View Maintenance in Object-Oriented Databases

Reda Alhajj[1] and Faruk Polat[2]

[1] Department of Computer Engineering and Information Science, Bilkent University, 06533 Ankara, Turkey

[2] Department of Computer Engineering, Middle East Technical University, 06531 Ankara, Turkey

Abstract. In this paper, we present a model that facilitates view maintenance within object-oriented databases. For that purpose, we differentiate between two categories of classes, *base* classes and *brother* classes. While the former constitute the actual database, the latter are introduced to hold virtual database, i.e., views derived from base classes. To achieve incremental view update, we introduce a modification list into each base class. A series of algorithms are developed to serve the purpose. Finally it happened that, view maintenance within object-oriented databases subsumes that within the nested and hence conventional relational models.

Keywords: Algorithms; Base Classes; Modification Lists; Object-Oriented Databases; Views.

1 Introduction

Views are derived (virtual) data that can be materialized and subsequently queried against. View maintenance is an important aspect of query models for having a lot of application areas ranging from integrity constraint maintenance to persistent queries among others. However, the difficulty and complexity of a view maintenance approach is dependent on the underlying data model. Although there have been man studies on view maintenance and materialization within the relational model [7, 8], much effort is still required on view maintenance within the nested relational model and object-oriented data models in addition to the effort done in this respect e.g. [1, 2, 6, 9, 10, 11].

In object-oriented databases, through the use of object identifiers and virtual data, the view mechanism can be implemented correctly. Any update made through a view would be an indirect update, because the real data would be accessed through the use of pointers. No replication is necessary, so no inconsistencies due to the generation or updating of views emerge. We argue that deferred update is more reasonable within the object-oriented context, because updates are most of the time due to indirect modifications and hence require much more effort to be reflected into related views. We should not speak about the violation of referential integrity unless objects are accessed. Only at that time references to deleted objects should be recognized and the violation of referential integrity should be checked for. Earlier recognition might be time wasting and useless.

In this paper, we describe a model which facilitates view maintenance in object-oriented databases. The basic idea in our approach is to distinguish modifications to objects on which a view is dependent since the last derivation (update) of that view. Therefore, in any subsequent view update only such modifications are considered and hence the number of objects to be accessed while updating a view is reduced. To achieve that, we categorize classes into *base* classes and *brother* classes. The latter correspond to view definitions and the former hold the actual database. Further, to each base class we add a *modification list* which keeps track of all modifications to be reflected on demand to dependent views. Different algorithms have been developed to handle the view maintenance.

The rest of this paper is organized as follows. The basic model on which our work is based, is presented in Section 2. In Section 3, we give the algorithms that govern and guarantee incremental view maintenance and materialization. Section 4 is the conclusions.

2 The Basic Model

In this section we introduce the data model that facilitates incremental view maintenance. To start with, any object with an *object identifier* o_{id} qualifies to be considered in the set of objects of a class c, denoted by $L_{instances}(c)$, if and only if object o_{id} understands nothing more than the behavior defined for objects of class c. The behavior for class c consists of two parts, *inherited behavior* and *locally defined behavior.* So, let $L_{behavior}(c)$ denotes the local behavior for class c. Then, the whole behavior, object instances and attributes for class c are recursively defined based on its direct superclasses, say $[c_{p_1}, c_{p_2}, ..., c_{p_n}]$.
$W_{behavior}(c) = L_{behavior}(c) + \sum_{i=1}^{n} W_{behavior}(c_{p_i})$
$W_{instances}(c) = L_{instances}(c) + \sum_{i=1}^{t} W_{instances}(c_{b_i})$
$W_{attributes}(c) = L_{attributes}(c) + \sum_{i=1}^{n} W_{attributes}(c_{p_i})$
Each method implements a certain function and has a receiver, former parameters and a result. Further, a method is invoked via a corresponding message of the general form $m(p_1, p_2, ..., p_k)$, where m is the method name, p_1 is the receiver and p_2 to p_k are the parameters.

So, based on what has been defined so far, a class is distinguished by some properties and constructs constituting the class definition. Consequently, consider a general class $\jmath$ to include all such class definitions. Therefore, by definition each object in $L_{instances}(\jmath)$ holds the definition of at least one class c (this will be clear later in this section). Such object is defined to be a quadrate $(C_p(c), C_b(c), L_{attributes}(c), L_{behavior}(c))$. It is obvious that, the definition of class $\jmath$ itself is an object in class $\jmath$.

To sum up, a class c is formally defined as a pair (P_d, P_o); where P_d is an identifier, it is either the o_{id} of an object in class $\jmath$ or the identifier of another existing class, say c'. For the former case, P_o refers to a pair $(L_{instances}(c),\ M_{list}(c))$, where $M_{list}(c)$ denotes a modification list consisting of *modification tuples.* For the latter case, on the other hand, P_o refers to a *view definition*; it is a view directly derived from class c'.

A modification tuple, denoted $M_{tuple}(V_{id})$, is a quadrate $(V_{id}, O_{inserted}, O_{deleted}, O_{updated})$, where V_{id} is an identifier of a view derived from class c; $O_{inserted}$, $O_{deleted}$ and $O_{updated}$ are respectively the sets of objects inserted into, deleted from and updated in $W_{instances}(c)$, after the last update of view V_{id} and before the update of another view based on class c. The latter update causes a new modification tuple to be added to the list of class c.

A view is a triplet $(V_{id}, W_{instances}(V_{id}), P_{expression}(V_{id}))$, where $W_{instances}(V_{id})$ is the set of o_{id}'s for objects from $W_{instances}(c')$ matching the conjunction of predicates in $P_{expression}(V_{id})$. In other words, $P_{expression}(V_{id})$ is a list of predicates the conjunction of which constitute the predicate expression that filters objects from $W_{instances}(c')$ to be included in $W_{instances}(V_{id})$. Further, $P_{expression}(V_{id})$ must be updated to reflect schema changes, i.e., if a schema change drops some of the messages incorporated within some predicates in $P_{expression}(V_{id})$, then those predicates should be adjusted to reflect the change [4]. For a detailed formal definition of predicates see our previous work on query models [3, 5].

This way, we differentiate between two categories of classes according to the instantiation of P_d; *base* classes and *brother* classes. A base class directly points to a class definition in class $\jmath$, i.e., its P_d is the o_{id} of an object in class $\jmath$. A *brother* class holds a view definition and indirectly refers to an object in class $\jmath$ via a base class, i.e., its P_d is the identifier of the base class from which the view is derived. According to this classification, all brother classes in the database are arranged into sets of *equivalent-classes* [***] and hence each group of brother classes sharing the same base class definition constitute an equivalent-class. The cardinality of each equivalent-class is the number of views derived directly from the related base class. On the other hand, the cardinality of the modification list of a class shows the number of views dependent on that class. Further, modification lists hold the update history of the actual database.

To illustrate what has been introduced above, consider the class hierarchy shown in Fig. 2 and the objects shown in Fig. 2. Next to each class in Fig. 2, there are three sets which include $L_{attributes}$, $L_{behavior}$ and $L_{instances}$, respectively. In addition, for every class given in Fig. 2, C_p and C_b are implicitly indicated via class connections. Some example brother classes are enumerated next.

- *Females* is a brother class of *person* with P_d=*person*, $P_{expression}(Females) = [sex(p) = "F"]$ [†], $W_{instances}(Females) = \{o_{id_2}, o_{id_4}\}$
- *BasicCourses* is a brother class of *course* with P_d=*course*,

[***] In the set theory, a set is partitioned into sets of equivalent-classes with each equivalent-class containing related elements; while all the equivalent-classes of a set are pairwise disjoint

[†] Variable p is used as a pointer to objects in the target base class, here *person*

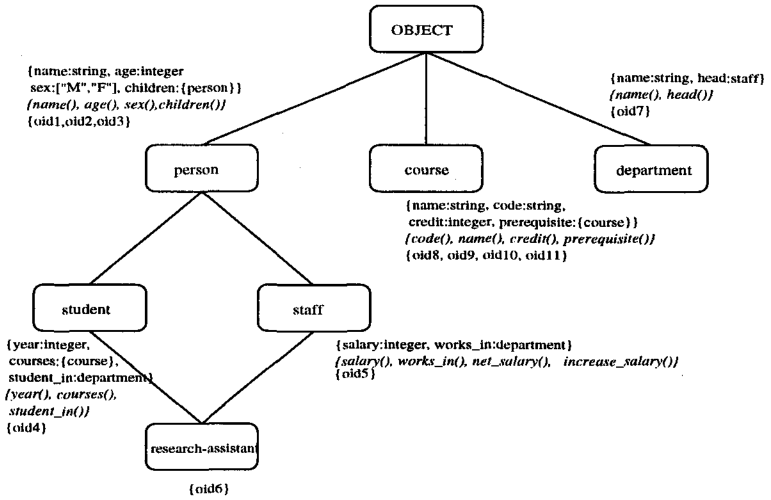
OBJECT
{name:string, age:integer
sex:["M","F"], children:{person}}
(name(), age(), sex(),children())
{oid1,oid2,oid3}
{name:string, head:staff}
(name(), head())
{oid7}
person
course
department
{name:string, code:string,
credit:integer, prerequisite:{course}}
(code(), name(), credit(), prerequisite())
{oid8, oid9, oid10, oid11}
student
staff
{year:integer,
courses:{course},
student_in:department}
(year(), courses(),
student_in())
{oid4}
{salary:integer, works_in:department}
(salary(), works_in(), net_salary(), increase_salary())
{oid5}
research-assistant
{oid6}

Fig. 1. An example class hierarchy

o_{id_1}<"*Jack*", 21, "*M*", ϕ> o_{id_2}<"*Mary*", 42, "*F*", $\{o_{id_1}, o_{id_4}\}$ >
o_{id_3}<"*John*", 65, "*M*", $\{o_{id_5}\}$> o_{id_4}<"*Susan*", 25, "*F*", ϕ, 5, $\{o_{id_9}, o_{id_{10}}\}$, o_{id_7}>
o_{id_5}<"*Smith*", 45, "*M*", $\{o_{id_1}, o_{id_4}\}$, 50*K*, o_{id_7}>
o_{id_6}<"*George*", 22, "*M*", ϕ, 5, $\{o_{id_{11}}\}$, o_{id_7}, 15*K*, o_{id_7}> o_{id_7}<"*CompSci*", o_{id_6}>
o_{id_8}<"*CS*101", "*Int. to Prog.*", 3, ϕ > o_{id_9}<"*CS*211", "*Prog. Lang.*", 3, $\{o_{id_8}\}$>
$o_{id_{10}}$<"*CS*330", "*Data Struc.*", 3, ϕ> $o_{id_{11}}$<"*CS*450", "*Databases*", 3, $\{o_{id_9}, o_{id_{10}}\}$>

Fig. 2. Example objects from the classes given in Figure 1

$P_{expression}(BasicCourses)$= $[prerequisites(p) = \phi]$ $W_{instances}(BasicCourses) = \{o_{id_8}, o_{id_{10}}\}$
- *StaffBrothersOfSusan* is a borther class of *staff* with P_d=*staff*,
$P_{expression}(StaffBrothersOfSusan)$= $[sex(p) = "M"$,
$\exists p_1(p_1 \in W_{instances}(person),\ name(p_1) = "Susan")$,
$\exists p_2(p_2 \in W_{instances}(person),\ \{p, p_1\} \subset children(p_2))]$.
$W_{instances}(StaffBrothersOfSusan) = \phi$

As it is obvious from the examples and detailed more in [4, 5], brother classes serve to hold the result of a query.

3 View Maintenance Algorithms

In this section, we elaborate more on the basic model introduced in Section 2 and describe the algorithms that facilitate incremental view maintenance and materialization.

Any base class, say c, is in general the root of two orthogonal subhierarchies. Consequently, it is not sufficient to reflect into views which are brothers of class c only *local modifications* to objects in $W_{instances}(c)$, *global modifications* should also be considered. An object is locally modified when it is accessed from within class c itself. On the other hand, global modifications against objects in

$W_{instances}(c)$ are performed from within a subclass of class c against objects subsumed in $W_{instances}(c)$ or else from within another class against shared objects along the class-composition hierarchy. Therefore, locating and controlling global modifications is as important as local modifications and also it is required in preserving database consistency and integrity.

$M_{list}(person)=[(Females,\ \phi,\ \phi,\ \phi)]$ $M_{list}(student)=[(Females,\ \phi,\ \{o_{id_4}\},\ \phi)]$
$M_{list}(staff)=[(Females,\ \phi,\ \phi,\ \{o_{id_5}\})]$ $M_{list}(res\text{-}ass)=[(Females,\ \phi,\ \phi,\ \phi)]$
$M_{list}(department)=[]$ $M_{list}(course)=[]$

(a) After the addition of *Females* view

$M_{list}(person)=[(Females,\ \phi,\ \phi,\ \phi)]$ $M_{list}(student)=[(Females,\ \phi,\ \{o_{id_4}\},\ \phi)]$
$M_{list}(staff)=[(Females,\ \phi,\ \phi,\ \{o_{id_5}\})]$ $M_{list}(res\text{-}ass)=[(Females,\ \phi,\ \phi,\ \phi)]$
$M_{list}(department)=[]$ $M_{list}(course)=[(BasicCourses,\ \phi,\ \{o_{id_9}\},\ \{o_{id_{10}}\})]$

(b) After the addition of *BasicCourses* view

$M_{list}(person)=[(Females,\ \phi,\ \phi,\ \phi),(StaffBrothersOfSusan,\ \phi,\ \phi,\ \phi)]$
$M_{list}(student)=[(Females,\ \phi,\ \{o_{id_4}\},\ \phi)]$
$M_{list}(staff)=[(Females,\ \phi,\ \phi,\ \{o_{id_5}\}),(StaffBrothersOfSusan,\ \phi,\ \phi,\ \phi)]$
$M_{list}(res\text{-}ass)=[(Females,\ \phi,\ \phi,\ \phi),(StaffBrothersOfSusan,\ \phi,\ \phi,\ \{o_{id_6}\})]$
$M_{list}(department)=[(StaffBrothersOfSusan,\ \phi,\ \phi,\ \phi)]$
$M_{list}(course)=[(BasicCourses,\ \phi,\ \{o_{id_9}\},\ \{o_{id_{10}}\})]$

(c) After the addition of *StaffBrothersOfSusan* view

Fig. 3. Modification lists of the base classes given in Figure 1

To keep track of global modifications, each addition of a modification tuple to $M_{list}(c)$ triggers the addition of a modification tuple with the same V_{id} to the list of every class along both subhierarchies rooted at class c. Such tuples are utilized to indicate objects to consider from $W_{instances}(c)$ in the process of updating view V_{id}. To illustrate this, shown in Fig. 3 are the modification lists of the base classes given in Fig. 2, after the addition of the example brother classes introduced in Section 2. First, we assume that after defining the *Females* view, object o_{id_4} is deleted from the *student* class and object o_{id_5} is updated in the *staff* class; this is reflected into the modification lists shown in Fig. 3(a). Second, as shown in Fig. 3 3(b) , after the *BasicCourses* view is generated, object o_{id_9} is deleted from the *course* class and object $o_{id_{10}}$ is updated in the *course* class. Finally, in Fig. 3(c), it is shown that after the *StaffBrothersOfSusan* view is generated, object o_{id_6} is updated in the *research-assistant* class.

Modifications along the inheritance hierarchy are located by recursively tracing the C_b's of classes along the inheritance subhierarchy rooted at class c. On the other hand, to successfully reflect modifications along the class-composition subhierarchy, it is necessary to locate modified objects in each particular class along that subhierarchy and to backtrack (mostly by utilizing an index) to locate in $W_{instances}(c)$ objects referencing such modified objects. We accomplish this by introducing two general base classes into the class hierarchy, namely *NEsting* *of Classes* (NEC) and *Complex Objects References* (COR), to keep track of the relationships between classes and objects, respectively.

Explicitly, NEC holds all class-to-class relationships along the class composition hierarchy, i.e., a relationship between two classes c_i and c_j is included in NEC to show that class c_i has an attribute the value of which is drawn from $W_{instances}(c_j)$. When a relationship between two classes c_i and c_j is registered in NEC, the relationships between their corresponding objects is reflected into COR to show that object o_{id_j} from class c_j is contained in the state of object o_{id_i} from class c_i. The definitions in class $\jmath$ for NEC and COR classes are given in Fig. 3. On the other hand, shown in Fig. 3 are the objects contained in NEC and COR classes, as the example classes given in Fig. 2 and the corresponding objects shown in Fig. 2 are concerned. This is achieved because in our query model [3, 5] we allow the specification of the result of a recursive query to be a subset of the transitive closure.

- $C_p(NEC)=\phi$, • $C_b(NEC)=\phi$,
- $L_{attributes}(NEC)=\{LeftClass{:}C,\ RightClass{:}C\}$ [‡]
- $L_{behavior}(NEC)=\{FindClassCompositionHierarchy()\}$

- $C_p(COR)=\phi$, • $C_b(COR)=\phi$,
- $L_{attributes}(COR)=\{LeftObject{:}O_{ID},\ RightObject{:}O_{ID}\}$ [§]
- $L_{behavior}(COR)=\{FindReferencingObjects(TargetClass)\}$

Fig. 4. Definitions of classes NEC and COR in class $\jmath$

$o_{id_{12}}<person,\ person>$ $o_{id_{13}}<student,\ department>$
$o_{id_{14}}<student,\ course>$ $o_{id_{15}}<staff,\ department>$
$o_{id_{16}}<course,\ course>$ $o_{id_{17}}<department,\ staff>$

$o_{id_{18}}<o_{id_2},\ o_{id_1}>$, $o_{id_{19}}<o_{id_2},\ o_{id_4}>$, $o_{id_{20}}<o_{id_3},\ o_{id_5}>$, $o_{id_{21}}<o_{id_4},\ o_{id_9}>$
$o_{id_{22}}<o_{id_4},\ o_{id_{10}}>$, $o_{id_{23}}<o_{id_5},\ o_{id_1}>$, $o_{id_{24}}<o_{id_5},\ o_{id_4}>$, $o_{id_{25}}<o_{id_6},\ o_{id_{11}}>$
$o_{id_{26}}<o_{id_9},\ o_{id_8}>$, $o_{id_{27}}<o_{id_{11}},\ o_{id_9}>$, $o_{id_{28}}<o_{id_{11}},\ o_{id_{10}}>$

Fig. 5. Objects in NEC and COR classes

Objects in the two classes NEC and COR are utilized by the algorithms given next in this section. According to Algorithm 3.2, related to a given view V_{id} which is a brother class of base class c, modification tuples in each class along the class-composition subhierarchy rooted at class c are located by tracing the hierarchy in the forward direction starting with class c and using objects in NEC. After that, COR is utilized to trace in the backward direction every object in the sets included within the located tuples to determine referencing objects in $W_{instances}(c)$. So, given view V_{id}, Algorithm 3.2 determines three sets of o_{id}'s for objects to be utilized in Algorithm 3.3 which updates that view. These sets $Wo_{inserted}$, $Wo_{deleted}$ and $Wo_{updated}$ contain respectively objects which are locally or globally inserted into, deleted from and updated within $W_{instances}(c)$ since the last update of view V_{id}. To have every modified object located, Algorithm 3.1 is utilized to determine modifications along the inheritance subhierarchy rooted at a particular class.

Algorithm 3.1 (*IMT*)
/ This algorithm considers only classes along the inheritance subhierarchy rooted at class c. The target is to determine modifications to objects in $W_{instances}(c)$, i.e., objects inserted into, deleted from or updated within $W_{instances}(c)$ since the last update to view V_{id} which directly or indirectly depends on base class c.*
Input: *a class c and a view V_{id}.*
Output: *sets of o_{id}'s $Co_{inserted}$, $Co_{deleted}$, $Co_{updated}$ of modified objects in $W_{instances}(c)$.*
Steps:

```
    Let $Co_{inserted}=Co_{deleted}=Co_{updated}=\phi$
    Let $C_{inheritance}=[c]$
/* $C_{inheritance}$ is a list to include all classes found along the inheritance subhierarchy
rooted at class c.
    Let $i=0$
    While not end of $C_{inheritance}$ do
        Let $c'=C_{inheritance}[i]$
        $C_{inheritance}= C_{inheritance} + C_b(c')$
        Find $M_{tuple}(V_{id})$ within $M_{list}(c')$.
        While not end of $M_{list}(c')$ do
                . $Co_{inserted}=Co_{inserted} \bigcup O_{inserted}$
                . $Co_{deleted}=Co_{deleted} \bigcup O_{deleted}$
                . $Co_{updated}=Co_{updated} \bigcup O_{updated}$
        EndWhile
        If there exists an immediate predecessor
          of $M_{tuple}(V_{id})$ in $M_{list}(c')$ then
                . Merge $M_{tuple}(V_{id})$ with its
                  immediate predecessor
        Else
                . Delete $M_{tuple}(V_{id})$
        EndIf
        Append ($V_{id}$, $\phi$ $\phi$ $\phi$) at the rear of $M_{list}(c')$
        $i=i+1$
    EndWhile
EndAlgorithm                                                    □
```

Algorithm 3.2 (*IDU*)
/ To determine since the last update to view V_{id}, the sets of objects inserted into, deleted from or updated within $W_{instances}(c)$, where c is the base class c of which view V_{id} is a brother class.*
Input: *V_{id} of a view which is a brother class of base class c.*
Output: *Sets of objects $Wo_{inserted}$, $Wo_{deleted}$,*

```
            $Wo_{updated}$ to be utilized in Algorithm 3.3 for
            incremental update of view $V_{id}$
```

Steps:

```
  Let $Wo_{inserted} = Wo_{deleted} = Wo_{updated} = \phi$
  Find $M_{tuple}(V_{id})$ within $M_{list}(c)$.
  If $M_{tuple}(V_{id})$ is not found then
/* derive view $V_{id}$ from scratch; it is a new view
          $Wo_{inserted} = W_{instances}(c)$
          $Wo_{deleted} = Wo_{updated} = \phi$
  Else
          Perform IMT($V_{id}$, c)
          $Wo_{inserted} = Co_{inserted}$
          $Wo_{deleted} = Co_{deleted}$
          $Wo_{updated} = Co_{updated}$
          $C_{cch} = FindClassCompositionHierarchy(c)$ /* $C_{cch}$ is a list to include classes
within the class-composition subhierarchy rooted at class c.
          Let $G = \phi$
/* G is a set to include all modified objects along the class-composition subhierarchy
rooted at class c.
          For every class $c_i$ in $C_{cch}$ do
             . Perform IMT($V_{id}$, $c_i$)
             . $G = G \bigcup Co_{inserted} \bigcup Co_{deleted} \bigcup Co_{updated}$          EndFor
          For every object $o_{id}$ in G do
             . $Wo_{updated} = Wo_{updated} +$
                          $FindReferencingObjects(o_{id}, c)$
          EndFor
          $Wo_{inserted} = Wo_{inserted} - Wo_{deleted}$
          $Wo_{updated} = Wo_{updated} - Wo_{inserted}$
          $Wo_{updated} = Wo_{updated} - Wo_{deleted}$
  EndIf
EndAlgorithm                                                        □
```

Algorithm 3.3 (*ViewUpdate*)
/ To utilize modified objects within $W_{instances}(c)$ in deriving (updating) view V_{id} which is a brother class of base class c.*

Input: V_{id} *of a view which is a brother class of*
base class c.

Output: $W_{instances}(V_{id})$

Steps:

```
  Perform $IDU(V_{id})$
  $W_{instances}(V_{id}) = W_{instances}(V_{id}) - Wo_{deleted}$
  $W_{instances}(V_{id}) = W_{instances}(V_{id}) - Wo_{updated}$
  $Wo_{inserted} = Wo_{inserted} \bigcup Wo_{updated}$
  For every object $o_{id}$ in $Wo_{inserted}$ do
     If $o_{id}$ satisfies $P_{expression}(V_{id})$ then
        $W_{instances}(V_{id}) = W_{instances}(V_{id}) \bigcup \{o_{id}\}$
     EndIf
  Endfor
EndAlgorithm                                                        □
```

$M_{list}(person) = [(StaffBrothersOfSusan,\ \phi,\ \phi,\ \phi),\ (Females,\ \phi,\ \phi,\ \phi)]$
$M_{list}(student) = [(Females,\ \phi,\ \phi,\ \phi)]$
$M_{list}(staff) = [(StaffBrothersOfSusan,\ \phi,\ \phi,\ \phi),\ (Females,\ \phi,\ \phi,\ \phi)]$
$M_{list}(res\text{-}ass)=[(StaffBrothersOfSusan\phi,\phi,\{o_{id_6}\}),\ (Females,\ \phi,\ \phi,\ \phi)]$
$M_{list}(department) = [(StaffBrothersOfSusan,\ \phi,\ \phi,\ \phi)]$
$M_{list}(course) = [(BasicCourses,\ \phi,\ \{o_{id_9}\},\ \{o_{id_{10}}\})]$

Fig. 6. Modification lists of the base classes given in Figure 1 after the update of *Females* view

$M_{list}(person) = [(Females,\ \phi,\ \phi,\ \phi),(StaffBrothersOfSusan,\ \phi,\ \phi,\ \phi)]$
$M_{list}(student) = [(Females,\ \phi,\ \{o_{id_4}\},\ \phi)]$
$M_{list}(staff) = [(Females,\ \phi,\ \phi,\ \{o_{id_5}\}),(StaffBrothersOfSusan,\ \phi,\ \phi,\ \phi)]$
$M_{list}(res\text{-}ass) = [(Females,\ \phi,\ \phi,\ \{o_{id_6}\}),(StaffBrothersOfSusan,\ \phi,\ \phi,\ \phi)]$
$M_{list}(department) = [(StaffBrothersOfSusan,\ \phi,\ \phi,\ \phi)]$
$M_{list}(course) = [(BasicCourses,\ \phi,\ \{o_{id_9}\},\ \{o_{id_{10}}\})]$

(a) Before the update of *Females* view

$M_{list}(person) = [(Females,\ \phi,\ \phi,\ \phi),(StaffBrothersOfSusan,\ \phi,\ \phi,\ \phi)]$
$M_{list}(student) = [(Females,\ \phi,\ \phi,\ \phi)]$
$M_{list}(staff) = [(Females,\ \phi,\ \phi,\ \phi),(StaffBrothersOfSusan,\ \phi,\ \phi,\ \phi)]$
$M_{list}(res\text{-}ass) = [(Females,\ \phi,\ \phi,\ \phi),(StaffBrothersOfSusan,\ \phi,\ \phi,\ \phi)]$
$M_{list}(department) = [(StaffBrothersOfSusan,\ \phi,\ \phi,\ \phi)]$
$M_{list}(course) = [(BasicCourses,\ \phi,\ \{o_{id_9}\},\ \{o_{id_{10}}\})]$

(b) After the update of *Females* view

Fig. 7. Modification lists of the base classes given in Figure 1 after the update of *StaffBrothersOfSusan* view

To illustrate the already introduced algorithms, assume that it is desired to access the *Females* view. However, accessing a view results in updating its objects because we employ deferred update. Consequently, Algorithm 3.3 is executed and hence Algorithms 3.2 and 3.1, since Algorithm 3.3 calls Algorithm 3.2 which, in its turn, calls Algorithm 3.1. So, on executing $ViewUpdate(Females)$, the related modification lists shown in Fig. 3(c) are traced by Algorithms 3.2 and 3.1 to locate modified objects in $W_{instances}(person)$; because *person* is the base class of which *Females* is a brother class. The following modification sets are returned by Algorithms 3.2:
$W_{o_{inserted}}=\phi$, $W_{o_{deleted}}=\{o_{id_4}\}$, $W_{o_{updated}}=\{o_{id_5}, o_{id_6}\}$. Algorithms 3.3 utilizes these modification sets to return $W_{instances}(Females) = \{o_{id_2}\}$. To mark the starting point of the forthcoming update of the *Females* view, Algorithm 3.1 updates the utilized modification lists from Fig. 3(c) into the lists shown in Fig. 3. To realize the change in the lists more explicitly, consider the execution of *ViewUpdate(StaffBrothersOfSusan)* by utilizing the lists in Fig. 3(c) and Fig. 3, i.e., without executing *ViewUpdate(Females)* and after executing *ViewUpdate(Females)*, respectively. Modification lists of the base classes after

such executions are shown in Fig. 3(a) and (b), respectively. Notice how in Fig. 3(a), o_{id_6} moved within $M_{list}(\textit{research-assistant})$.

4 Conclusions

In this paper, we presented a model that facilitates incremental view maintenance within object-oriented databases. Instant reflection of class updates into dependent views is not only time consuming, but also proved to be useless and hence time wasting. View maintenance within the object-oriented context is more challenging than that within the relational model. Thus, our algorithms serve more than the requirements of the relational model. Explicitly speaking, Algorithms 3.1, 3.2 and 3.3 are applicable for both the nested relational model and the relational model as well.

References

1. Abiteboul, S., Bonner, A.: Objects and Views. Proceedings of the ACM-SIGMOD International Conference on Management of Data (1991)
2. Alashqur, A., Su, S.Y., Lam, H.: OQL: A Query Language for Manipulating Object-Oriented Databases. Proceedings of the 15^{th} International Conference on Very Large Databases. Amsterdam (August 1989)
3. Alhajj, R., Arkun, M.E.: A Query Model for Object-Oriented Database Systems. Proceedings of the 9^{th} IEEE International Conference on Data Engineering. Vienna (April 1993)
4. Alhajj, R., Polat, F.: An Object-Oriented Query Model Enforcing Closure and Reusability. Journal of Mahtematical Modeling and Computing 6 (April 1996)
5. Alhajj, R., Polat, F.: Closure Maintenance in an Object-Oriented Query Model. Proceedings of the ACM International Conference on Information and Knowledge Management. Maryland (November 1994)
6. Dayal, U.: Queries and Views in an Object-Oriented Data Model. Proceedings of the 2^{nd} International Workshop on Database Programming Languages (June 1989)
7. Gupta, A., Mumick, I., Subrahmanian, V.: Maintaining Views Incrementally. Proceedings of the ACM-SIGMOD International Conference on Management of Data. Washington D.C. (1993)
8. Hanson, E.N.: A Performance Analysis of View Materialization Strategies. Proceedings of the ACM-SIGMOD International Conference on Management of Data (1987)
9. Heiler, S., Zdonik, S.B.: Object Views: Extending the vision. Proceedings of the 6^{th} IEEE International Conference on Data Engineering. Los Algeles (February 1990)
10. Kifer, M., Kim, W., Sagiv, Y.: Querying Object-Oriented Databases. Proceedings of ACM-SIGMOD International Conference on Management of Data. San Diego CA (June 1992)
11. Rundensteiner, E.A.: A Methodology for Supporting Multiple Views in Object-Oriented Databases. Proceedings of the 18^{th} International Conference on Very Large Databases. Vancouver BC (August 1992)