Supporting Views in Object-Oriented Databases

Marc H. Scholl and H.-J. Schek

ETH Zurich, Dept. of Computer Science, Information Systems – Databases
ETH Zentrum, CH–8092 Zurich, Switzerland. E-mail: {scholl,schek}@inf.ethz.ch

1 Introduction

Relational database systems provide a powerful abstraction mechanism: any query, since it returns a relation, can be used to define a *view*, that becomes a derived (or virtual) relation. Views are defined by a statement such as "**define view** <*name*> **as** <*query*>." Views can be used to tailor the global database schema.

- Query formulation is simplified, if frequent subexpressions are predefined.
- Application programs are insulated from changes to the underlying schema.
- Information can be restructured to better suit an application programs' requirements.
- Derived information is kept consistent with base data.
- Access restrictions (for authorization) can be enforced by hiding data.

In contrast to base relations, views are typically not stored permanently, but rather computed on demand. Queries to view relations are modified by query substitution so as to operate on the underlying base relations. Therefore, any update to a base relation propagates automatically to all views defined over it. Conversely, view relations can not be updated freely, since it is often ambiguous how to trace view updates back to updates of base tuples. Typically, only views containing the key of their (one) underlying base relation can be updated.

The view mechanism should also be offered by the next generation, object-oriented DBMSs. Object identity (OId) will alleviate the view update problem. If objects are identified independent of the values associated with them, it is possible to propagate view updates back to base objects.[1] If we want to define views as in relational DBMSs, by queries, this has important consequences on the query language. Most importantly, queries will have to preserve object identity. The other possibilities discussed in the literature are object-generating queries and queries returning data values (e.g., relations). We will restrict ourselves to queries returning *existing* objects. Otherwise, updates to query results would not propagate to the original objects. If queries return original objects, the main questions are: How can we allow restructuring operations (if they change the type of objects)? Is it possible to use query operators similar to the relational projection or join, and still have queries deliver base objects? What flexibility is needed in the type system, and can we still apply (some) static type checking?

These issues are addressed in this brief overview of the work performed in the COCOON project at ETH Zurich [8, 9]. We first discuss *object-preserving query semantics* of a generic object-oriented query language in the style of a relational algebra. Then we show how views defined by such query expressions can be updated: We elaborate on certain fundamental properties of the object model, such as the separation between types and classes, multiple instantiation (an object may be an instance of several types at the same time), and multiple class membership (an object may be a member of several classes at the same time).

2 Terminology

We use the object-function model called COCOON [9], that has been developed as an evolution from and generalization of the nested relational model [7]. The IRIS model [11] with its roots in DAPLEX has similar features. Objects are pure abstractions, all data (or objects) are associated to them (as "state" or "related" objects) by functions.[2] This means that "attributes", "components", or "instance variables" are not distinguished from "derived" or "computed" values. Similarly, we can often neglect the difference between (retrieval) functions and (update) methods, and treat them in the same way. We use the following terminology:

- *Objects* are instances of abstract types, specified by their interface operations. *Data* are instances of concrete types (e.g., numbers, strings) or constructed types, such as tuples or sets [2, 9].

[1] Formally, object identity is a prerequisite for updates: without OId we could only replace, but never modify data.

[2] In implementation terms, this means: the object itself is just an identifier. The "state" of the object is the collection of return values of all functions defined on it (that map the OId to other objects, i.e., OIds , or values).

- *Functions* are either retrieval functions or update methods. They are described by their name and signature (that is, domain and range types). Functions may be set-valued.
- *Types* are described by their name and the set of functions that are applicable to their instances. Types are arranged in a *subtype hierarchy*, where subtypes inherit functions from their supertypes. Objects can be instances of more than one type at the same time ("multiple instantiation").
- *Classes* are typed collections of objects (sometimes also called "type extents"). Classes are arranged in a *subclass hierarchy* that is exactly the set inclusion between the sets of objects they represent. Objects are "members of" classes, possibly more than one at a time ("multiple class membership"). Particularly, superclasses contain all members of their subclasses.

We note that the separation of types and classes is essential. In the relational model, a type corresponds to a relation's schema (the structure), a class to a relation's extent (the set of tuples). At first glance, classes could be viewed as (persistent) sets of objects. However, a set is data, whereas a class is an object [2, 9]. That is, it has an identity that is independent of that set of class members. Each class object is an instance of the type *classtype*, which has (among others) two important functions: For each class *C*, *membertype(C)* returns the associated type, and *extent(C)* returns a set of objects of this type, the class members. Class extents are polymorphic sets: member objects may be instances of many different types. However, they are uniform in that every member is (among others) an instance of *membertype(C)*. In the query language, where we use classes as arguments, type checking is based on this unique member type.

An important additional feature distinguishes our model from others: class predicates. These are usually found in knowledge representation (classification) languages such as KL-ONE [4, 3]. Our classes may be constrained by a predicate that must be satisfied by all members of the class. We distinguish two cases: class predicates may be only necessary or necessary and sufficient conditions. Class predicates can serve several purposes: first, they allow the specification of integrity constraints (necessary predicates). Second, they are our means of separating compile-time type checkable aspects of object types from run-time checks: the former are part of type definitions (e.g., function signatures), the latter are expressed as class predicates (e.g., cardinality restrictions for set-valued functions). Third, and most importantly, class predicates can be used to handle updates to (selection) views: if class predicates are necessary and sufficient conditions, then all (common) members of the superclass(es) that satisfy the predicate are *automatically classified* into the subclass. Conversely, if objects in a class (due to updates) no longer satisfy the class predicate, they are re-classified (recursively) to belong to the superclass only.

As an example, consider two classes *Persons* and *Adults*, both of the same type *persontype*. We know that a person is an adult, if and only if his or her age is over 17. So we define a necessary and sufficient class predicate for class *Adults* that will automatically include members of the *Persons* class into the subclass whenever their age is 18 or more. The system should automatically remove persons from the subclass (and keep them in the superclass only) whenever their age is changed to a value below 18. In COCOON, this situation is represented as follows:

define type *persontype* **isa** *objecttype* = *name*: string, *age*: integer, ...;
define class *Persons*: *persontype* **some** *Objects*;
define class *Adults*: *persontype* **all** *p:Persons* **where** *age(p)* > 17;

Type definitions list the (set of) supertypes (*objecttype*, the predefined top of the type hierarchy in our example) and the applicable functions with their range types. Class definitions include the membertype and the (set of) superclasses (*Objects*, the predefined top of the class hierarchy, for *Persons* and *Persons* for *Adults*). The optional predicate (none is present for the class *Persons*), is a necessary condition in case of a **some** qualifier, and necessary and sufficient in case of an **all** qualifier. Therefore, in all valid database states, the extent of class *Adults* will always be exactly those members of class *Persons* for which the *age* function returns a value over 17. Notice, that this class definition of *Adults* is just what we would expect from a (selection) view defined over the *Persons* class.

3 Object-Preserving Query Semantics

We use a *set-oriented* query language similar to relational algebra, where the inputs and outputs of the operations are *sets* of objects. Hence, query operators can be applied to extents of classes, set-valued function results, and query results. Many such object-oriented algebras have been proposed in the literature. We can distinguish three approaches to the exact semantics of queries, depending on what the result of queries are:

1. "*Relational semantics*": query results are data values, not objects. For example, every query may return a set of tuples containing some values describing properties of objects. This semantics is useful for generating query outputs (we do not want to deliver objects, or OIds, to the user), but it is not suited for the definition of (updatable) views, since object identity is lost, so updates make no sense.
2. "*Object-generating semantics*": queries generate new objects. The states of the new objects are (partial) copies of the states of qualifying objects. Again the problem is how to propagate updates back to the original objects. This kind of query semantics is motivated by object models that do not allow objects to be instances of more than one type (class). If the type ("structure") of objects is modified by the query (e.g., a projection), the result has to be a set of new objects.
3. "*Object-preserving semantics*": queries return (some of) the input objects. As an immediate consequence of type-changing query operators, such as projections or "joins", we have to allow multiple instantiation. Multiple class membership is a consequence of making query results classes. This semantics for queries allows the application of methods and generic update operations to results of a query, since these contain base objects.

To define updatable views by means of queries, we should opt for object-preserving operator semantics. Otherwise, one must play some implementation tricks when voting for other query semantics in order to provide updatable views. For example, one can internally keep the original OIds together with the new objects. However, object-preserving query semantics is the cleaner concept. Then views are additional (virtual) classes that need to be positioned in the class hierarchy, and their membertypes need to be positioned in the type hierarchy. The objects in these view classes are base objects.

In the sequel, we give a brief overview of the COOL query language and its semantics in terms of result types and extents. In COOL, operands are sets of objects, the operators are the relational algebra operators with appropriate extensions (syntactically, operands may be classes: formally, the operands are the *extents* of the classes). Query results are also sets of objects. View definitions introduce new (virtual) classes, whose extent is defined by the query. For views defined by each of the basic COOL operators, we describe what the membertype and extent is, and how these are positioned in the type and class hierarchies. For ease of presentation, assume that all views are defined over base classes. In general, views may also be defined over other views, or by composite queries (see [8] for a detailed exposition). In the following, let *C* be a class with member type *T*.

Selection (**define view** *V* **as select** [*P*] (*C*)). The view class *V* is a subclass of the base class *C*, with the same member type *T*. We now have two classes, *V* and *C*, of type *T*. The extent is the subset of *C*-members satisfying *P*. In fact, the effect of the view definition is precisely the same as if class *V* were defined with a necessary and sufficient class predicate: "**define class** *V*:*T* **all** *C* **where** *P*".

Projection (**define view** *V* **as project** [f_1, ..., f_n] (*C*)). The view class *V* is a superclass of *C*. The member type, say *T'*, of *V* is a supertype of *T* (less functions are defined, only those listed in the projection: f_1, ..., f_n), the extent of *V* is that of *C*. The effect is the same as a schema definition containing a statement "**define class** *V: T'* **all** *C* **where true**".

Extend (**define view** *V* **as extend** [f_i:=<$expr_i$>,...] (*C*)). Projection eliminates functions, extend defines new derived ones. <$expr_i$> can be any legal arithmetic, boolean, or set-expression. The view *V* is a subclass of *C*: their extents are the same and the member type of *V*, say *T'*, is a subtype of *T* (it has the old functions plus the new ones). The effect is the same as "**define class** *V: T'* **all** *C* **where true**".

Set operations. As the extent of classes are sets of objects, we can perform set operations as usual. Their effects on class extents is their standard set theoretic semantics. Due to the polymorphic type system, we need no restrictions on the operands' member types (ultimately, all objects are instances of *objecttype*). The member type of the result, however, depends on the operands' types: A union view is a common superclass of the base classes, whose member type is the lowest common supertype (in the type lattice) of the input types. Difference views are subclasses of their base class with the same membertype; finally, an intersection view is a common subclass with a member type that is the greatest common subtype of the input types.

4 View Updates

Beyond *type-specific* update operations (i.e., methods), we provide a collection of *generic* update operators to facilitate set-oriented processing. First, there is a set-iterator for updates, **update** [*m*] (<*set-expr*>), that take as

arguments a set of objects and an update operation, *m*, to be performed on all elements. The other generic update operators are **insert** and **delete** for creating and destroying objects, **add** and **remove** for including and excluding existing objects into/from sets (in contrast to **insert** and **delete**, **add** and **remove** have no effect on the existence of the objects), and **set** to assign values (data or objects) to functions.

For type-specific update operations (methods), there are no restrictions whatsoever on view classes and all methods included in the view's membertype can be invoked. Notice, that type-changing operations, such as projections, deal with update methods and retrieval functions in the same way: a projection list includes the methods that shall be visible in the view! For the generic update operators of COOL, the semantics of their application to view classes is always defined to be exactly the same as if the view class were defined as a base class with a necessary and sufficient class predicate ("**define class** *V*...**all**...**where**..."). Therefore, the foundation of our update semantics is automatic classification: updating objects may cause the objects to dynamically change class memberships. The alternative solution of disallowing all object modifications that cause class predicates to change their truth value is too restrictive.

Consider the following scenario: Let a class *Persons* have a subclass *Myfriends*. Certainly, we can not express a sufficient predicate on persons to decide who are my friends. So, we need to tell the system explicitly, by the **add** operation, which persons to put into that subclass. Suppose there is another subclass, *NewYorkers*. Obviously, this subclass can be defined with a (necessary and) sufficient predicate, namely "**define class** *NewYorkers: persontype* **all** *Persons* **where** *addr*='NewYork'". Alternatively, we could define *NewYorkers* as a view by the statement "**define view** *NewYorkers* **as select** [*addr* = 'NewYork'](*Persons*)". In any case, we expect from the system to (i) automatically add a *Persons* member to *NewYorkers*, if the *addr* function is set to New York, and (ii) to remove an object from *NewYorkers*, if *addr* is set to some other value. The latter is also true if we apply the update to class *NewYorkers*. On the other hand, suppose we add some person object *p* to *NewYorkers* explicitly (by using the generic **add** operator). If the person fails to have a New York address, the update will not succeed, because the necessary class predicate is not fulfilled.

In general, due to the dynamic reclassification of objects based on class predicates, we need to apply only few restrictions to view updates, since most "exceptions" are detected during the evaluation of class predicates. An example for updates that are disallowed is the assignment to derived functions (e.g., in extend-views), the insertion/addition into union views (that could only be allowed if the two base classes have a discriminating predicate, because otherwise we can not disambiguate the insertion), or—the symmetric case—removals from intersection views. Details for each kind of views and update operations are discussed in [8].

We did not mention join views yet. COOL has no join operator. We can express the same semantics by the **extend** operator. Instead of joining two classes, we extend one of them by a new function. The other one may automatically be extended by the inverse function. The new function returns, for each member of that class, the set of "join partners" from the other class [9]. The derived function is defined using a selection applied to the other class, where the predicate depends on the current member of the first class. For example, a view over *Persons* that shows for each person the *neighbors*, that is, those persons living at the same address is defined as

define view *NeighborPersons* **as extend** [*neighbors*:=**select**[*addr(n) = addr(p)*](*n:Persons*)](*p:Persons*).

This way of expressing joins is object-preserving, *NeighborPersons* contains base objects, namely all members of *Persons*. Therefore, we can also update "join views": we can modify all information about the person objects and their neighbors, the only restriction is that we disallow setting the *neighbor* function to a new value. We can, however, change its value indirectly by modifying persons' addresses.

5 Fundamental Properties

The following properties of the query language have been essential for the view definition capability and the view update semantics. If some of these properties are not met by a language, our solutions will fail, partly or completely. Thus, the results we obtained are not bound to the COOL language, but to these properties.

Object preservation is the central concept. It is crucial for a view definition facility. Object preserving operator semantics means that the results of queries are *existing* objects from the database, in contrast to object-generating (results are *newly generated* objects) or tuple-generating (results are *data*, not objects) semantics.

The *type/class separation* is a consequence of object preservation: if both projections and selections are to preserve objects, and if composite select/project queries are permitted, we need this separation in order to connect the view class properly with the base class. The position of query results in the type and class hierarchies have to be less precise without this distinction (see [6], where all query results are direct subclasses of "OBJECT"). Furthermore, no operation changes both, type and extent, except for union and intersection of two classes with differing types. So, the separation is a clarification of distinct concepts.

Multiple instantiation and multiple class membership are other consequences of object preservation: since we have type-changing operators (project, extend) all objects in their results "acquired" a new type. If we consider objects in results of queries as being members of the result class as well as the input class(es), we can treat updates to query results in the same way as updates to stored classes and the updates propagate automatically.

Dynamic reclassification during updates: Automatic classification functionality known from AI systems becomes necessary when we take into account, that objects can dynamically gain and loose types during their life time and that changes of an existing object can make it a member of a more specific class (because now it satisfies it's class predicate) or a more general one (if the class predicate of it's current class is violated by the update). Reclassification is the central concept in our view update semantics. While classification (predicate subsumption) is undecidable in general, we try to identify tractable predicates. Furthermore, *re*classification—relative to an original class and a specific update operation—is simpler than classification in general.

6 Related Work

Recently, there have been other proposals for view support in ooDBMSs [1, 5, 10]. These are different from our approach in that we use the standard way of defining views by nothing else than query language expressions. They either introduce special view definition features that duplicate parts of the query language capabilities [1] or use other facilities of their systems. FUGUE [5] uses type hierarchies for information hiding: the user can implement a new type for the view that uses 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. POSTGRES [10] uses the rule system to simulate views. Derived tables (views) can be defined by rules and other rules may define specialized update semantics for them.

Acknowledgement. COOL's view mechanism was developed together with Christian Laasch and Markus Tresch. We thank Bharat Bhargava for critical comments on a draft.

References

[1] S. Abiteboul and A. Bonner. Objects and views. Manuscript, INRIA, Dec. 1990.

[2] C. Beeri. A formal approach to object-oriented databases. *Data & Knowledge Engineering*, 5:353–382, Oct. 1990.

[3] A. Borgida, R. Brachman, D. McGuinness, and L. Resnick. CLASSIC: A structural data model for objects. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 58–67, Portland, OR, June 1989.

[4] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[5] S. Heiler and S. Zdonik. Object views: Extending the vision. In *Proc. IEEE Data Engineering Conf.*, pages 86–93, Los Angeles, Feb. 1990.

[6] W. Kim. A model of queries for object-oriented databases. In *Proc. Int. Conf. on Very Large Databases*, pages 423–432, Amsterdam, Aug. 1989.

[7] H.-J. Schek and M. H. Scholl. The two roles of nested relations in the DASDBS project. In S. Abiteboul, P. C. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects in Databases*. LNCS 361, Springer Verlag, Heidelberg, 1989.

[8] M. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. Technical Report 150, ETH Zürich, Dept. of Computer Science, Dec. 1990. Submitted for publication.

[9] M. Scholl and H.-J. Schek. A relational object model. In S. Abiteboul and P. Kanellakis, editors, *ICDT '90 – Proc. Int'l. Conf. on Database Theory*, pages 89–105, Paris, Dec. 1990. LNCS 470, Springer Verlag, Heidelberg.

[10] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In H. Garcia-Molina and H. Jagadish, editors, *Proc. ACM SIGMOD Conf. on Management of Data*, pages 281–290, Atlantic City, May 1990. ACM, New York.

[11] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris architecture and implementation. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):63–75, Mar. 1990. Special Issue on Prototype Systems.