updates and propose an algorithm to find the potential updates of a view from the definition of the view. The second issue is to maintain a view in response to the potential updates to the view. We propose incremental maintenance algorithms to compute and apply the change to a view in response to the potential updates to the view. We have implemented a prototype system for incremental maintenance of object-oriented views and have conducted a preliminary performance evaluation. The result shows that our approach is correct and efficient.

*Keywords: data warehousing, materialized views, incremental maintenance, object-oriented databases.*

## 1. Introduction

The concept of *materialized views* has regained much attention in the past few years due to the advent of data warehousing technology. A *data warehouse* is a global repository of integrated information primarily used for decision-making by means of online analytical processing (OLAP). A data warehouse typically extracts and integrates data from multiple heterogeneous, autonomous, and distributed data sources, and stores the integrated information as materialized views in order to provide fast access. As the source data is updated, materialized views may need to be maintained in order to keep the contents of materialized views consistent with the contents of the source data. To maintain a materialized view, there is a choice between recomputing the view from scratch and maintaining the view incrementally. To maintain a view incrementally, one computes the change to the view on the basis of the update to the source data, and applies the computed change to the view. Incremental maintenance is generally considered to be less expensive when the size of the update to the source data is small compared to the size of the source data.

In this paper, we study the problem of incremental maintenance of materialized views in object-oriented databases. We study the problem in a centralized database environment. That is, we assume that materialized views and source data are stored at the same site, which implies that all data required for incrementally maintaining materialized views are available without having to request any data from other sites. We consider a large class of object-oriented views and three types of updates to the source data: insertion, deletion, and modification.

Object-oriented databases have many unique features that are absent from relational databases, such as object identity, complex attributes, inter-object reference, class inheritance, etc. These unique features make the incremental maintenance of materialized views in object-oriented databases different from the incremental maintenance of materialized views in relational databases. We highlight some of the differences here. First, updates to certain classes not explicitly appearing in the definition of an object-oriented view may cause changes to the view. In contrast, only updates to tables explicitly appearing in the definition of a relational view can possibly affect the view. Second, computing and applying the change for an object-oriented view is generally more complicated than that for a relational view.

In this paper, we propose an approach to incremental maintenance of object-oriented views. In particular, we focus on two primary issues. The first issue is to determine during view compilation time which kinds of updates to which classes may cause changes to a view given the definition of the view.

Such updates are called the *potential updates* to the view. We distinguish six categories of potential updates and propose an algorithm to find the potential updates to a view from the definition of the view. The second issue is to maintain a view in response to the potential updates to the view. We propose incremental maintenance algorithms to compute and apply the change to a view in response to the potential updates to the view. We have implemented a prototype system for incremental maintenance of object-oriented views and have conducted a preliminary performance evaluation. The result shows that our approach to incremental maintenance of object-oriented views is correct and efficient.

The remainder of this paper is organized as follows. In Section 2 we review previous work on incremental maintenance of materialized views that is closely related to our work. In Section 3 we describe the overall process of incremental view maintenance in our approach. In Section 4 we concentrate on the issue of determining the potential updates to a view. In Section 5 we present our algorithms for incremental view maintenance. In Section 6 we show the results of our performance evaluation. Section 7 concludes this paper and gives some directions for future research.

## 2. Related Work

The problem of incremental maintenance of materialized views was first studied for relational databases in a centralized environment. Blakeley et al. [6] proposed a differential algorithm for maintaining select-project-join (SPJ) views. A portion of our view maintenance algorithms is based on this differential algorithm. Blakeley et al. [5] proposed necessary and sufficient conditions for determining at run time whether an update of a base relation cannot affect a view regardless of the database state (an *irrelevant update* to the view). The class of views considered was restricted to SPJ views. In contrast, our approach determines potential updates during view compilation time. Only potential updates are propagated and the maintenance process is terminated as soon as it is discovered that the potential update cannot affect the view. Gupta et al. [7] presented incremental algorithms to compute changes to SQL and Datalog views in response to updates to source relations. Their algorithms require access to source data for all updates, while our algorithms avoid access to source data for some updates to improve efficiency. Gupta and Mumick [8] gave a survey on the problems, techniques, and applications of view maintenance.

Later on, research on incremental view maintenance for relational databases was extended to a warehousing environment, where the data warehouse and data sources are decoupled. Zhuge et al. [14] showed that anomalies could occur if conventional view maintenance algorithms are used in a warehousing environment and proposed an incremental view maintenance algorithm, called Eager Compensating Algorithm (ECA), which is suitable in a warehousing environment. The ECA algorithm assumes that a data warehouse derives data from a single source. Zhuge et al. [15] later presented a family of incremental view maintenance algorithms, called Strobe algorithms, for a data warehouse derived from multiple data sources. Agrawal et al. [2] also proposed two incremental view maintenance algorithms, called SWEEP and Nested SWEEP, for a data warehouse derived from multiple distributed autonomous data sources. These two algorithms are more efficient than Strobe algorithms.

The problem of self-maintenance of materialized views is important, especially in data warehousing environments, and has attracted a lot of attention. A materialized view is *self-maintainable* if it can be maintained without accessing the source data [9]. Gupta et al. [9] derived conditions under which several types of SPJ views are self-maintainable upon insertions, deletions, and updates. Quass et al. [12] proposed an algorithm to derive a minimal set of auxiliary views for a single view such that the view and its auxiliary views together are self-maintainable. Huyn [10] proposed algorithms that test whether a view is self-maintainable with access to all views in a data warehouse. Samtani et al. [13] proposed a set of auxiliary views for a set of materialized views such that a view is self-maintainable with access to the set of materialized views and the set of auxiliary views. While [12,13] makes a view self-maintainable by additionally materializing auxiliary views that contain a subset of the source data, we only store the OIDs of objects that derive objects in the materialized views. Although currently not all views are self-maintainable, our approach avoids access to source data as much as possible.

Recently, techniques for incrementally maintaining materialized views in data models other than the relational model have been investigated. Zhuge and Garcia-Monila [16] investigated the problem of incrementally maintaining graph-structured views. Abiteboul et al. [1] studied the problem of incremental maintenance of materialized views over semistructured data. Alhajj and Polat [3] investigated the problem of incremental view maintenance in object-oriented databases. They proposed a model that facilitates incremental maintenance of single class based views by employing the deferred update mode. Although they proposed that updates to classes not

explicitly appearing in the definition of a view might affect the view, they did not provide an algorithm to determine the potential updates to a view. They also did not give a complete algorithm to compute and apply the change to a view. Liu et al. [11] investigated the problem of incrementally maintaining materialized views in object-relational databases. Ali et al. [4] proposed a solution to the problem of incremental maintenance of OQL views. They gave an algorithm to determine potential updates from a view definition but their algorithm did not consider updates to classes not explicitly appearing in the view definition. They described two types of incremental maintenance plans and how to choose a maintenance plan on the basis of the update type and the view type. However, they did not provide detailed algorithms to compute and apply the change to a view in response to the potential updates to the view.

## 3. Overall View Maintenance Process

In this section, we describe the overall process of incremental maintenance of object-oriented views in our approach. Our approach to incremental maintenance of object-oriented views is general and is not restricted to any specific object data model. Therefore, we adopt a generic object data model and language to describe the ideas and examples. We will use a simplified university database as the running example for the rest of this paper. The university database contains six base classes whose definition is shown in Figure 1.

```
class Person
{Name: string, Age: integer, Sex: char,
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,
Prerequisite: set (Course)};
class Department
{Name: string, Head: Staff};
```

Figure 1. Class Definition

The general form of a view definition in this paper is as follows.

**view** V ($A_1$: $T_1$, …, $A_r$: $T_r$)
**select** $A_{S1}$, …, $A_{Sm}$, $P_{S1}$, …, $P_{Sr\text{-}m}$
**from** $C_1$, …, $C_n$
**where** *pred* ($A_{W1}$, …, $A_{Wy}$, $P_{W1}$, …, $P_{Wz}$);

where

- V is the name of the view.
- $A_1$, …, $A_r$ are the attribute names of the view and $T_1$, …, $T_r$ are their corresponding types.
- $C_1$, …, $C_n$ are the names of the defining classes of the view.
- $A_{S1}$, …, $A_{Sm}$, $A_{W1}$, …, $A_{Wy}$ are qualified attribute names and have the form C.A where C is the name of a defining class and A is an attribute name of C.
- $P_{S1}$, …, $P_{Sr\text{-}m}$, $P_{W1}$, …, $P_{Wz}$ are path expressions and have the form $C.A_1.....A_x$ where C is the name of a defining class and $A_1.....A_x$ ($x \geq 2$) are attribute names such that $A_1$ is an attribute of C and $A_i$ ($i = 2, …, x$) is an attribute of the class $C_{i-1}$ that is the type of the attribute $A_{i-1}$.
- *pred* ($A_{W1}$, …, $A_{Wy}$, $P_{W1}$, …, $P_{Wz}$) is a condition defined over $A_{W1}$, …, $A_{Wy}$, $P_{W1}$, …, $P_{Wz}$.

The university database contains two materialized views whose definition is shown in Figure 2.

```
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, CN: string, CC: integer)
select  Student.Name, Course.Name, Course.Credit
from    Student, Course
where   Student.Major = "CS"
and     Course in Student.Courses
and     Course.Credit > 1 ;
```

Figure 2. View Definition

The overall process of view maintenance in our approach is shown in Figure 3 in which boxes indicate the actions taken in the maintenance process. The figure is functionally divided into two parts by a dotted line, the top part and the bottom part. Boxes in the top part constitute the preparation process for maintaining a view and are executed only once for each view. Given the definition of a view, the box labeled "Determine potential updates" finds the po-

tential updates to the view. Note that this box does not produce any update that can be determined not to affect a view according to the definition of the view. We will discuss the details of this box in Section 4. For each kind of potential update to a view, the box labeled "Create triggers" creates a trigger to detect occurrences of that kind of potential update.

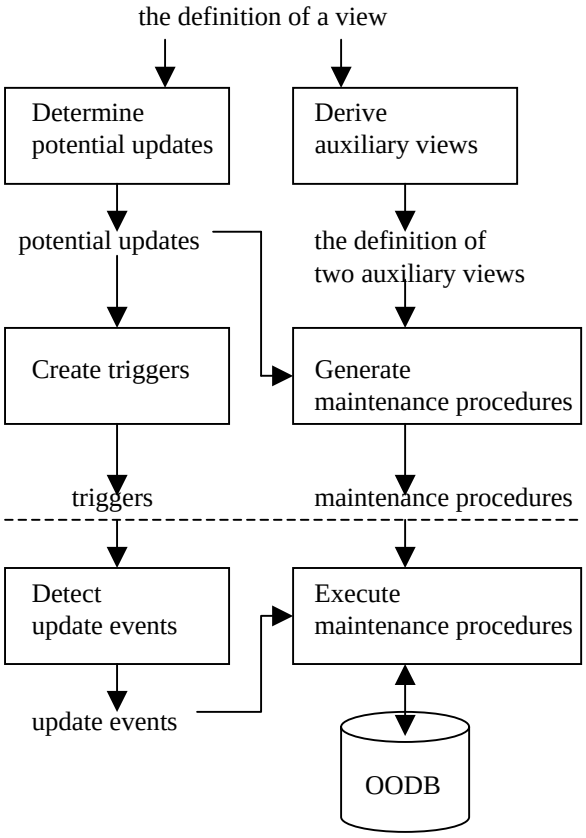


Figure 3. View Maintenance Process

For each materialized view V, the box labeled "Derive auxiliary views" derives two auxiliary views, AV1_for_V and AV2_for_V, which are used to assist in the maintenance of V. The view AV1_for_V is materialized but the view AV2_for_V is not materialized. For example, the definitions of the auxiliary views derived from the views V1 and V2 are shown in Figure 4 and Figure 5, respectively. AV1_for_V stores the OIDs of objects that derive objects of V. More specifically, the OIDs of objects in the defining classes of V that derive an object of V are associated with the OID of that object in AV1_for_V. Since AV1_for_V is materialized, there will be additional space overhead to store it and time overhead to maintain it. However, these additional overheads can be compensated by significant time saving in maintaining V. AV2_for_V is almost identical to V except that it includes additional attributes for the OIDs of objects that derive objects of V, if V does not include those attributes already. The precise usage and advantages for introducing these two auxiliary views will be seen in Section 5. The derivation of the definition of auxiliary views from the definition of a given view is a simple syntactic mapping. For example, the generated strings in the definition of auxiliary views are shown in italics in Figure 4. Note that the idea of using these two auxiliary views for maintaining views is not new and was also adopted in [4].

**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. Auxiliary Views for View V1

**view** *AV1_for_*V2 *(SO: Student, CO: Course, VO: V2)*
**select** *Student, Course,V2*
**from** Student, Course, *V2*
**where** Student.Major = "CS"
and Course in Student.Courses
and Course.Credit > 1
*and* *V2.SN = Student.Name*
*and* *V2.CN = Course.Name*
*and* *V2.CC = Course.Credit* ;

**view** *AV2_for_*V2 (*SO: Student, CO: Course,* SN: string, CN: string, CC: integer)
**select** *Student, Course,* Student.Name, Course.Name, Course.Credit
**from** Student, Course
**where** Student.Major = "CS"
and Course in Student.Courses
and Course.Credit > 1 ;

Figure 5. Auxiliary Views for View V2

The box labeled “Generate maintenance procedures” generates maintenance procedures for the potential update of a view. The generation of maintenance procedures for a view needs to refer to the definition of its auxiliary views. We will discuss the details of this box in Section 5.

Boxes in the bottom part of Figure 3 constitute the actual maintenance process for maintaining a view. When potential updates to a view occur, the corresponding maintenance procedures are triggered and executed to maintain the view and the auxiliary view AV1_for_V.

## 4. Determination of Potential Updates

In this section, we address the issue of determining the potential updates to a view. In our approach to view maintenance, it has to be determined during view compilation time which kinds of updates to which classes may cause changes to a view. Such updates are called the *potential updates* to the view. The box labeled “Determine potential updates” in Figure 3 finds the potential updates to a view from the definition of the view. Note that it only produces the potential updates to a view; i.e., it does not produce any update that can be determined not to affect a view according to the definition of the view. This is a significant improvement on the efficiency of view maintenance because detecting and propagating updates that cannot possibly affect a view is meaningless and a waste of time.

The issue of determining the potential updates to a view was almost not discussed in the literature on incremental maintenance of relational views mostly because it is very simple. The only tables whose updates may cause changes to a relational view are tables that appear in the FROM clause of the view definition. Because of several unique features of object-oriented databases such as class inheritance, inter-object reference, and path expressions, the issue of determining the potential updates to an object-oriented view is more complicated. This issue has been addressed in the literature on incremental maintenance of object-oriented views and object-relational views; e.g., [3,4,11]. However, to the best our knowledge, no comprehensive and satisfactory solution has been provided to determine the potential updates to an object-oriented view.

In the rest of this section, we will first give a comprehensive discussion about whether or not a particular kind of update to a particular kind of class is a potential update to an object-oriented view. Then we conclude the discussion by identifying six categories of potential updates to a view. Finally, we propose an algorithm to find the potential updates of a view from the definition of the view.

For the purpose of determining potential updates, we distinguish four different roles that a class can play for a view: a defining class, a referenced class, an inheriting class, and an irrelevant class. A class is a *defining class* of a view if the class appears in the FROM clause of the view. For example, the class Student is the only defining class of the view V1 and the classes Student and Course are the defining classes of the view V2. 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. For example, Course, Department, and Staff are the referenced classes of V1. Note that the class Person, although is referenced by the class Student in the class composition hierarchy, is not a referenced class of V1 because it is not referenced from Student within any path expression of V1. A class is an *inheriting class* of a view if the class directly or indirectly inherits a defining class or a referenced class of the view. For example, the class 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 it does not play any of the three roles mentioned above 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 illustrate 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 class or a 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 is therefore an irrelevant class of V1. Updates to objects of Person that are not objects of any of its subclasses cannot cause any change to V1. Second, a class that is referenced by a defining class only in the class composition hierarchy but not within any path expression of a view is also 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 class obviously cannot cause any change to the view.

Updates to a defining class, a referenced class, or an inheriting class of a view may cause changes to the view, but only for certain kinds of updates. Again for the purpose of determining potential updates, we distinguish five kinds of updates to a class as follows.

1. Insertion
2. Deletion
3. Modification of SELECT attributes (i.e., attrib-

utes that appear only in the SELECT clause of the view)
4. Modification of WHERE attributes (i.e., attributes that appear in the WHERE clause of the view)
5. 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 Course object will cause all objects derived from the deleted Course 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, deletion, or modification to a view. For example, changing 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 updates to 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 and deletion of 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, deletion, or modification to a view. For example, changing 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 class 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.

Based on the discussion above, we conclude that the following are the potential updates to a view, which are classified into six categories.

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** Gradate.Courses
5. **MRS** Department.Head; **MRS** Staff.Name — **MRS** Staff.Age
6. **MRW** Course.Name

Our algorithm to find the potential updates to a view is shown in Figure 6. Given the definition of a view, this algorithm produces the six categories of potential updates to the view.

## 5. View Maintenance Algorithms

In this section, we address the issue of maintaining a view in response to the potential updates to the view. In particular, we propose incremental algorithms for maintaining a view. In our approach to

view maintenance, only the potential updates to a view will be propagated to maintain the view. In Section 4 we have classified the potential updates to a view into six categories and have described the effect of each of these six categories of potential updates on the view. Therefore, we will give one incremental maintenance algorithm for each of the six categories of potential updates of a view.

**Algorithm** FindPotentialUpdates
/* This algorithm finds the potential updates to a view from the definition of the view. */
**Input**: the definition of a view V
**Output**: a set PU of the potential updates to V
**Steps**:
{PU := ∅;
Find the set DC of defining classes of V;
**foreach** defining class dc **in** DC **do**
{PU := PU ∪ {**Inc** dc, **Del** dc};
Find the set DCSA of attributes of dc that appear only in the SELECT clause of V;
**foreach** attribute a **in** DCSA **do**
{PU := PU ∪ {**MDS** dc.a}};
Find the set DCWA of attributes of dc that appear in the WHERE clause of V;
**foreach** attribute a **in** DCWA **do**
{PU := PU ∪ {**MDW** dc.a}};
Find the set DCIC of inheriting classes of dc;
**foreach** inheriting class ic **in** DCIC **do**
{PU := PU ∪ {**Ins** ic, **Del** ic};
**foreach** attribute a **in** DCSA **do**
{PU := PU ∪ {**MDS** ic.a}};
**foreach** attribute a **in** DCWA **do**
{PU := PU ∪ {**MDW** ic.a}}};
Find the set RC of referenced classes of dc;
**foreach** referenced class rc **in** RC **do**
{Find the set RCSA of attributes of rc that appear only in the SELECT clause of V;
**foreach** attribute a **in** RCSA **do**
{PU := PU ∪ {**MRS** rc.a}};
Find the set RCWA of attributes of rc that appear in the WHERE clause of V;
**foreach** attribute a **in** RCWA **do**
{PU := PU ∪ {**MRW** rc.a}};
Find the set RCIC of inheriting classes of rc;
**foreach** inheriting class ic **in** RCIC **do**
{**foreach** attribute a **in** RCSA **do**
{PU := PU ∪ {**MRS** ic.a}};
**foreach** attribute a **in** RCWA **do**
{PU := PU ∪ {**MRW** ic.a}}}}}
**End Algorithm**.

Figure 6. Algorithm to Find Potential Updates

Our view maintenance algorithms have several salient features that can improve maintenance efficiency significantly. First, whenever a source modification will not cause insertion to or deletion from a view, we do not treat such modification as a deletion followed by an insertion as most of the view maintenance algorithms do. Second, as described in Section 3, we use two auxiliary views AV1_for_V and AV2_for_V to assist in the maintenance of a view V. 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. Finally, note that a potential update of a view does not necessarily cause any change to the view. For a particular occurrence of a potential update to a view, therefore, our maintenance procedures will terminate whenever it is discovered that the view cannot be affected by this update occurrence. In the following, we present six incremental maintenance algorithms that compute and apply the changes to the views V and AV1_for_V for the six categories of the potential updates to V.

**Algorithm 5.1**
/* This algorithm is triggered by insertion to a defining class (or an inheriting class that inherits a defining class) to maintain V and AV1_for_V. */
**Input:** The name of the defining class dc and the inserted objects Δdc.
**Steps:**
1. Compute the objects to be inserted to V, ΔV, and the objects to be inserted to AV1_for_V, ΔAV1_for_V, by substituting Δdc for dc in AV2_for_V. Stop the algorithm if ΔV is empty.
2. Insert ΔV to V.
3. Insert ΔAV1_for_V to AV1_for_V.

For example, let us see how the view V1 is maintained according to Algorithm 5.1 if a collection of objects of type Student, ΔStudent, is inserted to the class Student. Step 1 computes the objects to be inserted to V1, ΔV1, and the objects to be inserted to AV1_for_V1, ΔAV1_for_V1, by evaluating the following expression.

**select** Δ*Student*, ΔStudent.Name,
ΔStudent.Courses.Name,
ΔStudent.Major.Head.Name,
ΔStudent.Major.Head.Age
**from** ΔStudent
**where** ΔStudent.Year = 4
and "BCC" in ΔStudent.Courses.Name

If ΔV1 is empty, the algorithm is terminated; otherwise, steps 2 and 3 insert ΔV1 and ΔAV1_for_V1 to

V1 and AV1_for_V1, respectively.

**Algorithm 5.2**
/* This algorithm is triggered by deletion from a defining class (or an inheriting class that inherits a defining class) to maintain V and AV1_for_V. */
**Input:** The name of the defining class dc and the deleted objects ∇dc.
**Steps:**
1. Find the objects to be deleted from V, ∇V, and the objects to be deleted from AV1_for_V, ∇AV1_for_V, by joining ∇dc with AV1_for_V. Stop the algorithm if ∇V is empty.
2. Delete ∇V from V.
3. Delete ∇AV1_for_V from AV1_for_V.

For example, let us see how the view V2 is maintained according to Algorithm 5.2 if a collection of objects of type Student, ∇Student, is deleted from the class Student. In step 1, the OIDs of objects of ∇Student are searched in AV1_for_V2 to find the OIDs of objects to be deleted from V2, ∇V2, and the objects to be deleted from AV1_for_V2, ∇AV1_for_V2. If ∇V2 is empty, the algorithm is terminated; otherwise, steps 2 and 3 delete ∇V2 and ∇AV1_for_V2 from V2 and AV1_for_V2, respectively.

**Algorithm 5.3**
/* This algorithm is triggered by modification of SELECT attributes of a defining class (or an inheriting class that inherits a defining class) to maintain V. AV1_for_V needs not be maintained. */
**Input:** The name of the defining class dc, the names and new values of the modified attributes, and the modified object ◊dc.
**Steps:**
1. Find the objects of V to be modified, ◊V, by joining ◊dc with AV1_for_V. Stop the algorithm if ◊V is empty.
2. Determine the affected attributes in V and compute new values for those attributes.
3. Modify the affected attributes of ◊V with new values computed in step 2.

For example, let us see how the view V1 is maintained according to Algorithm 5.3 if the attribute Major of a Student object is modified. Step 1 finds those objects of V1, ◊V1, that are derived from the modified Student object and are to be modified. If ◊V1 is empty, then V1 needs not to be maintained and the algorithm is terminated. Step 2 determines that the attributes HN and HA in V1 are affected and computes new values for those attributes. Step 3 modifies the attributes HN and HA of objects ◊V1 with new values computed in step 2.

**Algorithm 5.4**
/* This algorithm is triggered by modification of WHERE attributes on a defining class (or an inheriting class that inherits a defining class) to maintain V and AV1_for_V. */
**Input:** The name of the defining class dc, the names and new values of the modified attributes, and the modified object ◊dc.
**Steps:**
1. Find the objects to be deleted from V, ∇V, and the objects to be deleted from AV1_for_V, ∇AV1_for_V, by joining ◊dc with AV1_for_V. Jump to step 4 if ∇V is empty.
2. Delete ∇V from V.
3. Delete ∇AV1_for_V from AV1_for_V.
4. Compute the objects to be inserted to V, ΔV, and the objects to be inserted to AV1_for_V, ΔAV1_for_V, by substituting ◊dc (with modified attribute values) for dc in AV2_for_V. Stop the algorithm if ΔV is empty.
5. Insert ΔV to V.
6. Insert ΔAV1_for_V to AV1_for_V.

The rationale for Algorithm 5.4 is as follows. Modification of WHERE attributes results in one of three cases. In the first case, where the modified object does not derive data to the view both before and after the modification, the view is not affected by this modification. In the second case, this modification will cause objects to be inserted to and/or deleted from the view. In the third case, this modification will not cause any objects to be inserted or deleted from the view but may cause some of the attributes of the view to be modified. Modification can be handled by a deletion followed by an insertion. Therefore, to handle these three possible cases efficiently, Algorithm 5.4 first finds and deletes objects from V and AV1_for_V and then computes and inserts objects into V and AV1_for_V.

**Algorithm 5.5**
/* This algorithm is triggered by modification of SELECT attributes of a referenced class (or an inheriting class that inherits a referenced class) to maintain V. AV1_for_V needs not be maintained. */
**Input:** The name of the referenced class rc, the names and new values of the modified attributes, and the modified object ◊rc.
**Steps:**
1. Let dc be the defining class of rc. Find the objects of dc that reference the modified object, ◊dc. Stop the algorithm if ◊dc is empty.
2. Find the objects to be modified in V, ◊V, by

joining ◊dc with AV1_for_V. Stop the algorithm if ◊V is empty.
3. Determine the affected attributes in V and compute new values for those attributes.
4. Modify the affected attributes of ◊V with new values computed in step 3.

For example, let us see how the view V1 is maintained according to Algorithm 5.5 if the attribute Head of a Department object is modified. The class Student is the defining class of the class Department. Step 1 finds those Student objects that reference the modified Department object, ◊Student. If ◊Student is empty, then V1 is not affected and the algorithm is terminated. Step 2 finds those objects of V1, ◊V1, that are derived from ◊Student and are to be modified. If ◊V1 is empty, then again V1 is not affected and the algorithm is terminated. Step 3 determines that the attributes HN and HA in V1 are affected and computes new values for those attributes. Step 4 modifies the attributes HN and HA of objects ◊V1 with new values computed in step 3.

**Algorithm 5.6**
/* This algorithm is triggered by modification of WHERE attributes of a referenced class (or an inheriting class that inherits a referenced class) to maintain V and AV1_for_V. */
**Input:** The name of the referenced class rc, the names and new values of the modified attributes, and the modified object ◊rc.
**Steps:**
1. Let dc be the defining class of rc. Find the objects of dc that reference the modified object, ◊dc. Stop the algorithm if ◊dc is empty.
2. Find the objects to be deleted from V, ∇V, and the objects to be deleted from AV1_for_V, ∇AV1_for_V, by joining ◊dc with AV1_for_V.
   Jump to step 5 if ∇V is empty.
3. Delete ∇V from V.
4. Delete ∇AV1_for_V from AV1_for_V.
5. Compute the objects to be inserted to V, ΔV, and the objects to be inserted to AV1_for_V, ΔAV1_for_V, by substituting ◊dc for dc in AV2_for_V.
   Stop the algorithm if ΔV is empty.
6. Insert ΔV into V.
7. Insert ΔAV1_for_V into AV1_for_V.

The rationale for Algorithm 5.6 is a combination of those of Algorithm 5.4 and Algorithm 5.5. First, one has to find objects of the defining class that reference the modified object of the referenced class. Then, objects to be deleted from and/or inserted into the view are computed and applied to the view.

## 6. Performance Evaluation

We have implemented a prototype system for incremental maintenance of object-oriented views in a centralized environment. In the prototype system, databases are built on the ObjectStore object-oriented database management system and programs are written in the C++ object-oriented programming language. A preliminary performance evaluation has been carried out on a PC with the following hardware components: Intel Pentium II processor (400 MHz), 256KB cache, 128MB RAM, and 6.4GB SCSI hard disk. The database used in the performance evaluation is the university database shown in Figures 1 and 2. The numbers of objects in the classes Person, Student, Staff, Graduate, Course, and Department are approximately 100, 1000, 100, 20, 50, and 20, respectively. The numbers of objects in the views V1 and V2 are approximately 200 and 1000, respectively. We compare the execution time between incremental maintenance (IM) and recomputation (RC) of a materialized view in response to various potential updates to the view.

Figure 7 compares the execution time between IM and RC of V1 in response to inserting objects into Student. Figure 8 compares the execution time between IM and RC of V2 in response to deleting objects from Student. Figure 9 compares the execution time between IM and RC of V1 in response to modifying the attribute Major of Student objects. Figure 10 compares the execution time between IM and RC of V1 in response to modifying the attribute Year of Student objects. Figure 11 compares the execution time between IM and RC of V1 in response to modifying the attribute Head of Department objects. Figure 12 compares the execution time between IM and RC of V1 in response to modifying the attribute Name of Course objects. The update size in Figures 7 to 10 means the number of updated Student objects. The update size in Figures 11 and 12 means the number of Student objects that reference the modified objects. Measuring the update size in terms of the number of Student objects in the last two cases can express the effect of the update to the view more accurately.

Base on our empirical study, we come to the following two conclusions. First, our algorithms for determining potential updates and incrementally maintaining materialized views are correct. Second, our incremental maintenance algorithms are efficient because they significantly outperform recomputation in the majority of cases. It is until about 60% to 80% of the update percentage that our incremental algorithms are more expensive than recomputation.

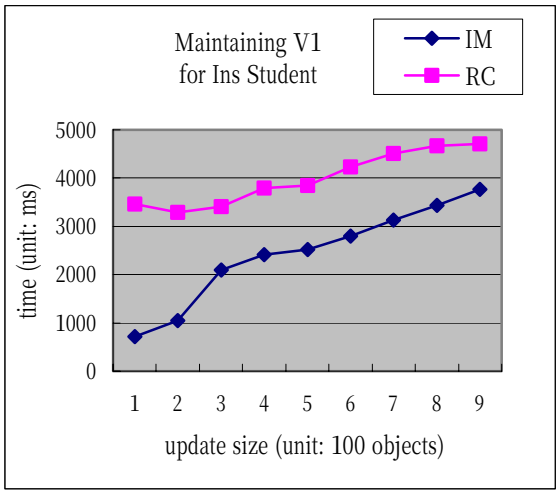


Figure 7. First Category of Potential Updates

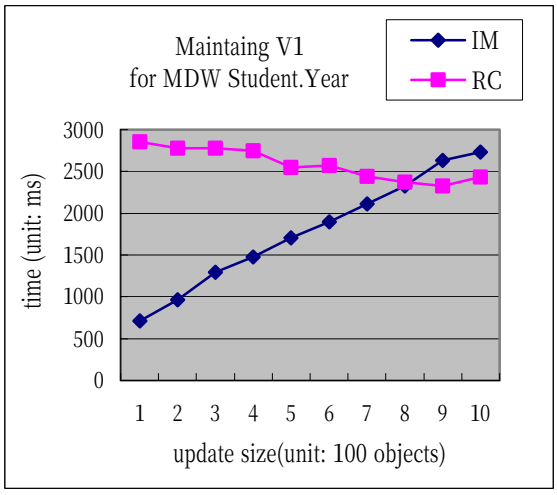


Figure 10. Fourth Category of Potential Updates

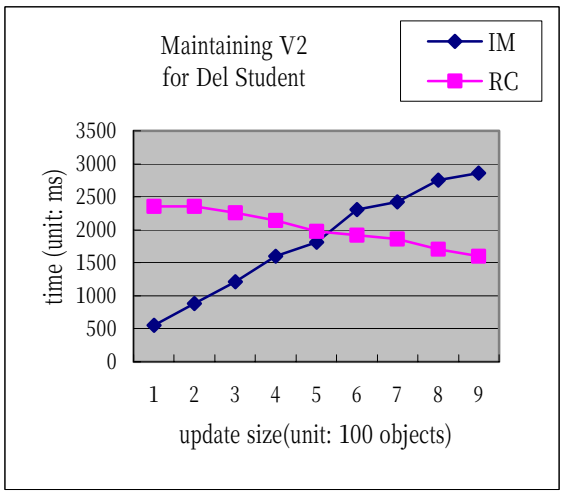


Figure 8. Second Category of Potential Updates

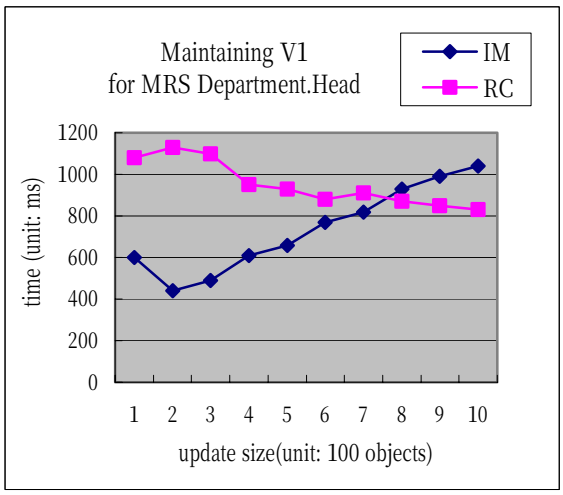


Figure 11. Fifth Category of Potential Updates

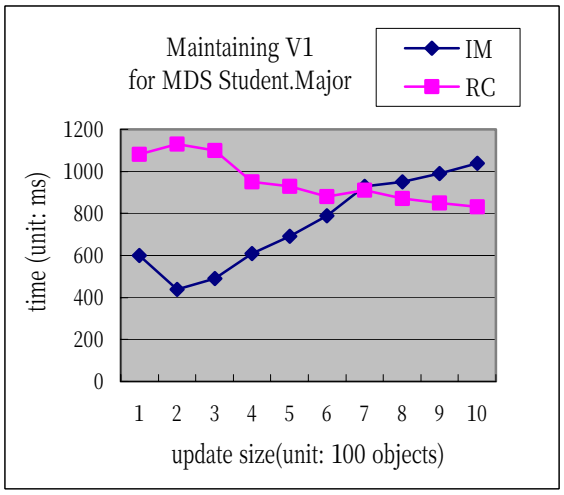


Figure 9. Third Category of Potential Updates

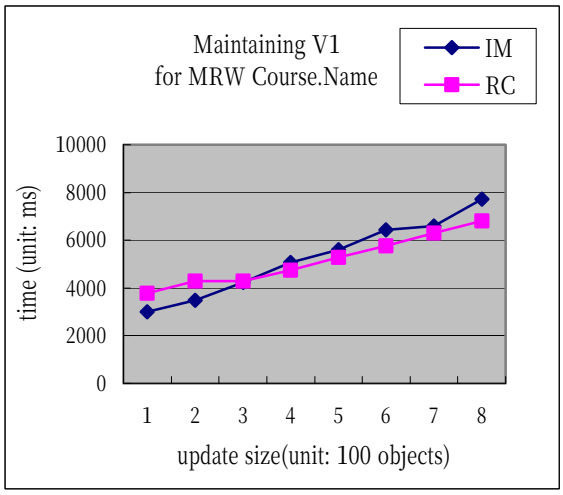


Figure 12. Sixth Category of Potential Updates

## 7. Conclusion and Future Work

Data warehousing is an emerging and important technology for information integration and decision support. Incremental maintenance of materialized views is a major issue in data warehousing. Most of the previous work on this problem has been confined to relational databases. This paper is one of few that study the problem of incremental maintenance of materialized views in object-oriented databases. There are two major contributions in this paper. First, we gave a comprehensive discussion of various updates to a view and classified six categories of potential updates to a view. Second, we proposed detailed algorithms for incrementally maintaining a view in response to potential updates to the view. Our empirical study shows that our approach to view maintenance is correct and efficient.

We plan to study two important problems on incremental maintenance of object-oriented views. First, we will study how to incrementally maintain an object-oriented view in a distributed environment where the materialized views and the source data are decoupled. Second, we will study the problem of self-maintenance of object-oriented views.

**Acknowledgements**

The author would like to thank students in my Senior Project course for implementing the prototype system and conducting performance evaluation.

## 8. References

[1] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J.L. Wiener, "Incremental Maintenance for Materialized Views over Semistructured Data," in *Proceedings of the 24th International Conference on Very Large Data Bases*, New York City, New York, USA, August 1998, pp. 38-49.

[2] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek, "Efficient View Maintenance at Data Warehouses," in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, USA, May 1997, pp. 417-427

[3] R. Alhajj and F. Polat, "Incremental View Maintenance in Object-Oriented Databases," *ACM Data Base for Advances in Information Systems*, Vol. 39, No. 3, 1998, pp. 52-64.

[4] M.A. Ali, A.A.A. Fernandes, and N.W. Paton, "Incremental Maintenance of Materialized OQL Views," in *Proceedings of 3rd ACM International Workshop on Data Warehousing and OLAP (DOLAP 2000)*, Washington D.C., USA, November 2000.

[5] J.A. Blakeley, N. Coburn, and P. Larson, "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates," *ACM Transactions on Database Systems*, Vol. 14, No. 3, September 1989, pp. 369-400.

[6] J.A. Blakeley, P.A. Larson, and F.W. Tompa, "Efficiently Updating Materialized Views," in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, Washington D.C., USA, June 1986, pp. 61-71.

[7] A. Gupta, I.S. Mumick, and V.S. Subrahmanian, "Maintaining Views Incrementally," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, D.C., USA, May 1993, pp. 157-166.

[8] A. Gupta and I.S. Mumick, "Maintenance of Materialized Views: Problems, Techniques, and Applications," *IEEE Data Engineering Bulletin*, Vol. 18, No. 2, June 1995, pp. 3-18.

[9] A. Gupta, H.V. Jagadish, and I.S. Mumick, "Data Integration Using Self-Maintainable Views," in *Proceedings of the 5th International Conference on Extending Database Technology*, Avignon, France, March 1996, pp. 140-144.

[10] N. Huyn, "Multiple-View Self-Maintenance in Data Warehousing Environments," in *Proceedings of the 23rd International Conference on Very Large Data Bases*, Athens, Greece, August 1997, pp. 26-35.

[11] J. Liu, M. Vincent, and M. Mohania, " Maintaining Views in Object-Relational Databases," in *Proceedings of the 9th International Conference on Information and Knowledge Management*, McLean, VA, USA, November 2000, pp. 102-109.

[12] D. Quass, A. Gupta, I.S. Mumick, and J. Widom, "Making Views Self-maintainable for Data Warehousing," in *Proceedings of the 4th International Conference on Parallel and Distributed Information Systems*, Miami Beach, FL, December 1996, pp. 158-169.

[13] S. Samtani, V. Kumar, and M. Mohania, "Self Maintenance of Multiple Views in Data Warehousing," in *Proceedings of the 8th International Conference on Information and Knowledge Management*, Kansas City, MO, USA, November 2-6, 1999, pp. 292-299.

[14] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom, "View Maintenance in a Warehousing Environment," in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, CA, USA, May 1995, pp. 316-327.

[15] Y. Zhuge, H. Garcia-Molina, and J.L. Wiener, "The Strobe Algorithms for Multi-Source Ware-

house Consistency," in *Proceedings of the International Conference on Parallel and Distributed Information Systems*, Miami Beach, FL, USA, December 1996, pp. 146-157.

[16] Y. Zhuge and H. Garcia-Molina, "Graph Structured Views and Their Incremental Maintenance," in *Proceedings of the 14th International Conference on Data Engineering*, Orlando, FL, USA, February 1998, pp. 116-125.