ELSEVIER

Data & Knowledge Engineering 29 (1999) 121–145

DATA & KNOWLEDGE ENGINEERING

Incremental materialization of object-oriented views

Reda Alhajj[a,*], Ashraf Elnagar[b]

[a]*Department of Computer Science, Sultan Qaboos University, P.O. Box 36, Alkhod 123, Muscat, Oman*
[b]*Department of Computer Science, University of Sharjah, P.O. Box 27272, Sharjah, UAE*

Received 27 March 1998; revised 14 July 1998; accepted 2 September 1998

Abstract

We present an approach to handle incremental materialization of object-oriented views. Queries that define views are implemented as methods that are invoked to compute corresponding views. To avoid computation from scratch each time a view is accessed, we introduce some deferred update algorithms that reflect for a view only related modifications introduced into the database while that view was inactive. A view is updated by considering modifications performed within all classes along the inheritance and class-composition subhierarchies rooted at every class used in deriving that view. To each class, we add a *modification list* to keep one *modification tuple* per view dependent on that class. Such a tuple acts as a reference point that marks the start of the next update to the corresponding view. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Algorithms; Deferred update; Incremental update; Materialized views; Object-oriented databases

1. Introduction

In general, a view is a query that may be temporary or required to be persistent. Temporary views are generated and used during a query session and cease to exist at the end of that session; they never become a part of the class hierarchy. Persistent views, however, must be placed in the hierarchy for subsequent access.

Views have a lot of application areas ranging from integrity constraint maintenance to persistent queries, among others. Another emerging application area that draws much attention to views is *data warehouses* [36] where it is necessary to reduce communication. Views may also be used to enforce security such that a view for a certain level of users can be defined so as not to include confidential information contained within the actual database. Consequently, research on views must receive more attention and researchers in the field must concentrate on adapting view maintenance models to emerging advanced data models. In this respect, any proposed view model must consider the possibility of incremental update because recomputing the whole view from scratch may be very

* Corresponding author. E-mail: alhajj@squ.edu.om

0169-023X/99/$ – see front matter
PII: S0169-023X(98)00042-1

expensive. However, the difficulty and complexity of a view maintenance approach is dependent on the underlying data model. Although a lot of research effort is concentrating on view maintenance and materialization within the relational model, e.g. [11,13,15,19,20,31,32], much more effort and consideration is still required when dealing with view maintenance within the nested relational model and more important, object-oriented data models. In other words, views have not been studied thoroughly within the object-oriented context, despite the effort done in this respect, e.g. [1,4,12,14,18,21,22,23,25,26,29,30,33]; this is due to the still lacking agreement on standardization in the field.

Complex structures (nesting) commonality between object-oriented and the nested relational models lead to having an object-oriented view maintenance approach applicable to the nested relational model and hence to the conventional relational model. However, the inheritance and hence reusability distinguishing feature of object-oriented models makes view maintenance within the object-oriented context unique, i.e., no approach developed for the relational model remains applicable to the object-oriented level. Adapting a relational view model to an object-oriented counterpart is unacceptable because such an approach eliminates the functionality and the distinguishing features of the object-oriented context; it is very similar to having a calculator while continuing to work with manual calculations. Actually, object-oriented models have features and characteristics that make views within such models more flexible and natural than their relational counterparts. In other words, it is necessary to benefit from the class concept and define a view to be a class with the corresponding query being a method in that class to decide on its objects. This way, closure is maintained within the model where a view can be smoothly used as an operand. We dealt with stored queries in our query model described in [5,7,9], and managed to handle the proper placement of a query result in the hierarchy. For instance, the result of *project* is a superclass of the class of the operand, the result of *join* is a subclass of the classes of both operands, while the result of *select* is a brother class of the class of the operand. Other operations are treated the same way.

In this paper, we present a model that facilitates incremental view materialization within object-oriented databases and is also applicable to the nested and relational models. The model is not restricted to handle only queries based on a single class (including selection and projection); multiple classes based queries are also considered. In other words, a view may be derived from one or more classes and each such class may be either an existing view or a class from the actual database. To have a view accepted as a first class citizen in the database, we categorize classes into *base* classes and *virtual* classes[1] in common with others [27,28]. The latter correspond to view definitions and hence hold the virtual part of the database. The former, on the other hand, hold the actual database. The basic idea in our approach is to distinguish and keep modifications related to a view since its last derivation (update). 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. However, a base class is in general the root of two subhierarchies, namely, inheritance and class-composition subhierarchies. Consequently, while updating a view, it is necessary to consider indirect modifications performed within any class along both subhierarchies rooted at every class used in deriving that view. To achieve that, to each base or virtual class we add a *modification list* to keep track of all modifications to be reflected on demand to views which depend on that class. A modification list in a class consists of a sequence of *modification tuples*, one per view dependent on that class. Each such

[1] In this paper, the two terms 'views' and 'virtual classes' are used interchangeably to refer to the same entity.

tuple acts as a reference point to help in deciding on the amount of information necessary to update the corresponding view. A modification tuple holds a view identifier and modification information related to the corresponding class. In this paper we consider only modification information related to objects. However, our approach can be easily extended to consider modifications related to behavior as well as to class variables, if necessary. This could be achieved by merely extending the modification tuple to include the desired information.

Explicitly, modification information inside tuples of base classes covers insertions, deletions and updates related to objects in those classes. On the other hand, modification tuples of virtual classes do not contain update information because updates are solely performed against base classes in our model. Furthermore, modifications performed on a class are always registered inside the last tuple in its modification list and a modification list is ordered such that its tail is the tuple holding the identifier of the most recently accessed view. On accessing a view, it is required to consider the union of modifications not known to that view because they have been made after its last update. Such modifications are present in the sequence of tuples starting with the one holding the identifier of the given view until the end of a modification list because the order of modification tuples inside a modification list reflects the access order of their corresponding views. The tuple at the head of a modification list corresponds to the least recently accessed view. Therefore, on accessing that view, it is necessary to consider all modifications present inside the list, i.e., taking the union of the modifications in all the tuples within the list. On the other hand, the tail tuple of a modification list corresponds to the most recently accessed view and only modifications found in that tuple need to be reflected to that view on update. As a view is updated, the modification sets in the tuple holding the identifier of that view are merged with the corresponding sets of its immediate predecessor tuple in the same list. Then, a new tuple holding the same view identifier with empty modification sets is appended at the tail of that list; to mark the starting point of the forthcoming update to that view. Finally, different algorithms have been developed to reflect modifications to virtual classes as they are accessed.

The rest of this paper is organized as follows. Section 2 includes a short description of the work already done to deal with view maintenance. The model on which our work is based is presented in Section 3. In Section 4, we elaborate more on the characteristics of the model and introduce the algorithms that govern and guarantee incremental view maintenance and materialization. Section 5 concludes.

2. Related work

Views have been studied within different contexts. Some researchers handled the problem of answering queries using materialized views, e.g. [16]. A systematic study of the complexity of the problem of answering queries using materialized views within the relational model is proposed in [2]. Materialized views have also been studied within the temporal relational model. For instance, the work described in [34] introduced a framework for maintaining temporal views over non-temporal information sources in a data warehousing environment. Finally, incremental materialization techniques over semistructured data have received much attention in recent years, e.g. [3]. The graph-based model OEM and the query language Lorel, developed at Stanford, form the framework for the work described in [3]. The authors proposed an algorithm that produces a set of queries that

compute changes to a view based upon changes to the source. Graph structured views and their incremental maintenance are also investigated in [37].

Many incremental view maintenance algorithms have been developed within the relational model, e.g. [11,13,15,19,20,31]. Most of them are designed for a traditional centralized database environment, where it is assumed that view maintenance is performed by a system that has full control and knowledge of the view definition, the base relations and the updated tuples. These algorithms differ somewhat in the view definitions they handle. While some algorithms depend on key information to deal with duplicate tuples [13], others use a counting approach [19]. POSTGRES [32] uses the rule system to simulate views, where derived tables (views) can be defined by rules and other rules may define specialized update semantics for such views. Another group who studied materialized views in distributed systems based their work on timestamping the updated tuples [31], and their algorithms assume that there is only one base table. In our model, we follow a similar approach by ordering views from the least recently accessed to the most recently accessed. However, we do not impose the restriction of a single class.

Some systems within the object-oriented context, such as ORION [24], do not support the use of views, while others regard views as one more way of querying the database without dealing with incremental view maintenance and materialization. For instance, views in OQL are known as contexts [4]. An OQL query contains a context clause, wherein the user specifies a series of class associations to derive a desired subdatabase. A query is then specified over the newly defined context. Because this context is actually a new virtual database, the context clause subsumes the need for views. Any time a view is needed, a context may be specified. On the other hand, in XSQL [23], views may be defined explicitly. Queries may also be used to define views, because the result of a query is a set of objects, which is in turn, a view. We follow the same approach of XSQL in preserving the correspondence between objects in views and objects in database classes and hence allow for easy and consistent updates where view updates are translated to database updates via object identifiers.

The view mechanism defined for the O_2 system [1,29] facilitates the access to values in views as if they are stored in those views. A view has a hierarchy, but no data of its own. Such hierarchy is intended to be closer to the needs of the user than the original class hierarchy from which it was extracted. To ensure consistency, an object is defined as being real in only one class, and also real in only one view. This allows update resolution to be implemented in a consistent way. The CROQUE approach to the maintenance of materialized views is discussed in [18], where views are treated as functions of database objects. The algebraic properties of those functions were examined to derive incremental update plans. The work described in [30] is based on an object functional model that has been developed as an evolution from and generalization of the nested relational model. Class predicates distinguish that model from others. Classes may be constrained by predicates that must be satisfied by all members of the class. Class predicates are also used to handle updates to views. In our model, we have a method in each class to decide on objects that qualify to be in that class and hence our approach is more general. Another approach is FUGUE [22] that uses type hierarchies for information hiding: the user can implement a new type for a view that utilizes some base type(s) and offers only a restricted functionality or extends the functionality. Also, not all instances of the base type may be exported. Described in [21] is a view materialization model in which updates are propagated using change files that represent histories of design sessions. However, objects are duplicated for virtual classes rather than merely storing references to objects. The work described in [25] addresses maintaining consistency for a particular type of join class formed along an existing

path in the aggregation graph. Finally, MultiView [26,27,28] is a view model implemented on top of GemStone and it incorporates algorithms for object preserving view classes. Furthermore, it focuses on the specification of a consistent view schema graph rather than an individual view class. Our approach described in [5,7,8,9] is a step in this direction where we handle the proper placement of a query result in order to maintain closure and maximize reusability.

As a result, through the use of object identifiers and virtual data, the view mechanism can be implemented correctly. Views can be entirely comprised of pointers information, i.e., virtual data. 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 and hence no inconsistencies emerge due to the creation or updating of views. However, it is important to decide on the appropriate time to reflect database modifications to related views. There are three general approaches to the timing of view maintenance: immediate update that updates the view after each update to the actual database; deferred update that updates the view only when a query is issued against the view; and periodic update that updates the view at periodic intervals. Performance studies on these strategies within the relational context determined that the efficiency of an approach depends heavily on the structure of the base relations and on update patterns [20]. The immediate update mode has the disadvantage that each update transaction incurs the overhead of updating the view. The overhead increases with the number of views and their complexity. Immediate update is not even possible in some applications such as data warehousing. If a component database does not know what views exist at the warehouse, it cannot modify transactions updating base classes so that they also refresh materialized views. Even if the system was a centralized one, it may be necessary to minimize the per-transaction overhead imposed by view maintenance. In such cases deferred maintenance is most appropriate. Some applications may be tolerant to out-of-date data, or even require that the view be frozen for analysis and other functions. In this case the view could be refreshed periodically or just before querying. Deferred maintenance makes it possible that several updates can be batched together.

We argue that deferred update is more reasonable within the object-oriented context. Explicitly, the conventional relational model is based on a simple data structure, that is the table, while an object-oriented model contains complex structures, including directed graphs and trees. Based on that, modifying a relation does not affect except the relation itself and its dependent views. On the other hand, modifications done against any node within a tree or a graph do not affect only that node and its dependent views, but also propagate to affect predecessor nodes and their related views. Here applying immediate update mode imposes on the system visiting each affected node to tell it and its dependent views about the performed modification. Doing this, other jobs will be frustrated waiting while the system is busy spending effort on telling some information not asked about. To avoid such a situation, we employ deferred update mode and argue that it is the most suitable mode for object-oriented systems where a node is told about modifications only when it is active willing to have up-to-date knowledge.

3. The basic model

In this section we introduce our model. An example class hierarchy is shown in Fig. 1 and some corresponding objects are included in Fig. 2. Both figures will be referenced frequently in the paper where illustrating examples are necessary. In Section 3.1, we concentrate on the terminology and

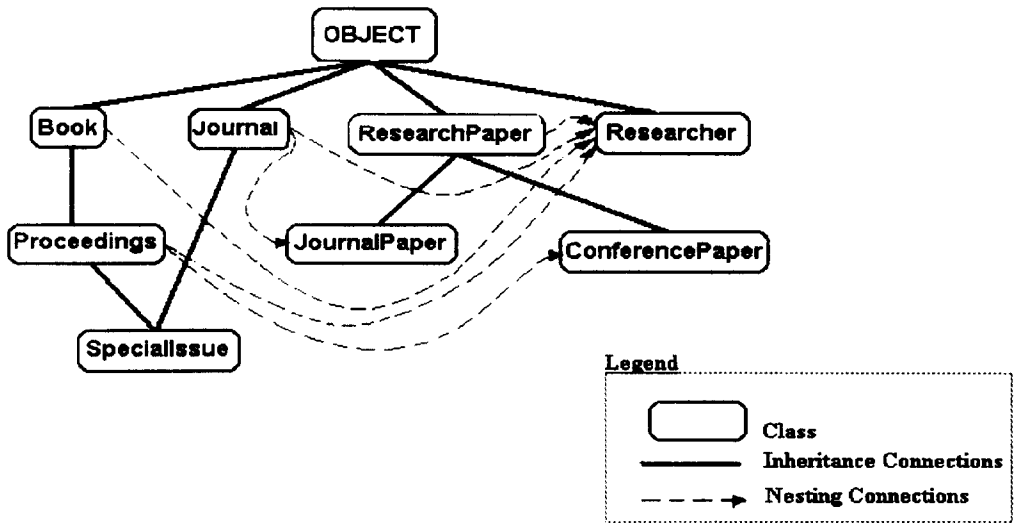


Fig. 1. An example class hierarchy.

definitions necessary for understanding the incremental view materialization approach described in Section 4. Some illustrating examples are given in Section 3.2.

3.1. Basic terminology and definitions

Any object with *oid* as *object identifier* qualifies to be considered in the set of objects of class c, denoted by $L_{instances}(c)$, if and only if object *oid* understands nothing more than the behavior defined for objects of class c, i.e., the set of methods/messages of class c. The behavior for class c consists of two parts, *inherited behavior* and *locally defined behavior*. However, enforcing polymorphism and overriding relaxes the model to allow the re-definition of any method in the local behavior instead of

o_{id_1}["Views in Databases",[o_{id_7}, o_{id_8}], 1997,"Computer Science","MEH Pub"]
o_{id_2}["Algorithms",[o_{id_9}], 1996,"Computer Science","BET Pub"]
o_{id_3}["EXT Information Engineering",[o_{id_8}, o_{id_9}], 1994,"Computer Science", Nil,"Vienna", $o_{id_{10}}$, {o_{id_7}, o_{id_9}}, 0.25,[o_{id_6}, $o_{id_{11}}$]]
o_{id_4}["ICT Transactions on Object Databases", o_{id_9}, {o_{id_7}, o_{id_8}},"MEH Pub",[$o_{id_{12}}$]]
o_{id_5}["Multiagent Systems",[$o_{id_{10}}$],"Artificial Intelligence",{"Agents","Cooperation"}, 10-10-95, 5-12-97, 2-1-98, 23, 4]
o_{id_6}["Feature based retrieval of Similar Shapes",[o_{id_7}, o_{id_9}],"Database",{}]
o_{id_7}["Mary Folk","F"]
o_{id_8}["Bill Richardson","M"]
o_{id_9}["Reanna Sedgewick","F"]
$o_{id_{10}}$["Susan Johnson","F"]
$o_{id_{11}}$["A Query Model for object Oriented Databases",[o_{id_8}, $o_{id_{10}}$],"Database",{"Object Algebra", "Closure"}]
$o_{id_{12}}$["A View Management Model",[o_{id_7}, $o_{id_{10}}$],"Database",{"Views", "Query Model"}, 10-10-90, 11-11-91, 12-12-91, 21, 3]
$o_{id_{13}}$["Object Engineering",[o_{id_7}, o_{id_9}], 1996,"Computer Science","BET Pub","New York", $o_{id_{10}}$, {o_{id_7}, o_{id_9}}, 0.35,[$o_{id_{14}}$]]
$o_{id_{14}}$["Version based Information Management",[o_{id_7}, o_{id_8}],"Database",{}]

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

inheriting it and this leads to a method having different implementations in different classes [8]. Each method implements a certain function and has a receiver, formal parameters and a result. Further, a method is invoked via a corresponding message of the general form $m(r, p_1, p_2, \ldots, p_k)$, where m is the method name, r is the receiver and p_1 to p_k are the parameters.

Definition 3.1 (Class Behavior). *Given a class c and let* $C_p(c) = [c_{p1}, c_{p2}, \ldots, c_{p_n}]$[2] *be a list of its direct superclasses. The whole behavior for class c, denoted by* $W_{behavior}(c)$, *is recursively defined to include the whole behavior of the classes in* $C_p(c)$. *Formally,*

$W_{behavior}(c) = L_{behavior}(c) = \cup_{i=1}^{n} W_{behavior}(c_{p_i})$

where $L_{behavior}(c)$ *denotes the local behavior for class c.*

Returning back to objects of class c, they are not limited to objects in $L_{instances}(c)$. There are some other objects, i.e., objects in the extent of class c that, due to inheritance, understand the behavior in $W_{behavior}(c)$. In other words, it is possible to access objects in the subclasses of class c as if they were defined in $L_{instances}(c)$.

Definition 3.2 (Class Extent). *Given a class c and let* $C_b(c) = \{c_{b_1}, c_{b_2}, \ldots, c_{b_t}\}$[3] *be the set of its direct subclasses. All objects that understand at least the behavior in* $W_{behavior}(c)$, *constitute the extent of class c, denoted by* $W_{instances}(c)$. *This set is recursively defined in terms of the extents of the classes in* $C_b(c)$. *Formally,*

$W_{instances}(c) = L_{instances}(c) \cup_{i=1}^{t} W_{instances}(c_{b_i})$

Of course, by enforcing polymorphism and overriding an object *oid* that qualifies to be in both $W_{instances}(c_i)$ and $W_{instances}(c_j)$ may respond differently to the same message, depending on whether *oid* has been accessed from within $W_{instances}(c_i)$ or $W_{instances}(c_j)$. This is true because, having a message m in both $W_{behavior}(c_i)$ and $W_{behavior}(c_j)$ does not restrict the method underlying m to have the same implementation in both c_i and c_j. However, for an object to be able to understand and deal with a certain behavior, it should have some predefined basic knowledge, i.e., possess a certain state. Such a state is specified by the values of a set of predefined instance variables (attributes). Consequently, a part of the behavior understandable by an object is devoted to deal with its state. The other part of the behavior is there to derive new values based on the present state (stored values). The former part includes two methods corresponding to each particular attribute, with the same message being used to invoke both methods but with different parameters. The first method, with a single parameter, sets the value of an attribute to the value specified by the parameter. The second method has no parameter and simply returns the current stored value of an attribute. For instance, given *title* as an attribute in Fig. 3, two messages *title*() and *title*(*t*) are there to retrieve the title of a receiving object and to set the title of a receiving object to the value *t*, respectively.

Definition 3.3 (Instance Variables). *Given a class c; the set of attributes that determine the state of*

[2] A list notation is used for the superclasses because their order is important for conflict resolution due to polymorphism and overriding. Conflicts are resolved according to certain predefined rules discussed in [6].

[3] Conflict resolution is not applicable here because only objects are concerned, hence the set notation is utilized.

NewClass: *Book*
- $L_{attributes}(Book)$={title:String, author:[Researcher], year:Date, subject:String, publisher:String},
- $L_{behavior}(Book)$={title(), title(t), author(), author(p), year(), year(y), subject(), subject(s), publisher(), publisher(p)},
- $C_p(Book)$=[]
- $C_b(Book)$={Proceedings}

NewClass: *Proceedings*
- $L_{attributes}(Proceedings)$ = {location:String, chairperson:Researcher, committee:{Researcher}, AcceptanceRate:Real, contents:[ConferencePaper]}
- $L_{behavior}(Proceedings)$ = {location(), location(l), chairperson(), chairperson(p), committee(), committee(p),
- $C_p(Proceedings)$=[Book]
- $C_b(Proceedings)$={SpecialIssue}

NewClass: *Journal*
- $L_{attributes}(Journal)$ = {title:String, EditorInChief:Researcher, EditorialBoard:{Researcher}, publisher:String, contents:[JournalPaper]},
- $L_{behavior}(Journal)$ = {title(), title(t), EditorInChief(), EditorInChief(e), EditorialBoard(), EditorialBoard(p), publisher(), publisher(p), contents(), contents(c)},
- $C_p(Journal)$=[]
- $C_b(Journal)$ = {SpecialIssue}

NewClass: SpecialIssue
- $L_{attributes}(SpecialIssue)$ = {Volume:Integer, Number:Integer}
- $L_{behavior}(SpecialIssue)$ = {Volume(), Volume(v), Number(), Number(n)}
- $C_p(SpecialIssue)$=[Proceedings, Journal]
- $C_b(SpecialIssue)$={}

NewClass: ResearchPaper
- $L_{attributes}(ResearchPaper)$ = {title:String, author:[Researcher], area:String, KeyWords:{String}},
- $L_{behavior}(ResearchPaper)$ = {title(), title(), author(), author(a), area(), area(a), KeyWords(), KeyWords(k)},
- $C_p(ResearchPaper)$=[]
- $C_b(ResearchPaper)$={JournalPaper, ConferencePaper}

NewClass: JournalPaper
- $L_{attributes}(JournalPaper)$ = {DateReceived:Date, DateRevised:Date, DateAccepted:Date, Volume:Integer, Number:Integer}
- $L_{behavior}(JournalPaper)$ = {DateReceived(), DateReceived(d), DateRevised(), DateRevised(d), DateAccepted(), DateAccepted(d), Volume(), Volume(v), Number(), Number(n)}
- $C_p(JournalPaper)$=[ResearchPaper]
- $C_b(JournalPaper)$={}

NewClass: ConferencePaper
- $L_{attributes}(ConferencePaper)$ = {}
- $L_{behavior}(ConferencePaper)$ = {}
- $C_p(ConferencePaper)$=[ResearchPaper]
- $C_b(ConferencePaper)$={}

NewClass: Researcher
- $L_{attributes}(Researcher)$ = {name:String, sex:Character}
- $L_{behavior}(Researcher)$ = {Name(), Name(s), Sex(), Sex(s)}
- $C_p(Researcher)$=[]
- $C_b(Researcher)$={}

Fig. 3. Properties of the example classes shown in Fig. 1.

each object in $L_{instances}(c)$ is denoted by $W_{attributes}(c)$ and defined recursively in terms of the attributes defined for objects of the classes in $C_p(c)$. Formally,

$$W_{attributes}(c) = L_{attributes}(c) \cup_{i=1}^{n} W_{attributes}(c_{p_i}),$$

where $L_{attributes}(c)$ denotes the additional attributes defined locally in class c.

Each attribute has a domain from which objects draw the values constituting their states. For instance, the domain of the attribute *title* is *String*. A domain may also be a set of objects from an existing class as it is the case with the attribute *author* in Fig. 3 that has as its domain a set of objects from *Researcher* class.

Based on what has been introduced so far, a class is distinguished by some properties and constructs (Definition 3.4) that form one part of its definition; the other part will be introduced next in Definition 3.5.

Definition 3.4 (Class Properties). *Let δ be a general class; an object in $L_{instances}(\delta)$ is defined to be a quadruple $(C_p(c), C_b(c), L_{attributes}(c), L_{behavior}(c))$, where $C_p(c)$, $C_b(c)$, $L_{attributes}(c)$, and $L_{behavior}(c)$ are the properties of a given class c present in the class hierarchy.*

It is obvious that, the definition of class δ itself is an object in class δ with as values:

- $C_p(\delta) = [\,]$,
- $C_b(\delta) = \phi$,
- $L_{attributes}(\delta) = \{C_p(c):[C],\ C_b(c):\{C\},\ L_{attributes}(c):\{I_V\},\ L_{behavior}(c):\{M_T\}$[4],
- $L_{behavior}(\delta) = \{AppendC_p(class,position),\ DropC_p(class),\ AddC_b(class),\ DropC_b(class)$
  $AddL_{attributes}(attribute,domain),\ DropL_{attributes}(attribute),$
  $AddL_{behavior}(message,function),\ DropL_{behavior}(message)\}$

Definition 3.5 (Class). *A class c is formally defined as a pair $(P_d(c), P_o(c))$, such that:*

- *$P_d(c)$ refers to a set of identifiers drawn from a domain that includes either objects in class δ or class identifiers of base and virtual classes. In other words, the value of $P_d(c)$ may be either a single object from class δ or a mixture of class identifiers and/or view identifiers.*
- *The value of $P_o(c)$ depends on the value of $P_d(c)$ as follows. For the former case, c is a base class and $P_o(c)$ 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, c is a virtual class and $P_o(c)$ refers to its additional properties; it is the view directly derived from classes/views the identifiers of which are included in $P_d(c)$.*

Definition 3.6 (View Dependency). *Given a class c and a view V_{id}. We say that view V_{id} depends on class c if and only if either one of the following is satisfied:*

(1) $c \in P_d(V_{id})$,
(2) $\exists c' \in P_d(V_{id})$ *and c is in the inheritance subhierarchy rooted at class c',*
(3) $\exists c''$ *that satisfies* (1) *or* (2) *(by substituting c'' for c) and c is in the class composition subhierarchy rooted at class c''.*

Definition 3.7 (Modification List). *Given a class c and let $V_{id_1}, V_{id_2}, \ldots, V_{id_n}$, be all the views that depend on class c. The modification list of class c, denoted $M_{list}(c)$, is a sequence of n modification tuples with one and only one tuple per view dependent on class c such that:*

(1) *the tuple at the tail of $M_{list}(c)$ represents the most recently accessed view,*
(2) *modifications related to objects of class c are always registered inside the tuple at the tail of $M_{list}(c)$,*
(3) *each tuple marks the starting point of the forthcoming update to the corresponding view, i.e., modifications inside the sequence of tuples starting with the representative tuple of a given view until the end of $M_{list}(c)$ are utilized to update that view.*

Definition 3.8 (Modification Tuple). *Given a class c and a view V_{id} that depends on class c. A modification tuple of V_{id} inside $M_{list}(c)$, denoted $M_{tuple}(V_{id})$, is a quadruple $(V_{id}, O_{inserted}, O_{deleted},$*

[4] C, I_V and M_T are the sets of all class names, instance variables and methods, respectively.

$O_{updated}$), where $O_{inserted}$, $O_{deleted}$ 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 while $M_{tuple}(V_{id})$ was occupying the tail of $M_{list}(c)$, i.e., before the update of another view that depends on class c.

Definition 3.9 (P_o of Virtual Classes). *For every virtual class V_{id}, $P_o(V_{id})$ refers to a triplet ($W_{instances}(V_{id})$, $L_{behavior}(V_{id})$, $M_{list}(V_{id})$), where $W_{instances}(V_{id})$ is the set of oid's for objects qualified to be in view V_{id} according to a method, FindObjects(V_{id}), in $L_{behavior}(V_{id})$. The method FindObjects(V_{id}), implements the (possibly optimized) query expression used in deriving view V_{id}; and hence has different implementations for different views. In addition to FindObjects(V_{id}), $L_{behavior}(V_{id})$ may include other methods particular to view V_{id}.*

Definition 3.10 (View Construction). *A view can be constructed using the following query language:*

Define view ⟨view-identifier⟩ as
Objects understand ⟨set-of-messages⟩
Utilize ⟨class-list⟩
Such that ⟨predicate⟩

where the 'Such that' part is optional and
⟨set-of-messages⟩ is a set of messages with each message presented as m:d, where d is either the identifier of one of the classes present in the class hierarchy such that $m \in W_{behavior}(d)$, or d may be a function that defines a new method with name m;
⟨class-list⟩ is a set of classes from the class hierarchy;
⟨predicate⟩ is a predicate expression that acts as a filter while deciding on objects to be included in a view. For more details on predicate expressions see [5].

This way, we differentiate between two categories of classes according to the instantiation of P_d; *base* classes and *virtual* classes. A base class directly points to a class definition in class δ, i.e., its P_d includes the *oid* of only one object in class δ. A *virtual* class, on the other hand, holds a view definition and indirectly refers to object(s) in class δ, i.e., its P_d includes the identifier(s) of the base class(es) and/or other view(s) from which it was derived. In other words, virtual classes share class definitions of base classes and objects of a virtual class V_{id} are determined depending on its *FindObjects*(V_{id}) method. Actually, views add a new dimension to the class hierarchy which is the view derivation line. Explicitly, each view is the root of a semi-lattice the leaves of which are base classes and all other intermediate internal nodes are virtual classes. When a given view is to be updated, then all virtual classes inside the semi-lattice rooted at that particular view have to be updated. This is recursively applied, as we will see in the next section, by utilizing representative modification tuples in the modification lists of related base and virtual classes.

3.2. Illustrating examples

To illustrate what has been introduced so far, included in Fig. 3 are the properties of the base classes shown in Fig. 1, where [ ] and { } indicate lists and sets, respectively. For instance, *author:*[*Researcher*], *author:*{*Researcher*} and *author:Researcher* indicate that the attribute *author* has its value as a list of objects, a set of objects and a single object from the *Researcher* class, respectively. Next to each class in Fig. 3, there are four collections that, respectively, include local

attributes, local behavior, direct superclasses and direct subclasses. As clear from Fig. 3, P_d of every class shown in Fig. 1 is defined to be an object in class δ. The P_o parts of the definitions of these classes are shown in Fig. 4. Modification lists of all classes are empty because no virtual classes have been defined yet. Concerning virtual classes, a user of our database might want to pose a series of questions about books published by Addison-Wesley. This could be accomplished by first generating a view of *Book*, where only books published by Addison-Wesley are included. This new virtual class would be the subject of further queries, thus making the query processing more efficient. Some other example virtual classes are enumerated next.

View 1. Find female researchers;

Define view *FemaleResearchers* as
Objects understand $W_{behavior}$(*Researcher*)
Utilize *Researcher*
Such that Sex(*p*) = "*F*" [5]

FemaleResearchers is a virtual class with P_d(*FemaleResearchers*) = {*Researcher*}, and $W_{instances}$(*FemaleResearchers*) = $\{o_{id_7}, o_{id_9}, o_{id_{10}}\}$
Notice that, *FemaleResearchers* and *Researcher* share the same object in class δ.

View 2. Find database research papers with at least one of the authors being a member in a program committee for a conference;

Define view *PCRP* as
Objects understand $W_{behavior}$(*ResearchPaper*)
Utilize *ResearchPaper*, *Proceedings*
Such that *Area*(*p*) = "*database*" and $\exists p_1 \exists p_2 (p_1$ *in Author*(*p*), $p_2 \in W_{instances}$(*Proceedings*) and $p_1 \in$ *Committee*(p_2))

PCRP is a virtual class with P_d = {*ResearchPaper*, *Proceedings*}, and $W_{instances}$(*PCRP*) = $\{o_{id_6}, o_{id_{12}}, o_{id_{14}}\}$

View 3. Find all publications of "*MEH Pub*";

P_o(*Book*):	• $L_{instances}$(*Book*) = $\{o_{id_1}, o_{id_2}\}$,	• M_{list}(*Book*)=[ ]
P_o(*Proceedings*):	• $L_{instances}$(*Proceedings*) = $\{o_{id_3}\}$,	• M_{list}(*Proceedings*)=[ ]
P_o(*Journal*):	• $L_{instances}$(*Journal*) = $\{o_{id_4}\}$,	• M_{list}(*Journal*)=[ ]
P_o(*SpecialIssue*):	• $L_{instances}$(*SpecialIssue*) = $\{o_{id_{13}}\}$,	• M_{list}(*SpecialIssue*)=[ ]
P_o(*ResearchPaper*):	• $L_{instances}$(*ResearchPaper*) = {},	• M_{list}(*ResearchPaper*)=[ ]
P_o(*JournalPaper*):	• $L_{instances}$(*JournalPaper*) = $\{o_{id_5}, o_{id_{12}}\}$,	• M_{list}(*JournalPaper*)=[ ]
P_o(*ConferencePaper*):	• $L_{instances}$(*ConferencePaper*) = $\{o_{id_6}, o_{id_{11}}, o_{id_{14}}\}$,	• M_{list}(*ConferencePaper*)=[ ]
P_o(*Researcher*):	• $L_{instances}$(*Researcher*) = $\{o_{id_7}, o_{id_8}, o_{id_9}, o_{id_{10}}\}$,	• M_{list}(*Researcher*)=[ ]

Fig. 4. The P_o parts of the definitions of the example classes shown in Fig. 1.

[5] Variable *p* is a pointer to receiver objects in the target class, here *Researcher*.

Define view MehPublications as
Objects understand $W_{behavior}(Book) \cap W_{behavior}(Journal)$
Utilize *Book*, *Journal*
Such that *Publisher*(*p*) = "*MEH Pub*"
MehPublications is a virtual class with $P_d = \{Book, Journal\}$, and
$W_{instances}(MehPublications) = \{o_{id_1}, o_{id_4}\}$
It is the union of the selections from *Book* and *Journal* classes based on the specified predicate.

As is obvious from the examples and detailed more in [7,8,9], virtual classes serve to hold query results. Their introduction do help in reusability maximization as they share the same class definitions with certain base classes. More than that, any changes to a base class are dynamically reflected to all its dependent virtual classes. Also, updates to virtual classes are actually performed against the related base class(es) as the latter are the only container of the actual database. Such updates are dynamically reflected to all accessing base and virtual classes. This way, database consistency and integrity are guaranteed. Beyond that, to update objects in any of the example virtual classes, it is necessary to execute Algorithm 4.3, given next in Section 4, that filters modified objects in the corresponding base class(es) against the *FindObjects* method of the given example class.

4. View maintenance algorithms

In this section, we elaborate more on the basic model introduced in Section 3 and describe the algorithms that facilitate incremental maintenance and materialization of object-oriented views.

4.1. The mechanism of modification lists

Recall that, any base class is in general the root of two orthogonal subhierarchies, more precisely a semi-lattice and a subgraph corresponding to the inheritance subhierarchy and the class-composition subhierarchy, respectively. Based on that and by definition, $W_{instances}(c)$, for any base class c, subsumes those of classes along the inheritance subhierarchy rooted at class c. Furthermore, every object in $W_{instances}(c)$ references directly or indirectly via nesting, due to complex structures, objects in classes along the class-composition subhierarchy rooted at class c. Consequently, it is not sufficient to reflect into views that depend on class c only *local modifications* to objects in $W_{instances}(c)$, *global modifications* should also be considered.

Definition 4.1 (Local and Global Modifications). *Given a class c, an object* $oid \in W_{instances}(c)$ *and any view* V_{id}. *Object oid is said to be locally modified with respect to class c if and only if:*
$oid \in L_{instances}(c)$ *and* $\exists M_{tuple}(V_{id}) \in M_{list}(c)$ *such that* $oid \in O_{inserted} \cup O_{deleted} \cup O_{updated}$
On the other hand, oid is said to be globally modified with respect to class c if and only if:

either $oid \in L_{instances}(c')$ *where c′ is a class from the inheritance subhierarchy rooted at class c and* $\exists M_{tuple}(V_{id}) \in M_{list}(c')$ *such that* $oid \in O_{inserted} \cup O_{deleted} \cup O_{updated}$
or $\exists oid'' \in L_{instances}(c'')$ *where c″ is a class in the class-composition subhierarchy rooted at class c*

and $\exists M_{tuple}(V_{id}) \in M_{list}(c'')$ *such that oid*$'' \in O_{inserted} \cup O_{deleted} \cup O_{updated}$ *and there is a path from oid to oid'', i.e., oid references oid''.*

According to Definition 4.1, 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. Locating and controlling global modifications is as important as local modifications and also it is necessary and required in preserving database consistency and integrity.

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 view identifier, say V_{id}, to the modification list of each class along both subhierarchies rooted at class c. Such tuples are utilized in locating which objects from $W_{instances}(c)$ to consider in the process of updating view V_{id}. To illustrate this, shown in Fig. 5 and Fig. 6 are the modification lists of the base classes in Fig. 1, after the definition of the example virtual classes introduced in Section 3. Modification lists in Fig. 5 reflect a situation where database contents have not been modified since the first view *FemaleResearchers* was defined. On the other hand, in Fig. 6 we assume that some operations were performed on certain objects after defining each of the three views in order to show how modifications are reflected into tuples within modification lists.

Explicitly, in Fig. 5, a modification tuple, (*FemaleResearchers*, ϕ, ϕ, ϕ), is added to M_{list}(*Researcher*) after defining virtual class *FemaleResearchers* because *Researcher* is the only class in P_d(*FemaleResearchers*) and there is no other class present in the class-composition subhierarchy or the inheritance subhierarchy rooted at the *Researcher* class. After virtual class *PCRP* is defined, (*PCRP*, ϕ, ϕ, ϕ) is added to M_{list}(*ResearchPaper*) and M_{list}(*Proceedings*) because the two classes belong to P_d(*PCRP*). Also, (*PCRP*, ϕ, ϕ, ϕ) is added to M_{list}(*JournalPaper*), M_{list}(*ConferencePaper*) and M_{list}(*SpecialIssue*) because the latter class belongs to the inheritance subhierarchy rooted at *Proceedings* and the other two classes are in the inheritance subhierarchy rooted at *ResearchPaper*. The class-composition subhierarchy rooted at *Proceedings* includes two classes, *ConferencePaper* and *Researcher*. As the former class has already been considered, (*PCRP*, ϕ, ϕ, ϕ) is added to M_{list}(*Researcher*). The process of defining *PCRP* is terminated by considering the class-composition subhierarchy rooted at *ResearchPaper* which contains *Researcher* that has already been considered. Finally, the effect of defining virtual class *MehPublications* is treated the same way.

As far as Fig. 6 is concerned, assume that after *FemaleResearchers* is defined, object o_{id_7} is deleted from and object o_{id_9} is updated in the *Researcher* class; this is reflected into the modification lists shown in Fig. 6a. Second, as shown in Fig. 6b, following the definition of *PCRP*, object o_{id_9} is deleted

$$
\begin{aligned}
&M_{list}(Researcher)=[(FemaleResearchers, \phi, \phi, \phi), (\overline{PCRP}, \phi, \phi, \phi),\\
&\qquad\qquad (MehPublications, \phi, \phi, \phi)]\\
&M_{list}(Book)=[(MehPublications, \phi, \phi, \phi)]\\
&M_{list}(Proceedings)=[(PCRP, \phi, \phi, \phi), (MehPublications, \phi, \phi, \phi)]\\
&M_{list}(Journal)=[(MehPublications, \phi, \phi, \phi)]\\
&M_{list}(SpecialIssue)=[(PCRP, \phi, \phi, \phi), (MehPublications, \phi, \phi, \phi)]\\
&M_{list}(ResearchPaper)=[(PCRP, \phi, \phi, \phi)]\\
&M_{list}(JournalPaper)=[(PCRP, \phi, \phi, \phi), (MehPublications, \phi, \phi, \phi)]\\
&M_{list}(ConferencePaper)=[(PCRP, \phi, \phi, \phi), (MehPublications, \phi, \phi, \phi)]
\end{aligned}
$$

Fig. 5. Modification lists of the base classes in Fig. 1 with the assumption that database contents are not modified.

$M_{list}(Researcher)$=[($FemaleResearchers$, ϕ, {o_{id_7}}, {o_{id_9}})]
$M_{list}(Book)$=[]
$M_{list}(Proceedings)$=[]
$M_{list}(Journal)$=[]
$M_{list}(SpecialIssue)$=[]
$M_{list}(ResearchPaper)$=[]
$M_{list}(JournalPaper)$=[]
$M_{list}(ConferencePaper)$=[]

(a) After the addition of *FemaleResearchers* view

$M_{list}(Researcher)$=[($FemaleResearchers$, ϕ, {o_{id_7}}, {o_{id_9}}), ($PCRP$, ϕ, {o_{id_9}}, ϕ)]
$M_{list}(Book)$=[]
$M_{list}(Proceedings)$=[($PCRP$, ϕ, ϕ, ϕ)]
$M_{list}(Journal)$=[]
$M_{list}(SpecialIssue)$=[($PCRP$, ϕ, ϕ, ϕ)]
$M_{list}(ResearchPaper)$=[($PCRP$, ϕ, ϕ, ϕ)]
$M_{list}(JournalPaper)$=[($PCRP$, ϕ, ϕ, ϕ)]
$M_{list}(ConferencePaper)$=[($PCRP$, ϕ, ϕ, {$o_{id_{11}}$})]

(b) After the addition of *PCRP* view

$M_{list}(Researcher)$=[($FemaleResearchers$, ϕ, {o_{id_7}}, {o_{id_9}}), ($PCRP$, ϕ, {o_{id_9}}, ϕ), ($MehPublications$, ϕ, ϕ, ϕ)]
$M_{list}(Book)$=[($MehPublications$, ϕ, {o_{id_1}}, ϕ)]
$M_{list}(Proceedings)$=[($PCRP$, ϕ, ϕ, ϕ), ($MehPublications$, ϕ, ϕ, ϕ)]
$M_{list}(Journal)$=[($MehPublications$, ϕ, ϕ, ϕ)]
$M_{list}(SpecialIssue)$=[($PCRP$, ϕ, ϕ, ϕ), ($MehPublications$, ϕ, ϕ, ϕ)]
$M_{list}(ResearchPaper)$=[($PCRP$, ϕ, ϕ, ϕ)]
$M_{list}(JournalPaper)$=[($PCRP$, ϕ, ϕ, ϕ), ($MehPublications$, ϕ, ϕ, {o_{id_5}})]
$M_{list}(ConferencePaper)$=[($PCRP$, ϕ, ϕ, {$o_{id_{11}}$}), ($MehPublications$, ϕ, ϕ, ϕ)]

(c) After the addition of *MehPublications* view

Fig. 6. Modification lists of the base classes in Fig. 1 with the assumption that some modifications are performed on database contents after defining each of the three virtual classes.

from the *Researcher* class and object $o_{id_{11}}$ is updated in the *ConferencePaper* class. Finally, it is shown in Fig. 6c that after view *MehPublications* is defined, object o_{id_1} is deleted from class *Book* and object o_{id_5} is updated in the *JournalPaper* class. Although we concentrate on deletion and updating of objects in Fig. 6, additions are treated the same way by including the *oid* of an added object in the first modification set inside the modification tuple at the tail of the modification list of the class to which the object is added. So, as it is obvious from Fig. 5 and Fig. 6, the creation of a virtual class V_{id} results in the addition of a modification tuple to the modification list of every class in $P_d(V_{id})$ as well as to classes in the inheritance and class-composition subhierarchies rooted at class c. To update a view, it is necessary to locate and utilize all relevant modification information.

4.2. Locating global modifications

Modifications related to class c along the inheritance hierarchy are located by recursively tracing the C_b's of classes found in the inheritance subhierarchy rooted at class c. On the other hand, to successfully reflect modifications along the class-composition subhierarchy rooted at class c, it is necessary to find 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. The definitions in class δ for both *NEC* and *COR* classes are given next.

- $C_p(NEC) = [\,]$,
- $C_b(NEC) = \phi$,
- $L_{attributes}(NEC) = \{LeftClass{:}C, RightClass{:}C\}$,
- $L_{behavior}(NEC) = \{FindClassCompositionHierarchy(\,)\}$,
- $C_p(COR) = [\,]$,
- $C_b(COR) = \phi$,
- $L_{attributes}(COR) = \{LeftObject{:}O_{ID}, RightObject{:}O_{ID}\}$,[6]
- $L_{behavior}(COR) = \{FindReferenceingObjects(TargetClass)\}$

Actually, *NEC* and *COR* classes were introduced as a part of the Object-Oriented Database Management System developed at Bilkent University [5,6,7,8,9,10,17,35]. 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 relationship 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. To illustrate this, shown in Fig. 7 are the objects contained in *NEC* and *COR* classes, as the example classes in Fig. 1 and the corresponding objects in Fig. 2 are concerned.

NEC

$o_{id_{15}}$[*Book, Researcher*]	$o_{id_{16}}$[*Journal, Researcher*]	$o_{id_{17}}$[*ResearchPaper, Researcher*]
$o_{id_{18}}$[*Proceedings, Researcher*]	$o_{id_{19}}$[*Journal, JournalPaper*]	$o_{id_{20}}$[*Proceedings, ConferencePaper*]

COR

$o_{id_{21}}[o_{id_7}, o_{id_1}]$	$o_{id_{22}}[o_{id_8}, o_{id_1}]$	$o_{id_{23}}[o_{id_9}, o_{id_2}]$	$o_{id_{24}}[o_{id_8}, o_{id_3}]$	$o_{id_{25}}[o_{id_9}, o_{id_3}]$
$o_{id_{26}}[o_{id_{10}}, o_{id_3}]$	$o_{id_{27}}[o_{id_7}, o_{id_3}]$	$o_{id_{28}}[o_{id_6}, o_{id_3}]$	$o_{id_{29}}[o_{id_{11}}, o_{id_3}]$	$o_{id_{30}}[o_{id_9}, o_{id_4}]$
$o_{id_{31}}[o_{id_7}, o_{id_4}]$	$o_{id_{32}}[o_{id_8}, o_{id_4}]$	$o_{id_{33}}[o_{id_{12}}, o_{id_4}]$	$o_{id_{34}}[o_{id_{10}}, o_{id_5}]$	$o_{id_{35}}[o_{id_7}, o_{id_6}]$
$o_{id_{36}}[o_{id_9}, o_{id_6}]$	$o_{id_{37}}[o_{id_8}, o_{id_{11}}]$	$o_{id_{38}}[o_{id_{10}}, o_{id_{11}}]$	$o_{id_{39}}[o_{id_7}, o_{id_{12}}]$	$o_{id_{40}}[o_{id_{10}}, o_{id_{12}}]$
$o_{id_{41}}[o_{id_7}, o_{id_{13}}]$	$o_{id_{42}}[o_{id_9}, o_{id_{13}}]$	$o_{id_{43}}[o_{id_{10}}, o_{id_{13}}]$	$o_{id_{44}}[o_{id_{14}}, o_{id_{13}}]$	$o_{id_{45}}[o_{id_7}, o_{id_{14}}]$
$o_{id_{46}}[o_{id_8}, o_{id_{14}}]$				

Fig. 7. Objects in *NEC* and *COR* classes.

[6] O_{ID} is the set of all object identifiers.

It is worth mentioning that each of the methods underlying the behavior of both *NEC* and *COR* is implemented as a recursive query [10]. This is achieved because in our query model described in [5,9] we allow the specification of the result of a recursive query to be a subset of the transitive closure. This way, the method underlying *FindClassCompositionHierarchy*(c) determines all the classes along the class-composition subhierarchy rooted at the receiver class c. While doing that, classes in $C_p(c)$ are considered because attributes with complex domains in $W_{attributes}(c)$ may be either locally defined or inherited. For instance, *FindClassCompositionHierarchy*(*Proceedings*) returns the list [*Researcher*, *ConferencePaper*]. On the other hand, the method underlying *FindReferencingObjects*(*ObjectID*, *TargetClass*) determines all objects within $W_{instances}$(*TargetClass*) and referencing the receiver object. However, this method imposes the restriction that the class of the receiver object must be in the class-composition subhierarchy rooted at the parameter *TargetClass*. For instance, *FindReferencingObjects*($o_{id_{10}}$, *Journal*) returns $\{o_{id_{13}}\}$. Finally, objects in the two classes *NEC* and *COR* are utilized by the algorithms given in the next section to facilitate the creation and incremental update of any virtual class V_{id}.

4.3. The algorithms

The target of incrementally updating a given view V_{id} is broken down into finding all relevant modifications and employ such modifications in the update process. Modifications of concern are contained in all classes found in the class-composition and inheritance subhierarchies rooted at each class in $P_d(V_{id})$. Algorithm 4.3 achieves the targeted update process by breaking the latter task down further into finding modifications related to each class c in $P_d(V_{id})$ by utilizing Algorithm 4.2. Then, all such modifications are accumulated and used as input to the *FindObjects*(V_{id}) method in $L_{behavior}(V_{id})$. The latter method performs the actual update by reflecting located modifications to $W_{instances}(V_{id})$. Algorithm 4.2 takes itself the responsibility of finding modifications along the class-composition subhierarchy rooted at class c and asks for the help of Algorithm 4.1 which is specialized in finding modifications along the inheritance subhierarchy rooted at a given class. The three algorithms are given next.

Algorithm 4.1 (Inheritance Modifications).
Input. *A view* V_{id} *and a class* c.
Output. $C_{O_{inserted}}(c)$, $C_{O_{deleted}}(c)$, *and* $C_{O_{updated}}(c)$, *that are the sets of oid's of modified objects in* $W_{instances}(c)$.

Steps:

Let $C_{O_{inserted}}(c) = C_{O_{deleted}}(c) = C_{O_{updated}}(c) = \phi$
Let $C_{inheritance} = [c]$
/* $C_{inheritance}$ *is a list to include all classes found along the inheritance subhierarchy rooted at class* c. */
Let $i = 1$
While not end of $C_{inheritance}$ *do*
Let $c' = C_{inheritance}[i]$
$C_{inheritance} = C_{inheritance} + C_b(c')$

Let $M_{tuple}(V_{id})$ *be at position* j *within* $M_{list}(c')$ *and let* $k = j$.
While not end of $M_{list}(c')$ *do*
/* *The tuple at position* k *in* $M_{list}(c')$ *includes three sets, namely* $O_{inserted}[k]$, $O_{deleted}[k]$ *and* $O_{updated}[k]$. */
$C_{O_{inserted}}(c) = C_{O_{inserted}}(c) \cup O_{inserted}[k]$
$C_{O_{deleted}}(c) = C_{O_{deleted}}(c) \cup O_{deleted}[k]$
$C_{O_{update}}(c) = C_{O_{updated}}(c) \cup O_{updated}[k]$
$k = k + 1$
EndWhile
If there exists an immediate predecessor of $M_{tuple}(V_{id})$ *in* $M_{list}(c')$ *then*
. *Merge the modification information inside* $M_{tuple}(V_{id})$ *with that of its immediate predecessor tuple by setting*:
$O_{inserted}[j-1] = O_{inserted}[j-1] \cup O_{inserted}[j]$,
$O_{deleted}[j-1] = O_{deleted}[j-1] \cup O_{deleted}[j]$ *and*
$O_{updated}[j-1] = O_{updated}[j-1] \cup O_{updated}[j]$
EndIf
Delete $M_{tuple}(V_{id})$ *from* $M_{list}(c')$
/* *To mark the starting point for the next update of view* V_{id}. */
Append $(V_{id}, \phi, \phi, \phi)$ *at the tail of* $M_{list}(c')$
EndWhile
EndAlgorithm

Algorithm 4.2 (OperandRelatedModifications).
Input. *View* V_{id} *and a class* c *from* $P_d(V_{id})$
Output. *Sets of objects* $W_{O_{inserted}}(c)$, $W_{O_{deleted}}(c)$, $W_{O_{updated}}(c)$ *to be utilized in Algorithm 4.3 for incremental update to view* V_{id}.

Steps:
Let $W_{O_{inserted}}(c) = W_{O_{deleted}}(c) = W_{O_{updated}}(c) = \phi$
$C_{cch} = FindClassCompositionHierarchy(c)$
/* C_{cch} *is a list to include classes within the class-composition subhierarchy rooted at class* c. */
Find – $M_{tuple}(V_{id})$ *within* $M_{list}(c)$.
If $M_{tuple}(V_{id})$ *is not found then*
/* *View* V_{id} *is a new view and should be derived from scratch.* */
$W_{O_{inserted}}(c) = W_{instances}(c)$
$W_{O_{deleted}}(c) = W_{O_{updated}}(c) = \phi$
For every class c' *in* $[c] + C_{cch}$ *do*
Let $C_{inheritance} = [c']$ *and let* $i = 1$
While not end of $C_{inheritance}$ *do*
Let $c'' = C_{inheritance}[i]$
$C_{inheritance} = C_{inheritance} + C_b(c'')$
Append $(V_{id}, \phi, \phi, \phi)$ *at the tail of* $M_{list}(c'')$
$i = i + 1$

EndWhile
EndFor
Else
Perform InheritanceModifications(V_{id}, c)
$W_{O_{inserted}}(c) = C_{O_{inserted}}(c)$
$W_{O_{deleted}}(c) = C_{O_{deleted}}(c)$
$W_{O_{updated}}(c) = C_{O_{updated}}(c)$
Let $G = \phi$
/* *G is a set to include all modified objects along the class-composition subhierarchy rooted at class c. Here we are interested only in deleted and updated objects because existing object references are affected only by indirect deletions and updates. Objects added along the class-composition hierarchy affect the update process only when they are referenced by some other objects that are marked as updated objects and hence are already included in G.* */
For every class c_i *in* C_{cch} *do*
Perform InheritanceModifications(V_{id}),
$G = G \cup C_{O_{deleted}}(c_i) \cup C_{O_{updated}}(c_i)$
EndFor
For every object oid $\in G$ *do*
. $W_{O_{updated}}(c) = W_{O_{updated}}(c)$ + *FindReferencingObjects*(oid, c)
EndFor
$W_{O_{inserted}}(c) = W_{O_{inserted}}(c) - W_{O_{deleted}}(c)$
$W_{O_{updated}}(c) = W_{O_{updated}}(c) - W_{O_{inserted}}(c)$
$W_{O_{updated}}(c) = W_{O_{updated}}(c) - W_{O_{deleted}}(c)$
EndIf
EndAlgorithm

Algorithm 4.3 (View Update):
Input: *A virtual class* V_{id} *and* $P_d(V_{id})$
Output: *The updated version of* V_{id}

Steps:
Let $W_{O_{inserted}}(V_{id}) = W_{O_{deleted}}(V_{id}) = \phi$
For every class $c \in P_d(V_{id})$ *do*
If c is an object in class δ *then* /* *c is a base class* */
Perform OperandRelatedModifications(Vid, c)
/* *When an object in class c is updated, virtual class* V_{id} *must recognize that change. This is achieved by forcing* V_{id} *to recognize an updated object as a new object.* */
$W_{O_{deleted}}(c) = W_{O_{deleted}}(c) \cup W_{O_{updated}}(c)$
$W_{O_{inserted}}(c) = W_{O_{inserted}}(c) \cup W_{O_{updated}}(c)$
For every object oid $\in W_{O_{deleted}}(c)$ *do*
$W_{O_{deleted}}(V_{id}) = W_{O_{deleted}}(V_{id}) \cup$ *FindReferencingObjects*(oid, Vid)
EndFor

Else /* *c is a virtual class* */
Let $W_{O_{inserted}}(c) = W_{O_{deleted}}(c) = \phi$
Perform ViewUpdate$(c, P_d(c))$
If $M_{tuple}(V_{id})$ *doesnot exist in* $M_{list}(c)$ *then*
$W_{O_{inserted}}(c) = W_{instances}(c)$
$W_{O_{deleted}}(c) = \phi$
Else
Let $M_{tuple}(V_{id})$ *be at position* j *within* $M_{list}(c)$.
While not end of $M_{list}(c)$ *do*
/* *The tuple at position* j *in* $M_{list}(c)$ *includes two sets, namely* $O_{inserted}[j]$ *and* $O_{deleted}[j]$. */
$W_{O_{inserted}}(c) = W_{O_{inserted}}(c) \cup O_{inserted}[j]$
$W_{O_{deleted}}(c) = W_{O_{deleted}} \cup O_{deleted}[j]$
$j = j + 1$
EndWhile
If there exists an immediate predecessor of $M_{tuple}(V_{id})$ *in* $M_{list}(c)$ *then*
. *Merge modification information inside* $M_{tuple}(V_{id})$ *with that of its immediate predecessor tuple in* $M_{list}(c)$
EndIf
Delete $M_{tuple}(V_{id})$ *from* $M_{list}(c)$
EndIf
/* *To mark the starting point for the next update of view* V_{id} */
Append (V_{id}, ϕ, ϕ) *at the tail of* $M_{list}(c)$
EndIf
EndFor
Determine $W_{O_{inserted}}(V_{id})$ *by executing FindObjects*(V_{id}) *against* $W_{instances}(c)$ *and* $W_{O_{inserted}}(c)$ *of every class* c *in* $P_d(V_{id})$
If $M_{list}(V_{id})$ *is not empty then*
/**To be able to reflect the already done update to dependent views.* */
Let the tuple at the tail of $M_{list}(V_{id})$ *be at position* k
. $O_{inserted}[k] = O_{inserted}[k] \cup W_{O_{inserted}}(V_{id})$
. $O_{deleted}[k] = O_{deleted}[k] \cup W_{O_{deleted}}(V_{id})$
EndIf
$W_{instances}(V_{id}) = W_{instances}(V_{id}) - W_{O_{deleted}}(V_{id}) \cup W_{O_{inserted}}(V_{id})$
EndAlgorithm

Informally, given a class c and a view V_{id}, Algorithm 4.1 determines, related to objects in the extent of class c, modifications that are to be used in updating view V_{id}. This is achieved by considering a group of modification lists including that of class c itself and all classes along the inheritance subhierarchy rooted at class c. In each such list, the contents of $M_{tuple}(V_{id})$ are examined together with the contents of its successor tuples until the tail of that modification list. Then the modification sets inside $M_{tuple}(V_{id})$ are merged with those of its immediate predecessor tuple within the same list because such contents are still necessary to views corresponding to its predecessor tuples until the head of the modification list. Further, to indicate that only subsequent modifications are necessary in

the forthcoming update of view V_{id}, a modification tuple $(V_{id}, \phi, \phi, \phi)$ is appended at the tail of each accessed list to mark the starting point for the next update to view V_{id}. Subsequently, Algorithm 4.1 is utilized in Algorithm 4.2 that determines all the modifications within $W_{instances}(c)$, where c is one of the classes in $P_d(V_{id})$. While the inheritance subhierarchy is considered in Algorithm 4.1, the class-composition subhierarchy rooted at class c is considered in Algorithm 4.2 that checks first whether V_{id} is a new or an existing view. For the former case, all objects in $W_{instances}(c)$ are considered in deriving $W_{instances}(V_{id})$ because V_{id} does not possess any prior knowledge about database contents. For the latter case, on the other hand, all related classes are visited to determine relevant modifications done on the database since the last update of view V_{id}. In both cases, a modification tuple is appended at the tail of visited modification lists to mark the starting point for the next update of view V_{id}. Finally, located modifications are necessary for Algorithm 4.3 that employs recursion because any view is the root of a semi-lattice in which internal nodes are virtual classes and leaves are base classes. Consequently, it is necessary to update all views constituting internal nodes before updating the actual target view at the root. Recursion guarantees the gradual update of internal views starting with those solely dependent on base classes until the target view is updated. Finally, Algorithm 4.3 executes the appropriate implementation of *FindObjects*(V_{id}) that has different implementations for different virtual classes.

To illustrate the already introduced algorithms, assume that it is required to access *FemaleResearchers* view. Because we employ deferred update, objects in a virtual class are updated only when that class is accessed. Consequently, Algorithm 4.3 is executed and calls Algorithm 4.2 which, in its turn, utilizes Algorithm 4.1. So, on executing *ViewUpdate*(*FemaleResearchers*), the related modification lists shown in Fig. 6c are traced by Algorithms 4.2 and 4.1 to locate modified objects in $W_{instances}$(*Researcher*) because *Researcher* is the only class in P_d(*FemaleResearchers*). The only processed modification list is M_{list}(*Researcher*) and the following three sets are returned by Algorithms 4.2:

$W_{O_{inserted}}(Researcher) = \phi$,
$W_{O_{deleted}}(Researcher) = \{o_{id_7}, o_{id_9}\}$,
$W_{O_{updated}}(Researcher) = \phi$.

Algorithm 4.3 utilizes these three sets to return $W_{instances}(FemaleResearchers) = \{o_{id_{10}}\}$. To guarantee the correctness of the next incremental update to the virtual class *FemaleResearchers*, its modification tuple is eliminated from M_{list}(*Researcher*) for being the first tuple in that list; otherwise, contents of that tuple should have been merged with its predecessor tuple before being eliminated. A new tuple holding the identifier of *FemaleResearchers* with empty sets is appended at the end of M_{list}(*Researcher*), as shown in Fig. 8.

$M_{list}(Researcher)$=[$(PCRP, \phi, \{o_{id_9}\}, \phi)$, $(MehPublications, \phi, \phi, \phi)$,
$(FemaleResearchers, \phi, \phi, \phi)$]
$M_{list}(Book)$=[$(MehPublications, \phi, \{o_{id_1}\}, \phi)$]
$M_{list}(Proceedings)$=[$(PCRP, \phi, \phi, \phi)$, $(MehPublications, \phi, \phi, \phi)$]
$M_{list}(Journal)$=[$(MehPublications, \phi, \phi, \phi)$]
$M_{list}(SpecialIssue)$=[$(PCRP, \phi, \phi, \phi)$, $(MehPublications, \phi, \phi, \phi)$]
$M_{list}(ResearchPaper)$=[$(PCRP, \phi, \phi, \phi)$]
$M_{list}(JournalPaper)$=[$(PCRP, \phi, \phi, \phi)$, $(MehPublications, \phi, \phi, \{o_{id_5}\})$]
$M_{list}(ConferencePaper)$=[$(PCRP, \phi, \phi, \{o_{id_{11}}\})$, $(MehPublications, \phi, \phi, \phi)$]

Fig. 8. Modification lists of the classes in Fig. 1 after the update of the *FemaleResearchers* view.

$M_{list}(Researcher)$=[($FemaleResearchers$, ϕ, $\{o_{id_7}, o_{id_9}\}$, $\{o_{id_9}\}$), ($MehPublications$, ϕ, ϕ, ϕ), ($PCRP$, ϕ, ϕ, ϕ)]
$M_{list}(Book)$=[($MehPublications$, ϕ, $\{o_{id_1}\}$, ϕ)]
$M_{list}(Proceedings)$=[($MehPublications$, ϕ, ϕ, ϕ), ($PCRP$, ϕ, ϕ, ϕ)]
$M_{list}(Journal)$=[($MehPublications$, ϕ, ϕ, ϕ)]
$M_{list}(SpecialIssue)$=[($MehPublications$, ϕ, ϕ, ϕ), ($PCRP$, ϕ, ϕ, ϕ)]
$M_{list}(ResearchPaper)$=[($PCRP$, ϕ, ϕ, ϕ)]
$M_{list}(JournalPaper)$=[($MehPublications$, ϕ, ϕ, $\{o_{id_5}\}$), ($PCRP$, ϕ, ϕ, ϕ)]
$M_{list}(ConferencePaper)$=[($MehPublications$, ϕ, ϕ, ϕ), ($PCRP$, ϕ, ϕ, ϕ)]

(a) Before the update of *FemaleResearchers* view

$M_{list}(Researcher)$=[($MehPublications$, ϕ, ϕ, ϕ), ($FemaleResearchers$, ϕ, ϕ, ϕ), ($PCRP$, ϕ, ϕ, ϕ)]
$M_{list}(Book)$=[($MehPublications$, ϕ, $\{o_{id_1}\}$, ϕ)]
$M_{list}(Proceedings)$=[($MehPublications$, ϕ, ϕ, ϕ), ($PCRP$, ϕ, ϕ, ϕ)]
$M_{list}(Journal)$=[($MehPublications$, ϕ, ϕ, ϕ)]
$M_{list}(SpecialIssue)$=[($MehPublications$, ϕ, ϕ, ϕ), ($PCRP$, ϕ, ϕ, ϕ)]
$M_{list}(ResearchPaper)$=[($PCRP$, ϕ, ϕ, ϕ)]
$M_{list}(JournalPaper)$=[($MehPublications$, ϕ, ϕ, $\{o_{id_5}\}$), ($PCRP$, ϕ, ϕ, ϕ)]
$M_{list}(ConferencePaper)$=[($MehPublications$, ϕ, ϕ, ϕ), ($PCRP$, ϕ, ϕ, ϕ)]

(b) After the update of *FemaleResearchers* view

Fig. 9. Modification lists of the classes in Fig. 1 after the update of view *PCRP*.

Here, it worth elaborating more on the advantage of employing deferred update. From Fig. 6c, it is obvious that object o_{id_9} was updated before being deleted. However, only the deletion is reflected into the virtual class *FemaleResearchers* during the above mentioned update process. It is not necessary and even time consuming to inform the virtual class *FemaleResearchers* about the update of object o_{id_9} because that class has never become active during the validity of the update of object o_{id_9}. To realize the change in the modification lists more explicitly, consider two executions of *ViewUpdate(PCRP)*; one by utilizing the lists in Fig. 6c and another by utilizing Fig. 8, i.e., before and after executing *ViewUpdate(FemaleResearchers)*, respectively. Modification lists of the base classes after such executions are shown in Fig. 9a and Fig. 9b, respectively. The merging of modification tuples is illustrated in the first execution of *ViewUpdate(PCRP)* where object o_{id_9} moved from the set of deleted objects inside $M_{tuple}(PCRP)$, the second tuple within $M_{list}(Researcher)$ in Fig. 6c, into the set of deleted objects inside $M_{tuple}(FemaleResearchers)$, the first tuple within $M_{list}(Researcher)$ in Fig. 9a.

5. Conclusions

In this paper, we presented a model that facilitates incremental materialization of views where a view may be derived from a number of existing views and/or classes. We argue that our model is a step forward in filling the existing gap about the requirements of object-oriented views. To have a

homogeneous system where closure is maintained, we defined a view to be a class. This way, a view possesses the characteristics of a class and hence qualifies to be an operand in any applicable query. To achieve deferred incremental update, to each class we added a modification list to include class related modifications ordered such that the most recent modifications are at the tail. Also, a view has a reference point in each related modification list to mark the start of its next update. In addition, we introduced some algorithms that utilize such modifications to help in avoiding view computation from scratch each time a view is accessed. Instead, the algorithms locate and reflect to a view only related modifications introduced into the database while that view was inactive. Such a method proved to be vital, especially in distributed environments including data warehouses where it is necessary to reduce network communication. Furthermore, deferred update proved to be more practical than immediate update especially when object-oriented databases are considered.

In general, database contents undergo a sequence of changes and a view is interested only in values valid at the time it is accessed without being interested in modification stages that led to those values. In addition, inheritance and nesting are missing in the conventional relational model and hence the cost of immediate update in that model is very low compared to nested relational and object-oriented models. However, due to inheritance and nesting, indirect modifications are more frequent in object-oriented databases and immediate reflection of indirect modifications to all related views negatively affects the performance of the system. This is illustrated in Fig. 8 where object o_{id_9} was updated then deleted and virtual class *FemaleResearchers* could not recognize the update because it was inactive during the validity of that update. One can say that applying immediate reflection is similar to asking the system to pay for some sandwiches that views will never eat. On the other hand, paid for sandwiches will always be eaten when deferred update is applied.

The presented model has been implemented as a part of an object-oriented database management system prototype under development at Bilkent University. Our first experience with the discussed model is that it gives very good results for databases that are not frequently modified. It includes the overhead of maintaining and processing the modification lists. To be more specific, consider a view V_{id} that may be either derived from scratch or incrementally maintained. Concerning the from scratch case, it is required to access each object in $W_{instances}(c)$ for every class c in $P_d(V_{id})$. On the other hand, only objects inserted or updated since the last materialization of view V_{id} need to be accessed for the deferred update case. The overhead with from scratch computation is processing untouched objects each time a view is accessed. Concerning the deferred update case, the number of modification lists to be processed is limited by the number of classes along both subhierarchies rooted at classes in $P_d(V_{id})$. As the number of objects in a database is very large compared to the number of classes, deferred update performs better than from scratch computation for small numbers of modified objects. From scratch computation gives better results for small numbers of untouched objects. Explicitly, the lower and upper bounds on the number of modified objects are, respectively, zero and $\Sigma_{i=1}^{n} W_{instances}(c_i)$, where n is the number of classes in $P_d(V_{id})$. From scratch computation performs better as the number tends to $\Sigma_{i=1}^{n} W_{instances}(c_i)$, and deferred update performs better as the number goes to zero. In general, only few objects in the database are inserted or modified between different sessions and most of the objects remain untouched. Therefore, as the number of objects in the database increase, deferred update performs better than from scratch computation.

Although our algorithms are designed for object-oriented view maintenance, they are still applicable to the nested and conventional relational models. In other words, we considered the inheritance and complex structures features of the object-oriented context in those algorithms, and

hence served more than the requirements of the relational model. However, it is necessary to include in a view the primary key of every operand because a primary key will act like an object identifier. Explicitly, as the inheritance concept is missing in both the nesting and conventional relational models, Algorithm 4.1 that concentrates on locating modifications along the inheritance hierarchy is not requested when dealing with views in such models. For the nested relational model, it is required to utilize both Algorithms 4.2 and 4.3. Concerning the conventional relational model, on the other hand, Algorithm 4.3 is enough to deal with views in that model because the conventional model does not cover nesting which is the responsibility of Algorithm 4.2. Finally, in addition to feeding the algorithms with the modifications required for view updates, modification lists serve some other purposes. More specifically, the information kept inside modification lists facilitate answering questions like the following:

- Total number of views directly dependent on a given class.
- Total number of views indirectly dependent on a given class.
- Classes with no dependent views.
- The order according to which views were defined, accessed or updated.
- The most recently accessed view among those directly dependent on a given class.
- The least recently accessed view among those dependent on a given class.
- Modifications perfomed on a given class since the last update of a certain view.
- The modification history of the database.

Currently, we are exploring the parallelization of the algorithms introduced in this paper to improve the performance of the system benefiting from the nature of the searched information. In other words, the searched for information is in the classes from which a view is derived and hence each such class may be searched independent of other classes. The independent results are then accumulated to update the target view. Furthermore, each class is the root of two subhierarchies that may be investigated independently. Another area that we are investigating is to extend the model to have a multi-agent system suitable for data warehousing applications where data models may be heterogeneous with an object-oriented model being the common language understandable by all agents. The agents must coordinate and negotiate together to have a given view updated. Finally, we are also looking into testing the efficiency of our approach for distributed object-oriented databases.

References

[1] S. Abiteboul, A. Bonner, Objects and views, *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, (1991) pp. 238–247.

[2] S. Abiteboul, O.M. Duschka, Complexity of answering queries using materialized views, *Proc. ACM PODS Int. Conf. on Principles of Database Systems*, (1998) pp. 254–263.

[3] S. Abiteboul et.al., Incremental maintenance for materialized views over semistructured data, *Proc. 24th Int. Conf. on Very Large Databases* (August 1998).

[4] A. Alashqur, S.Y. Su, H. Lam, OQL: A query language for manipulating object-oriented databases, *Proc. 15th Int. Conf. on Very Large Databases*, Amsterdam (August 1989) pp. 433–442.

[5] R. Alhajj, M.E. Arkun, A query model for object-oriented database systems, *Proc. 9th IEEE Int. Conf. on Data Engineering*, Vienna (April 1993) pp. 163–172.

[6] R. Alhajj, F. Polat, Reusability and schema evolution in an object-oriented query model, *Proc. European Conf. on Systems Design and Applications*, France (July 1996) pp. 21–29.

[7] R. Alhajj, F. Polat, Proper handling of query results towards maximizing reusability in object-oriented databases, *Information Sciences: An International Journal*, 107(1–4) (June 1998) 247–272.

[8] R. Alhajj, F. Polat, An object-oriented query model enforcing closure and reusability, *Proc. 10th Int. Conf. on Mathematical and Computer Modelling and Scientific Computing*, Boston, MA (July 1995), An extended version appeared in: *Journal of Mathematical Modeling and Computing* 6 (1996).

[9] R. Alhajj, F. Polat, Closure maintenance in an object-oriented query model, *Proc. ACM Int. Conf. on Information and Knowledge Management*, (November 1994) pp. 72–79.

[10] R. Alhajj, M.E. Arkun, Recursion in object-oriented databases, *Proc. European Conf. on Systems Design and Applications*, England (July 1994).

[11] L. Baekgaard, L. Mark, Incremental computation of set difference views, *IEEE Trans. on Knowledge and Data Engineering* 9(2) (1997) 251–261.

[12] E. Bertino, A view mechanism for object-oriented databases, *Proc. 3rd Int. Conf. on Extending Database Technology*, Vienna (March 1992) pp. 136–151.

[13] S. Ceri, J. Widom, Deriving production rules for incremental view maintenance, *Proc. 17th Int. Conf. on Very Large Databases*, Barcelona (September 1991) pp. 577–589.

[14] U. Dayal, Queries and views in an object-oriented data model, *Proc. 2nd Int. Workshop on Database Programming Languages* (June 1989) pp. 80–102.

[15] U. Dayal, H. Hwang, View definition and generalization for database integration in a multidatabase system, *IEEE Trans. on Software Engineering* (November 1984) pp. 628–645.

[16] O.M. Duschka, M.R. Genesereth, Answering recursive queries using views, *Proc. ACM PODS Int. Conf. on Principles of Database Systems* (June 1997) pp. 109–116.

[17] G. Eray et al., Storage and indexing facilities of an object-oriented database management system, *Proc. DECSYM*, Side (March 1992).

[18] D. Gluche et al., Incremental update for materialized OQL views, *Proc. 5th Int. Conf. on Deductive and Object-Oriented Databases*, (December 1997) pp. 52–66.

[19] A. Gupta, I. Mumick, V. Subrahmanian, Maintaining views incrementally, *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Washington DC (1993) pp. 157–166.

[20] E.N. Hanson, A performance analysis of view materialization strategies, *Proc. ACM-SIGMOD Int. Conf. on Management of Data* (1987) pp. 440–453.

[21] M. Hardwick, B.R. Downie, On object-oriented databases, materialized views, and concurrent engineering, In: A. Saxena (Ed.), *Proc. Database Symp. of the American Society of Mechanical Engineers* (August 1991).

[22] S. Heiler, S.B. Zdonik, Object views: Extending the vision, *Proc. 6th IEEE Int. Conf. on Data Engineering*, Los Angeles (February 1990) pp. 86–93.

[23] M. Kifer, W. Kim, Y. Sagiv, Querying object-oriented databases, *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, San Diego, CA (June 1992) pp. 393–402.

[24] W. Kim, A model of queries for object-oriented databases, *Proc. 15th Int. Conf. on Very Large Databases*, Amsterdam (1989) pp. 423–432.

[25] S. Konomi, T. Furukawa, Y. Kambayashi, Super-key class for updating materialized derived classes in object bases, *Proc. 3rd Int. Conf. on Deductive and Object-Oriented Databases* (December 1993).

[26] H.A. Kuno, E.A. Rundensteiner, Using object-oriented principles to optimize update propagation to materialized views, *Proc. 12th IEEE Int. Conf. on Data Engineering* (1996) pp. 310–317.

[27] E.A. Rundensteiner, A classification algorithm for supporting object-oriented views, *Proc. ACM Int. Conf. on Information and Knowledge Management, Maryland* (November 1994) pp. 18–25.

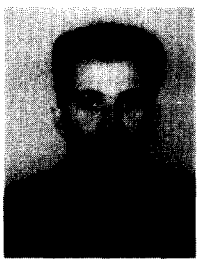
[28] E.A. Rundensteiner, A methodology for supporting multiple views in object-oriented databases, *Proc. 18th Int. Conf. on Very Large Databases*, Vancouver, BC (August 1992) pp. 187–195.

[29] C.S. Santos, Design and implementation of object-oriented views, *Proc. 6th Int. Conf. on Database and Expert Systems Applications*, London (September 1995) pp. 91–102.

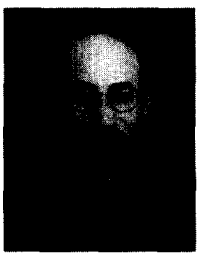
[30] M.H. Scholl, C. Laasch, M. Tresch, Updateable views in object-oriented databases, *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases*, Munich (December 1991) pp. 189–207.

[31] A. Segev, W. Fang, Optimal update policies for distributed materialized views, *Management Science* 37(7) (July 1991) pp. 851–870.

[32] M. Stonebraker et. al., On rules, procedures, caching and views in database Systems, *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Atlantic City (May 1990) pp. 281–290.
[33] G. Wiederhold, Views, objects and databases, *IEEE Computer* 19(12) (December 1986) 37–44.
[34] J. Yang, J. Widom, Maintaining temporal views over non-temporal information sources for data warehousing, *Proc. Int. Conf. on Extending Database Technology* (March 1998) pp. 389–403.
[35] F. Yazar et al., The Development of an object-oriented database management system, *Proc. 7th Int. Symp. on Computers and Information Sciences*, Kemer-Antalya (November 1992).
[36] Y. Zhuge et al., View maintenance in data warehousing environments, *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, San Jose (May 1995) pp. 316–327.
[37] Y. Zhuge, H. Garcia-Molina, Graph structured views and their incremental maintenance, *Proc. 14th IEEE Int. Conf. on Data Engineering* (1998) pp. 116–125



Reda Alhajj received his B.Sc. degree in Computer Engineering in 1988 from the Middle East Technical University, Ankara, Turkey. Later, he obtained his M.Sc. degree and Ph.D. in Computer Engineering and Information Sciences from Bilkent University, Ankara in 1990 and 1993, respectively. Dr. Alhajj was prompted to Associate Professor in 1995 by the Turkish Institute for Higher Education. He served as an Assistant Professor in the Information Systems Department at King Saud University, Saudi Arabia in 1993–1996. After that, he spent a year at Bilkent University. Currently he is with the Sultan Qaboos University in Oman. He published more than 30 papers in refereed international journals and conferences. Reda Alhajj's primary work and research interests are in the areas of query languages and query processing, view maintenance, data models, and index structures in object-oriented databases, multi-agent based query processing, multimedia databases, re-engineering of legacy systems, temporal databases and pattern recognition.



Ashraf Elnagar received B.S. and M.S. degrees from Kuwait University in 1986 and 1988, respectively, and a Ph.D. from the University of Alberta, Canada, in 1993. All degrees were earned in Computer Science. Dr. Elnagar served as a faculty member of the Department of Computed Science at PAHET, Kuwait, from 1988 to 1990. He also spent one academic year (1993/1994) at the University of Alberta, Canada, before joining the Department of Computer Science at EQU (Sultan Qaboos University), Oman, in January 1995 as a faculty member, until August 1998. Since September 1998, he has been with the Department of Computer Science at the University of Sharjah. His research interests include robot motion planning, pattern recognition, object-oriented views, and data re-engineering. Dr. Elnagar was the recipient of one of the NSERC post-doctoral fellowship awards in 1994.