*Zuerst ersch. in : Proceedings / ICDT 90 : Third International Conference on Database Theory, Paris, France, December 12 - 14, 1990 / S. Abiteboul ... (eds.). INRIA. - Berlin ; Heidelberg [u.a.] : Springer, 1990. - S. 89-105. - (Lecture notes in computer science ; 470). - ISBN 3-540-53507-1*
*DOI : 10.1007/3-540-53507-1_72*

# A Relational Object Model

**Marc H. Scholl and Hans-Jörg Schek**

ETH Zürich, Dept. of Computer Science, Information Systems — Databases
ETH Zentrum, CH–8092 Zürich, Switzerland, e-mail: {Scholl,Schek}@inf.ethz.ch

**Abstract:** The relational model and its extensions are often considered incompatible with object-orientation. However, on the one hand nested relations provide the complex object features demanded by object models. Particularly, powerful query languages exploit the complex data structure while keeping the advantages of the declarative, set-oriented paradigm. On the other hand, object models provide semantically rich constructs for advanced modeling, and abstractions of operations as well as data. In this paper, we show an *evolutionary* path from relational, essentially nested relational, to object-oriented data models and query languages. Basically, allowing nested relation schemes to be recursively defined yields the necessary flexibility w.r.t. structure. The query language, i.e., nested relational algebra, carries over to this "network" model. As a first step towards the object-oriented integration of cooperative systems, different views onto the objects have to be supported. We present a powerful view definition facility that basically allows object views as well as relational views to be defined in our object algebra.
**Keywords:** Object model, object algebra, object views, nested relations, query optimization.

## 1 Introduction — Motivation

"Why isn't there an object model?" is a question raised in [26]. "How to fit round objects into square databases" is another prominent quotation [45]. These questions reflect important research directions and can be rephrased as "what are the essential differences between object-based and value-based models?", or "how can the advantages of the relational model be kept when the essentials of object-orientation are added?"

General agreement exists that the relational algebra with its set orientation, view definition facility, and algebraic optimization are on the pro side for relations. Nested relations and complex objects are extensions which try to keep these advantages [1]. On the OO side the most convincing advantage is the high level of abstraction which is obtained by considering objects as instances of abstract data types, only manipulated by functions, and by organizing the types into a generalization (semi-) lattice.

The problem is whether these concepts from both sides can be combined into one model or whether they are incompatible by their very nature. While the integration of two different approaches is an interesting task per se, leading to deeper insight and understanding of the differences and of the similarities, there is a second, practically more important aspect: Providing transformations from one system to another or viewing the same data differently depending on from what system we look at them, is important if we want to or if we have to distinguish several systems. This is necessary and important for application programmers, i.e., application specialists, who develop methods and algorithms to cooperate with the database, for database type implementors who have to integrate or extend existing types by new methods in extensible database systems [43, 11, 18, 33], or database administrators who have to integrate heterogeneous subsystems and various (often existing) applications. Especially tools for

the definition of views in a general sense are desirable, encompassing relational views, complex objects views, data definitions providing object representations for special algorithms. Even within one object model the facility to define views simply by using the result of any query as in the relational model is an open question.

In this paper we will show, how an evolution from the relational model to an object model can be achieved. Therefore cooperation is facilitated rather than manifesting the separation into value-based and object-based worlds. An evolution from the relational model to an object model has been described in [6]. However, it is crucial to start from *nested* relations rather than with flat relations. This allows to include also complex objects and to handle quite naturally set-valued functions as relation-valued attributes. We will show that the algebra for nested relations can be used almost without any change as an object algebra. Therefore, query languages for object models [13, 14, 24, 6, 29, 4, 12, 21, 42] are much closer to languages for complex objects or nested relations than to flat relational ones [31, 32, 11, 23, 28, 15].

We will describe a powerful query language and discuss its use for view definitions in this paper. We regard this as a very first step into the area of coupling heterogeneous systems in a cooperative environment. We will be able to define *object views* by expressions of our object algebra in a similar way as we are used to do with relations. In this aspect we are more general than [21] because there, most queries result in *new* objects; we can *preserve* object identity, though. We will just briefly mention how nested or flat *relational views* can be expressed in our language over objects, an aspect that is covered in a separate paper [41]. This kind of views facilitates cooperation with other DBMSs or with existing software tools.

In summarizing, the new aspects presented in this paper are:

1. An evolution from relations over nested relations to objects by adding sub-/supertypes and by allowing recursive schema definitions with reference semantics. The number of constructs is restricted intentionally; a fairly simple object algebra can then serve as the basis for query languages, for optimization, and for the transformation to the lower system layers, in our case to a nested relational storage manager, the DASDBS Kernel [33].
2. An investigation on how views can be defined on the object model by allowing any query for the definition of a view. We distinguish object preserving operations from object generating ones and discuss these two kinds when joins and projections are introduced.

The paper first describes a re-interpretation of relations and nested relations which smoothly leads to an object model in Section 2. The operations are presented in the main Section 3, together with a discussion of their suitability for object view definitions. The most relevant questions answered there are how to fit views into the object lattice. Finally, we give some examples of the benefits of building onto (nested) relational theory by examples of algebraic equivalences.

## 2 The Relational Object Model

### 2.1 From Relations to Objects

We start with an example, borrowed from [21] dealing with companies, vehicles, and persons. We will explain how a re-interpretation of relations with some additions yields an object model for this example. We show the type definitions in a "data definition language" and in a graphical notation, adapted from the semantic data model KL-ONE [8] (Fig. 1). The graphical representation distinguishes primitive types

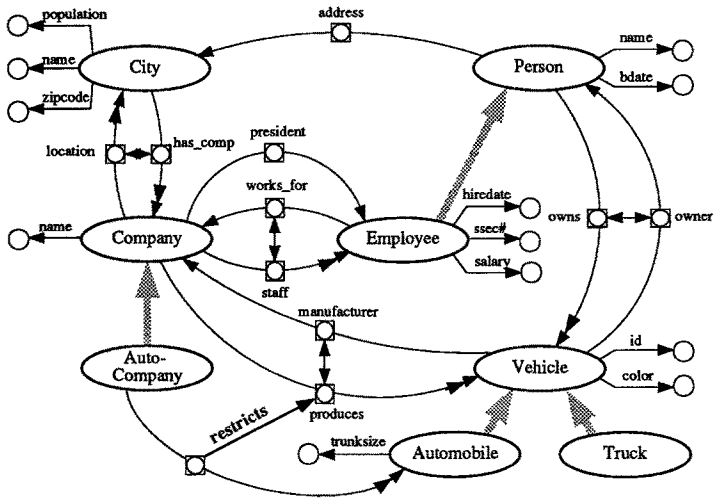


**Figure 1.** Example database in a graphical representation borrowed from KL-ONE

(circles) from defined types (ovals), both are called "generic concepts" in KL-ONE. Functions ("roles" in KL-ONE) are drawn as arrows with single arrow heads for single-valued and double arrow heads for set-valued functions. Grey arrows represent the **is_a**-relationship. The **restricts** edge connecting two roles is a specialization of functions (for auto-companies, *produces* returns only special vehicles; see below).

**type** *Vehicle* = *id* : integer,
*color* : string,
*owner* : *Person* **inverse** *owns*,
*manufacturer* : *Company* **inverse** *produces* ;

**type** *Company* = *name* : string,
*location* : **set_of** *City* **inverse** *has_comp*,
*president* : *Employee*,
*staff* : **set_of** *Employee* **inverse** *works_for*,
*produces* : **set_of** *Vehicle* **inverse** *manufacturer* ;

**type** *Automobile* **is_a** *Vehicle* = *trunksize* : integer ;

**type** *AutoCompany* **is_a** *Company* = *produces* : **set_of** *Automobile* **restricts** *Company.produces* ;

**type** *City* = *name* : string,
*zipcode* : string,
*population* : integer,
*has_comp* : **set_of** *Company* **inverse** *location* ;

**type** *Person* = *name* : string,
*bdate* : date,
*address* : *City*,
*owns* : **set_of** *Vehicle* **inverse** *owner* ;

**type** *Employee* **is_a** *Person* = *hiredate* : date,
*ssec#* : integer,
*salary* : integer,
*works_for* : *Company* **inverse** *staff* ;

**database** *CarDB* = {**class** *Vehicle*, *Company*, *Automobile*, *AutoCompany*, *City*, *Person*, *Employee*...};

The above definitions clearly are not table definitions for flat relations. The definition of *Vehicle*, for instance, in order to be a 1NF relation would have to show the *Company* key (say, *name*) as a foreign key instead of (a reference to) the *Company*. The *Company* definition looks more like a nested relation than a flat one, because *location* looks like a relation-valued attribute showing the *City* information of cities where the company is located. Also, *produces* "holds" the set of vehicles produced by the company. In addition, a kind of referential integrity constraint is given by the "inverse" keyword requiring that the company producing a vehicle can also be seen if we retrieve the *manufacturer* "attribute value" of that vehicle.

All this seems to be very much in-line with nested relations and complex objects, but there is an important difference: considering *Vehicle* and *Company* as schema definitions of two nested relations, we see that the "schema" definitions are mutually recursive. The *Company* type is defined in terms of *Vehicle* and vice versa. This is not allowed in the nested relational models proposed so far, but we will allow it now. The essential idea is to adopt "reference semantics" [27] when (object) variables of these types will be used in assignments and in comparisons. Thus, we are able to describe network data models instead of hierarchical data only. It is interesting to note that a recursive data model has already been proposed in [22] for similar reasons. By adopting the interpretation of a tuple as an instance of an abstract data type, we need only little changes in the algebra for nested relations and consequently in related query languages. For instance, we add a type test predicate as a consequence of subtyping (**is_a**).

As an introductory example let us identify all companies producing at least one red vehicle: This is a usual nested selection:

**select** [ $\emptyset \neq$ **select** [ *color*='red' ] (*produces*) ] (*Company*)

Or suppose we want to produce a report which shows for every company the name and the zip-codes of the cities where the company is located (**extract** is a particular form of (nested) projection, see below).

**extract** [ *coname* := *name*, *cityinf* := **extract** [ *zipcode* ] (*location*) ] (*Company*)

We will discuss all operations in detail in Section 3, but these preliminary examples already demonstrate that the basic structure of nested relational languages is preserved. Re-interpreting (nested) tuples as instances of an abstract data types is fairly natural. We can even imagine that a number of nested relational views are contained within these type definitions, such as

*Companies*(*name*, *location*(*zipcode*), *president*, *produces*(*id*,*color*))

or *Cities*(*zipcode*, *has_comp*(*name*, *president*, *produces*(*id*,*color*)))

or *Cities*(*zipcode*, *has_comp*(*name*, *location*(*zipcode*, *has_comp*(*name*, *location*(*zipcode*))))))).

Our algebra, in fact the $NF^2$ algebra as shown in [35], will enable us to dynamically create all these different representations (views). Things become slightly more complicated when the **is_a**-graph is considered in connection with updates and with views.

## 2.2 The Object Model

Essentially, our object model is an object-function-model (cf. [47, 13, 12]). Its constituents are

- objects,
- functions,
- types,
- classes and variables.

**Objects** are instances of abstract data types (ADTs). Objects can be manipulated only by means of their interface, a set of functions. Particularly, object identity is maintained by the system (we can think of internal invisible identifiers, OIds, used for representing objects).

We may distinguish a few primitive object types ("literal" objects in [47], "concrete" objects in [20]). They are considered elementary, that is, they are not defined using other types (leaves of the type graph). Essentially, these primitive objects have a name known outside the system [7].

**Functions** are defined by giving a name, domain, and range. Domain and range of functions are given by names of *types*. Functions can be single-valued or set-valued. In the latter case, lower and upper bounds on the number of result values can be specified. For instance,

**function** *f*: *A* → **set_of** [*n*:*m*] *B*;

defines a function *f* that returns between *n* (minimum) and *m* (maximum) objects of type *B*, when applied to an object of type *A*. Functions are a uniform abstraction of "attributes" and "relationships" of classical data models.

A useful feature is the capability of defining inverses of functions. In the example above, we can imagine a function *g* mapping back from objects of type *B* to those object(s) of type *A* that relate to them via *f*. In this case the function definition of *f* above would be changed to "**function** f: A → **set_of** [n:m] B **inverse** *g*". This is an integrity constraint on legal values of *f* and *g*; the constraint will be enforced by the system.

**Types** describe the common interface of all instances of that type, that is, the collection of applicable functions. The definition of a type consists of a type name and, optionally, a list of functions. Functions with more than one argument will usually be defined separately from types. For example, the following two forms defining a type *A* with two functions *f* and *g* are equivalent:

| | |
|---|---|
| **type** *A*= | **type** *A*= ... ; |
| *f*: **set_of** [*n*:*m*] *B*, | **function** *f*: *A* → **set_of** [*n*:*m*] *B*; |
| *g*: *C*; | **function** *g*: *A* → *C*; |

**Subtyping.** If type *A'* is defined as a *subtype* of type *A* (**type** *A'* **is_a** *A* = ... ;), then all instances *a'* of *A'* are also instances of *A*. Therefore, *a'* may occur in all places where an instance of *A* is required (substitutability). If the subtype relationship **is_a** (generalization) holds between supertype *A* and subtype *A'*, the following must be true:

- functions(*A*) ⊆ functions(*A'*); that is, all functions defined on *A* are defined on *A'* too; and
- functions on *A'* may be more restricted than they are on *A* (that is, domain and/or range of the function in *A'* may be subtypes of those in *A*, and/or cardinality restrictions can be sharper) and/or a different implementation may be assigned to them (*overriding*).

The first condition reflects the Cardelli-like [10] semantics of subtyping. The specialization of functions in the second condition is similar to "role restrictions" and "role differentiation" in KL-ONE [8] (see the *AutoCompany* example). Subtyping defines a partial order "$\preceq$" on types. Typically, this ordering forms a (semi-) lattice of types, such that for types *A*,*B* their lowest upperbound is always defined. The top element of the lattice is the most general type "Object" (therefore, all instances of defined types in

the database are also instances of "Object"). We allow multiple inheritance, that is, a type may have more than one (direct) supertype.

As a consequence of subtyping, objects may be an instance of more than one type. For an object $o$, each of its types provides a certain view on the object [19]; if viewed as an instance of type $T$, then only the functions defined in $T$ are applicable to $o$. The algebra is strongly-typed and allows *static type checking* [9]. We assume a (type test) predicate $T$ to be defined for each type $T$. $T(o)$ is true iff object $o$ is an instance of type $T$. Type predicates can serve as selection predicates, or to guard (sub-) type-specific operations [48].

**Classes.** We distinguish types from classes, following e.g. [3, 7]. A class is a set of objects that is associated with a type. There may be more than one class of a type (for instance, as the result of defining a selection view; see below). Notice that object sets are homogeneous in the sense that one single object type is associated with the set (class), even though objects in the set may also be instances of several other types. A subclass relationship can be defined on classes as a combination of the subtype relationship among their types and the subset relationship among their extents: class $C'$ is a subclass of superclass $C$, iff

- type($C'$) $\preceq$ type($C$); (that is, the underlying type is a subtype), and
- extent($C'$) $\subseteq$ extent($C$); (that is, set $C'$ is a subset of set $C$).

Notice that, unlike e.g. [14, 24, 21], we consider a class $C$ as the set of objects in $C$ and all of its subclasses.

In a class definition, we specify a class name and the element type, for instance:

**class** *Employees:* **set_of** *Employee*

As a standard naming convention we use the type name also as a class name denoting the set of all instances of that type, e.g., the above is equivalent to "**class** *Employee*" (cf. the $O_2$-option "with extension" for type definitions). In our query language, we can use variables in the usual programming language sense, that is, as temporary names for objects or object sets.

## 2.3 Comparison with Relations

Our claim in the beginning was, that little reinterpretation would suffice to "turn the nested relational model into an object model", that is, allow nested relational-style operations to be applied to objects in the usual way. Table 1 summarizes the correspondence between nested relational and object models. Such correspondences have already been observed between relations and objects, see e.g. [6, 46]. However, it is essential to start from *nested* relations when moving towards objects, for relation-valued attributes reflect set-valued functions in object models, and nesting of algebraic subexpressions corresponds to function composition. This explains why object query languages proposed to date are in the spirit of *nested* relations [2, 7].

Apart from the important organization of types in an **is_a** graph which will be visible when views and updates are discussed in the next section, the only essential change is that now objects are abstract (instances of ADTs). Therefore, references are no longer by means of (user-controlled) 'foreign keys' like in relations, but a built-in feature of the model (by object identity, which is system-controlled). However, we need not explicitly introduce OIds or surrogates, as done in LauRel [23], for instance. Rather we assume implicit OIds that are transparent at the model interface, following [7] and object oriented languages, such as Eiffel [27], or functional data models, e.g., PDM [13, 12].

| Nested Relational Model | Object Model |
|---|---|
| nested relation | class |
| schema of a relation ... set of attributes A | type ... set of functions f |
| extension of a relation ... set of tuples t | extension of a class ... set of objects o |
| tuple t ... maps attributes to values | object o ... instance of an ADT |
| attribute value t(A) ... atomic, or | function value f(o) ... primitive type, or |
| ... tuple-valued, or | ... abstract type, or |
| ... relation-valued | ... set_of some type |
| domains ... finite, defined bottom-up | "domains" ... infinite, not defined explicitly |

**Table 1.** Nested relational versus object model concepts

## 3 Object Algebra and View Definition

**Variables and Assignments.** In order to be able to refer to objects and to results of previous algebra expressions, we allow the use of variables and assignments. Variables are used as temporary names ("handles") for objects or object sets. They have to be declared with their type in the database sublanguage—either explicitly or by a query—, such that compile-time type checking applies to variables too. For example,

**var** *My_Cars* : **set_of** *Automobile*,

**var** *My_Chevy* : *Automobile*

declares variables of type **set_of** *Automobile* and *Automobile*, respectively. Hence, the set variable can, for instance, keep the result of a selection on the database class *Automobile*; the object variable can, for example, be assigned the result of an object creation in order to identify the new object in subsequent operations:

*My_Cars* := **select** [ *P* ] (*Automobile*);

*My_Chevy* := **insert** [ ... ] (*Automobile*).

In the example, the query defines *My_Cars* to have as value a subset of the persistent objects from the input class.

All updates that are eventually applied to variables are propagated to the persistent database objects they hold as values (at end-of-transaction). In this sense, objects are not *copied* into the variable, rather the variables serve as temporary names for the object or the object set. Additional operations (clone, equal) may then be useful for (shallow) copying and (shallow) equality tests between objects. These notions have been introduced in object-oriented languages, e.g., in Eiffel [27], and in [5, 14, 24]. They can easily be derived from the operations defined here.

**Object Views.** It is our goal to allow queries to serve as view definitions. That is, if <*query*> is a well defined query, then

**define view** <*name*> **as** <*query*>

is a view definition. We distinguish two kinds of views: *object views* and *relational views*. For object views, after execution of this statement, <*name*> will appear as a (persistent) class of the database, pretty much like the other classes. The "extension" of an object view, however, is usually not stored explicitly, but rather computed upon demand from the extent-defining query. An important problem, as noticed in [21], is to define the position of the view class in the object lattice. That is, the relevant questions are (i) what is the type of the result objects, (ii) where is this (usually new) type located relative to the input

types in the **is_a** graph, and (iii) what is the relationship between input and output classes (i.e., are the result objects new or pre-existing). As one of our primary goals in the algebra is suitability for view definitions, unlike [21, 42], we will mainly consider *object preserving* and not *object creating* operators. For object preserving operations, the resulting objects shall be identical with input objects so as to allow subsequent operations on them (e.g., updates) to propagate to the stored objects. Queries can thus be used as view definitions, as the argument of (set-oriented) update operations, or to assign their result to (set) variables of the appropriate type for later use.

Like a relational algebra, the object algebra consists of some independent basic operators that can be combined to define derived operators. We will concentrate on basic operators in this paper. The (intuitive) semantics of the operators introduced is obvious from the relational context. The algebra is a set-oriented language, that is, its inputs and outputs are *sets* (of objects). If we—syntactically—apply operators to a class, the semantics are to operate on its extent, i.e., a set. Therefore, all query results are sets. In the sequel, however, we discuss the positioning of the results in the class lattice, since we are interested in where a view class defined by a query belongs. In fact, the view definition "**define view** V as <query>" turns the query result (a set) into a class.

## 3.2 Selection

Selection in the relational model selects a subset of the tuples in the input relation. The output relation consists of those tuples satisfying the selection predicate, the schema of the output relation is the same as that of the input. Essentially, selection in the object algebra does the same: Selections require a predicate that is evaluable on the type of the input class. It returns the subset of the objects in the input class satisfying the predicate. The selection predicate is a boolean-valued function defined on the type of the input. Basic comparators are built-in (such as "=" for *all* types, and possibly others, like "<", for *some* types), arbitrary further (type-specific) comparison operators, i.e., boolean functions, can be defined by type implementors or users.

**Selection views are a subclass of the input class** with the same type, but only a subset of the instances. Like in the nested relational model, selection predicates may include nested subqueries on set-valued functions and/or other classes. For instance, we can select companies having at least one branch in New York:

*NYComp* := **select** [ $\emptyset \neq$ **select** [ *name* = 'New York' ] (*location*) ] (*Company*).

This is a well-defined selection, since "$\emptyset \neq$ **select** [*name* = 'New York'] (*location*)" is a valid predicate, which happens to include a (sub) query. Technically, a query is a (set-valued) function mapping an input set to an output set. Thus, queries can be used in predicates as any other functions can. In order to make the query syntax more legible, we can (temporarily) assign a new name to a composite function. For instance, the above is equivalent to:

**let** *NYloc* = **select**[*name*='New York'](*location*) **in** {
*NYComp* := **select** [ $\emptyset \neq$ *NYloc* ] (*Company*) }.

Here, we defined a function *NYloc* as a (functional) composition of *location* and the 'inner' **select**. The scope of *NYloc* is limited to the query enclosed in the braces of the **let** statement (cf. the "use" statement in [31]).

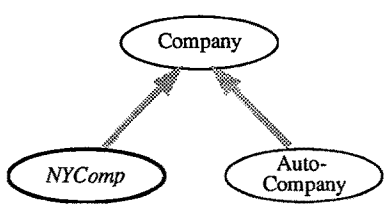


**Figure 2.** Selections create subclasses.

## 3.3 Projection

Projections in the relational model drop some columns of tables, that is, reduce the number of attributes of the tuples. Also, duplicates that may eventually result from dropping distinguishing attributes are eliminated. This operation serves two main purposes: (i) to determine the information that is to be output by a query, and (ii) to "hide" some attribute values from views such as the salary component of employee tuples).

Object projection shall behave similar to relational projection. However, duplicate elimination is not an issue, since two objects are different even if they have some (or even all) function values in common. This helps in avoiding some of the problems with (updates on) projection views over relations. We will use two operators for the two purposes. For purpose (i) the operator **extract** (as introduced in [41]) will be used. This is an object- (in fact, tuple-) generating operator, similar to the projection used by, e.g., [42, 21], or the relation-generating operations of [2, 7]. We will not discuss **extract** in detail in this paper. For purpose (ii), though, we need an object-preserving semantics of projection, this is our **project** operator. To our knowledge, none of the previous object algebras provided a means of projecting objects onto some of their functions while preserving their identity. But if we want to use projections in view definitions, we need this very feature.

Projections require a list of functions defined on the type of the input. The type of the output of a projection is a (usually new) type, a supertype of the input type, as less functions are defined on the output, namely only those listed in the projection. The objects in the input set are also elements of the output set (*object preservation*). Thus, **a projection view is a superclass of the input class.**

For example, we may wish to "hide", i.e., project out, the salary from some users by giving them a view on *Employee* that does not contain the *salary* function:

**define view** *PublicEmpl* **as**

**project** [ "all but *salary*" ] (*Employee*).

Notice that, as type checking works on the *class* level, using the *salary* function on class *PublicEmpl* would result in a (compile-time) type error, even though each employee *object* "has" both types, *Employee* and *PublicEmpl*.

If the projection list includes all functions defined on the immediate supertype of the input class, *Person*, in our example, then the new type becomes a subtype of this immediate supertype. Otherwise it becomes a subtype of a more general type.

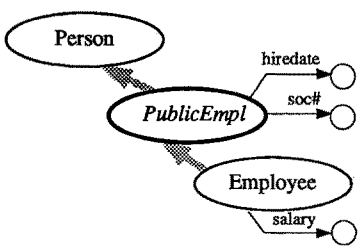


**Figure 3.** Projections create superclasses.

Users of this view operate on exactly the same objects as those in *Employee*, they can use all the employee functions, except *salary*. That is, projection "has no effect" on the instance level, it just affects object types by "hiding" some functions (like a "type cast"). Particularly, also updates can be performed on the view. These updates will affect the elements of class *Employee*, since they *are the same* as those of *PublicEmpl*.

As in the relational model, some updates will not be possible on views. For example, after creating a new *PublicEmpl* object, we can not assign a salary to the object unless we explicitly make it an *Employee*. On the other hand, relational view update problems arising from projections and duplicate eliminiation do not occur here, since projection views preserve object identity.

## 3.4 Extend

Projection, as in the (flat) relational case, eliminates functions. For views, defining some new, derived, functions can also be a very useful operation. We can, for instance, define a derived function *resp_for* for employee objects (that are presidents of some company) by the following query. The new function is the inverse of *president*; that is, for each employee, *resp_for* returns the set of companies for which he/she is president:

*PresEmpl* := **extend** [ *resp_for* :=
**select** [*president*=**self**.*Employee*] (*Company*)
] (*Employee*).

Notice that we used **self**.*Employee* as a variable being bound to the "current" employee, the selection on *Company* is performed once per object in *Employee*. An alternative would be to use a query language syntax with explicit object variables (like "Select ... From *e* in *Employee* ...", cf. [6, 41]).

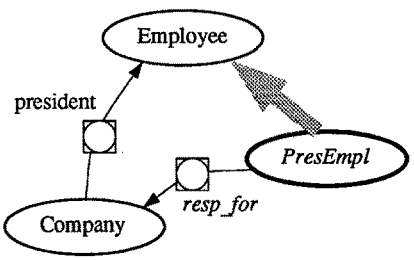


**Figure 4.** Extend defines subtypes with new functions.

Notice that *all* employees will be contained in the result (*resp_for* returns an empty set for non-president employees). If we want to retain only presidents, i.e., those for which the new function actually yields a value, we can use a subsequent selection "*resp_for* $\neq \emptyset$". In general, **extend** takes as input an object set and a (set of) function definitions. The function(s) have to be defined on the input type, that is, the functions need to be evaluable in the context of the objects in the input set (cf. "dynamic constants" in the $NF^2$ algebra [34]). As functions are added to the type of the input, the result type is a *subtype* of the input type, for all the old functions plus the new one are defined on it. The objects of the input set are preserved, that is, **extend views create a new class that is a subclass of the input class**, with the same set of instances but a new type, a subtype of the input type. Notice again, we use our definition that objects may 'have' more than one type and be a member of more than one class.

If necessary, we can make an **extend** operation automatically define the *inverse* of the new function, too. We simply append the phrase "**with inverse** <*invname*>" to the expression to give the inverse function a name of our choice.

## 3.5 Join

Here, we consider an operation comparable to the relational join. In a relational database, information about entities is often spread among several relations, such that applications heavily depend on collecting data by (natural) joins. Unlike relations, object models provide explicit means of building "complex objects" or "linking" related objects together (in our case by functions). Thus, neither in order to collect information describing complex entities nor for following relationships among entities do we need to formulate joins. Rather, we follow the predefined "links" by applying functions (this results in what

is called a query graph in [21]). However, it is often useful to provide a query facility for 'arbitrary' (sometimes called 'unstructured') joins. This supports queries relating objects via conditions that are not (directly) represented in the predefined functions.

**Object Preserving Join.** We have already seen a way of establishing new 'relationships' among objects, that is, defining new functions between them: the **extend** operator does this. So, one way of expressing 'joins' in our model is to define the required relationship as a (set of two inverse) new function(s) connecting the "matching pairs" of objects. For example, to answer a query like "employees living in a town in which some (not necessarily *their*) company is located" — which would require an (equi-) join of the two corresponding relations on the predicate "COMP.LOC=EMP.ADDR" in a relational database—, we can define a new function *LocalComp* on the employee class returning companies located at the employees' home town by the following **extend** operation:

**define view** *JoinedEmpl* **as extend** [ *LocalComp* := **select** [ *address* ∈ *location* ] (*Company*) ] (*Employee*).

The resulting view *JoinedEmpl* will be a subclass of *Employee* containing all *Employee*-objects (see above), the new function returns the set of "join partners". A simple subsequent selection with the predicate "*LocalComp* ≠ ∅" will return the qualifying employee objects, that is, *JoinedEmpl*, as defined above, is actually a (one-sided) outer join; the additional selection makes it an inner one.

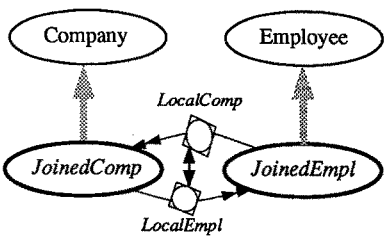


**Figure 5.** Result of "joining" by symmetric **extend**.

Adding the clause "**with inverse** *LocalEmpl*" to the definition of *LocalComp* in the **extend** operation will result in a symmetrical "join" result, since the inverse function will then connect the companies back to the employees. Technically, the symmetric **extend** operation results in *two* new classes, one new subclass for each of the input classes. Notice that, if we were interested in employees living in the same town as *their* company's location, we would have used the "*works_for*" function instead of the class "*Company*" in the **extend** operation. "Joining" objects by means of the **extend** operator is *object preserving*, that is, the result of this "join" is a new "relationship" among *existing* objects.

It appears that we do not need an explicit join operation in our algebra, the desired functionality can already be expressed using **extend**'s. This has also been observed in the context of nested relations [37, 38], where it turned out that join (or relational product) can be derived from nested projection (which corresponds to **extend**). In fact, any "join" with a predicate $P$ among classes $C_1$ and $C_2$ can be expressed as a (symmetric) **extend** operation using the predicate $P$ for a selection inside the **extend**. Hence, a derived **p-join** operator could be added based upon **extend** that perserves objects.

**Object Creating Join.** If we want to *create objects* by an explicit **join** operator in the object algebra, we are given two choices, depending on the "structure" of the result: talking relationally, shall the result be a set of pairs (like the result of a straight set-theoretic product), or do we want to "flatten" the result, such that we obtain (n+m)-tuples? Relationally, in order to retain the closure property, the result must be a set of (n+m)-tuples. Secondly, a set-of-pairs semantics has the disadvantage that joins would not be associative, that is, (R **join** S) **join** T would be different from R **join** (S **join** T). This, however, would exclude some of the most significant optimization means. The first argument does no longer hold since

we have richer structures anyway, so the clear, mathematical semantics of product, and hence join, seems to be more attractive. The second argument, however, is still worth consideration, since we do want to optimize object algebra queries in a way similar to relational algebra.

Nevertheless, let us start with the "pairs" alternative and focus on the associativity problem later: let **join** be an operator of the object algebra that takes two input classes and a predicate defined over the types of both, then an operation like

**define view** *CE* **as**

*Employee* **join** [ *address*∈*location* ] *Company*

results is a new object class *CE*, the type of which contains two functions *Employee* and *Company*. For each (new) object in the result set, these functions return the employee and the company, respectively, that 'contributed' to the result object. That is, this join creates a new type (direct subtype of "object"). The result class contains new objects (of this new type).

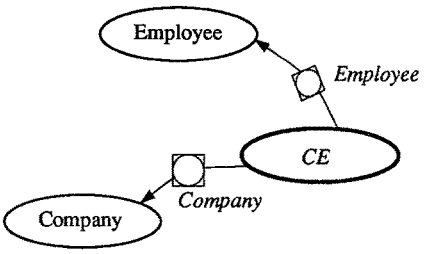


**Figure 6.** Join creating new objects.

Two functions relate the new join objects to the two input objects. The same is true in [42] for "non-tuple" input types, whereas tuple input types result in (n+m)-tuples. In [21], the join result is a set of pairs of old OIds as long as the result is not "saved" in the database (as a snapshot), if saved, new OIds are generated for the tuple components. In a way, our result is still useful as a view, since the contributing input objects can be 'reached' from the join-objects. The remaining problem is non-associativity.

The result can easily be made associative, though, by defining new derived functions (with the extend operator): one for each "attribute" of the input types of the join. By composing the function defined by the join and the "old" attribute function, we can construct an "associative result", *CE'* in our example:

*CE'*:= **extend** [*ename* := *name*(*Employee*), ... *name* := *dname*(*Company*)] (*CE*).

The result *CE'* is a class (subclass of *CE*) with a type that has (n+m+2) functions defined on it: the (n+m) functions from the two join-input classes, plus one function per input class "pointing" to the objects that resulted in the join-object (inherited from *CE*). Thus, if we want associative joins (for optimization purposes), we can use this "associative join" operation (which is derived from the former join and the extend as shown above). If this new join operation is used, the system is free to decide on join orders. It should be noted that already the non-associative join operator, and thus the derived associative one, can be defined in terms of a new type and class definition, extend and a few other operators, just like in the nested relational case. That is, if we restrict our scope of interest to the basic independent operators of the object algebra, we can disregard join anyway.

## 3.6 Other Operators

**Set operations.** We do not need to take any special care about set operators (union, difference, and intersection). As we operate on sets of objects, we can perform set operations as usual. One notable point

is the criterion for duplicate elimination (or equality determination): the mathematical notion of equality is identity in case of abstract objects. So, this is the notion that is used in the set operations[1]. Other types of equality (like shallow or deep equality) can be defined based on the values of functions defined on the objects, recursively. They may also be useful for creating new objects as "copies" of existing ones.

**Pick.** Note that we did not introduce (persistent) names for single objects. For getting hold of single objects, users have to use class names in combination with functions and selection predicates. Even selections with only one qualifying object return a (singleton) set, though. Therefore, we need an operator for converting singleton sets into the only element (that is, drop spurious set braces), we called it **pick** [41]. Notice, that **pick** is a limited version of "unnest" known from the nested relational model.

**Extract.** The operators presented so far result in (sets of) *abstract objects*. One operator of our language, however, can be used to produce sets of *tuples* (that is, values [7]): the **extract** operator [41]. Essentially, **extract** is another kind of projection that constructs tuples with one component for each element of the projection list. If, recursively, all components are either sets of tuples or primitive data values, then the output of an **extract** is a (nested) relation. Hence, we can include explicit relations [2] or generic tuple objects [42] into our model, given that we provide the standard (relational) operations on them. Also, **extract** can serve as a means of coupling an ooDBMS with other (value-based) services in a heterogeneous environment.

## 3.7 Remarks on Update Operations

Update operations are described in detail in [41]. Here we add a few remarks on updates on views and explain differences between (set) variables and views in case of updates.

**Updating Views.** Views can be used as the arguments of update operations, as they are simply names for queries. As already observed in the relational context, updating views, however, may be subject to restrictions. For example, insertions into (projection) views result in undefined attribute values in relations. In the object model, due to object identity, these problems are less severe. Nonetheless, suppose we create a new employee object in the *PublicEmpl* view defined by the **project** operation as shown in Section 3.3. Obviously, we cannot assign a salary to the new object, since this function is hidden in the view. Thus, in order to pay the new employee, we have to make him/her an instance of the more specific type *Employee* first:

*NewEmp* := [ *name* := ..., *addr* := ..., *ssec#* := ... ] (*PublicEmpl*);
**add** [ *NewEmp* ] (*Employee*);
**set** [ *salary* := 2000 ] (*NewEmp*).

A detailed discussion of the effects of all update operations on views, depending on the query operators used in the view definition, is contained in [40].

**Variables versus Views.** As we permit the use of variables (and assignments) as part of our database sublanguage, a natural question to ask is, what are the effects of update operations applied to variables? Particularly, what are the differences between set variables and views?

---

[1] this is the problem of determining uniqueness of objects as mentioned in [21]. The other two problems mentioned there, heterogeneity and scope of classes, are treated differently in our model: classes are considered homogeneous, and the extension always includes all subclasses (see above).

The idea behind variables in our language is the standard programming language concept: variables may be used to hold temporaries in a complex program interacting with the database. The idea of object sharing applies to variables too. That is, if a variable $v$ holds an object $o$ and an update operation $m$ is applied to $v$, then $o$ gets updated, and this update is visible everywhere in the database where $o$ occurs. The same is true for set variables: if a set variable $V$ holds a set of objects $\{o_1,...,o_n\}$, then all operations performed on $V$ are actually performed on $\{o_1,...,o_n\}$. Particularly, the updates are visible in all persistent classes of the database that contain some $o_i$ from the set.

In contrast to set variables, views are persistent classes of the database, that is, their scope starts with the transaction that created them and eventually ends with the (other) transaction that might drop them. In the meantime, no user or application program can distinguish views from base classes. Set variables, on the other hand, are only existent during the execution of the one database program that declared them. Their scope ends at the end of the enclosing transaction.

## 3.8 Algebraic Equivalences

The main advantage of using an object algebra in the spirit of (nested) relational algebra is that we can draw from a broad background on query processing and optimization. An extensive discussion of optimization rules is beyond the scope of this paper. In fact, rather than develop a new theory of algebraic equivalences and optimization heuristics, we try to adopt results obtained previously for the nested and flat relational algebras. Actually, in looking at first work on optimization of object algebra expressions, see [42, 44], strong similarities can be observed. For instance, commutativity of selections, distribution of selections over unions or joins, or composition of nested subqueries are algebraic properties that hold for nested relational algebra, too. Since we have already investigated nested algebra in some detail [36, 39, 38], and others have also worked on query optimization in this context (see, e.g. [1]), we hope to obtain object algebra optimization results also in an evolutionary way.

To illustrate how algebraic optimization carries over from the nested relational to the object algebra, we list a few equivalences from [36, 38], rewritten into the syntax of our object algebra:

(1) **select** [$P_1$] ( **select** [$P_2$] ($C$) )
≡ **select** [$P_2$] ( **select** [$P_1$] ($C$) )
≡ **select** [$P_1 \wedge P_2$] ($C$)

(2) **select** [ $P$ ] ( **project** [ $L$ ] ($C$) )
≡ **project** [ $L$ ] ( **select** [ $P$ ] ($C$) ), if $L$ contains all functions mentioned in $P$

(3) **extend** [ $g$ := <$expr_2$>($f$) ] ( **extend** [ $f$ := <$expr_1$> ] ($C$) )
≡ **extend** [ $g$ := <$expr_2$>( <$expr_1$> ) ] ($C$)

(4) **select** [ $P$(<*expr*>) ] ($C$)
≡ **project** [ "**all but** $f$" ] ( **select** [ $P(f)$ ] ( **extend** [ $f$ := <*expr*> ] ($C$) ) )

(5) **image** [ $f$ ] ( $C$ ), where $f: C \rightarrow$ **set_of** $B$
≡ **select** [ $\emptyset \neq$ **select** [ **self**.$B \in f$ ] ( $C$) ] ( $B$)

Equivalence (1) is the commutativity and combination of selections, (2) is commuting selection and projection, (3) is combination of subsequent **extends**, which is a transcript of rules on nested projections, and (4) shows that nesting of algebra expressions into selections is not essential, as already shown in [34]. Essentially, we can define the nested subexpression as a derived function (using **extend**) first, apply a simple selection, and finally remove the derived function, if necessary.

An operator, usually found in functional models, extends functions defined on a type $T$ to type **set_of** $T$. It is called **image** in [42], other usual names for the same operator include "map" and "apply_to_all". Let $f$: $A \rightarrow B$ be a function and $S$ be of type **set_of** $A$, then **image** [$f$] ($S$) is the image of set $S$ under the function $f$ (in the mathematical sense), i.e., **image** [$f$] ($S$) = { $f(s)$ | $s \in S$ }. Image differs from (single-function) projection in that the result is the set of function values, whereas a projection returns the set of objects in the input set, but with just this one function left. Notice that **image** is a derived operator, it can be defined by a (nested) selection as shown in equivalence (5).

## 4 Conclusions

Many object models are based on the notion of objects as abstract data types with type-specific functions (object-function-models). Usually functions may return sets of objects as well as single objects. Set-valued functions are the reason why a nested relational query language is better suited as a starting point for an object query language than a flat relational one: objects are very similar to a tuple in a nested relation, where attributes (the functions) may be single-valued or set-(of-tuple)-valued. We have shown that in fact the reinterpretation of tuples as abstract objects, the "recursive relational schema definitions" obtained this way, and the use of reference semantics for assignments and equality (i.e., identity) predicates turn the (nested) relational model into the core of an object model. That is, there is in fact an evolutionary path from relational to object models. Nested relations play the role of an intermediate stage, where nesting of query language expressions according to the nesting structure of the complex objects is already possible.

The strong similarities between query languages proposed so far for object models and complex object models is therefore quite natural. Hence, a reasonable direction of future research is to try and carry over the theoretical results obtained about completeness, complexity, and optimization of complex object (nested relational) algebras to the new object algebras. We sketched, how some equivalences of our nested relational algebra still apply. For the implementation of OODBMS, this similarity should make it easier to map an OODB interface to a complex object storage manager, since the conceptual distance between their query languages is smaller.

The additional important ingredient of all object models is the notion of subtypes and supertypes and its organization into a type/class lattice. We investigated the problem of how to fit query results into the class hierarchy. This problem has received high attention in current research. We have shown that it is crucial to distinguish object preserving operations from object generating operations. We put more emphasis on object preserving operations because they can be used to define views over the existing object base providing some potential for view updates. We introduced operators to **select** subsets of objects, to hide functions from views (**project**), to define new (derived) functions in views (**extend**), and, put particular emphasis on how to provide object preserving join semantics (by using **extend** to define new relationships between objects). In summarizing, the object algebra presented here allows arbitrary operations in view definitions because all necessary operators preserve object identity.

A prototype system including most of the described retrieval and update functions has been implemented with a SQL-like query language. In its current implementation it maps the (KL-ONE) object model to a flat relational interface (ORACLE). At present we are working on a re-implementation on top of our $NF^2$ relational DASDBS Kernel system [30, 33] to make use of advanced clustering and query processing techniques and will compare and evaluate the two solutions. The interface of this storage manager is a subset of the nested relational algebra. Therefore, we expect query optimization to benefit

from the uniformity of query representations from the object model all the way down to the storage structures. Besides algebraic optimization along the lines of nested relational techniques, we will include extensible optimization techniques like those discussed in [16, 25, 17].

## References


[1] S. Abiteboul, P. C. Fischer, and H.-J. Schek, editors. *Nested Relations and Complex Objects in Databases*, volume 361 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 1989.

[2] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 159–173, Portland, June 1989. ACM, New York.

[3] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985.

[4] F. Bancilhon. Query languages for object-oriented database systems: Analysis and a proposal. In T. Härder, editor, *Proc. GI Conf. on Database Systems for Office, Engineering and Scientific Applications*, pages 1–18, Zürich, March 1989. Springer IFB 204, Heidelberg.

[5] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. Int. Conf. on Very Large Databases*, pages 97–105, Brighton, September 1987.

[6] D. Beech. A foundation for evolution from relational to object databases. In J.W. Schmidt, S. Ceri, and M. Missikoff, editors, *Advances in Database Technology — EDBT'88*. Springer LNCS 303, March 1988.

[7] C. Beeri. Formal models for object-oriented databases. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases*, pages 370–395, Kyoto, December 1989. North-Holland.

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

[9] V. Breazu-Tannen, P. Buneman, and A. Ohori. Static type-checking in object-oriented databases. *IEEE Data Engineering*, 12(3):5–12, September 1989. Special Issue on Database Programming Languages.

[10] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[11] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A data model and query language for EXODUS. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 413–423, Chicago, IL, May 1988. ACM, New York.

[12] U. Dayal. Queries and views in an object-oriented data model. In R. Hull, R. Morrison, and D. Stemple, editors, *2nd Int'l Workshop on Database Programming Languages*, pages 80–102, Oregon Coast, June 1989. Morgan Kaufmann, San Mateo, Ca.

[13] U. Dayal, F. Manola, A. Buchmann, U. Chakravarthy, D. Goldhirsch, S. Heiler, J. Orenstein, and A. Rosenthal. Simplifying complex objects: The PROBE approach to modelling and querying them. In H.-J. Schek and G. Schlageter, editors, *Proc. GI Conf. on Database Systems for Office, Engineering and Scientific Applications*, pages 17–37, Darmstadt, April 1987. Springer IFB 136, Heidelberg.

[14] O. Deux et al. The story of $O_2$. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):91–108, March 1990. Special Issue on Prototype Systems.

[15] K.R. Dittrich, W. Gotthard, and P.C. Lockemann. DAMOKLES – the database system for the UNIBASE software engineering environment. *IEEE Database Engineering Bulletin*, 10(1), March 1987.

[16] G. Graefe and D. J. DeWitt. The EXODUS optimizer generator. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 160–172, San Francisco, May 1987. ACM, New York.

[17] G. Graefe and D. Maier. Query optimization in object-oriented database systems. In K. R. Dittrich, editor, *Proc. Int. Workshop on Object-Oriented Database Systems*, pages 358–363, Bad Münster, September 1988. Springer LNCS 334, Heidelberg.

[18] L.M. Haas, J.C. Freytag, G.M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 377–388, Portland, OR, May 1989. ACM, New York.

[19] S. Heiler and S.B. Zdonik. Views, data abstractions, and inheritance in the FUGUE data model. In K.R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, Heidelberg, September 1988. Springer LNCS 334.

[20] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

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

[22] W. Lamersdorf, G. Müller, and J. W. Schmitt. Language support for office modelling. In *Proc. Int. Conf. on Very Large Databases*, pages 280–288, Singapore, August 1984.

[23] P.-Å. Larson. The data model and query language of LauRel. *IEEE Database Engineering Bulletin*, 11(3):23–30, September 1988. Special Issue on Nested Relations.

[24] C. Lécluse and P. Richard. Modeling complex structures in object-oriented databases. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 360–368, Philadelphia, PA, March 1989. ACM, New York.

[25] G.M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 18–27, Chicago, June 1988. ACM, New York.

[26] D. Maier. Why isn't there an object-oriented data model? Technical Report CS/E-89-002, Oregon Graduate Center, Beaverton, OR, May 1989.

[27] B. Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, 1988.

[28] B. Mitschang. Extending the relational algebra to capture complex objects. In *Proc. Int. Conf. on Very Large Databases*, pages 297–305, Amsterdam, August 1989.

[29] S.L. Osborn. Identity, equality, and query optimization. In K.R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, pages 346–351, Heidelberg, September 1988. Springer LNCS 334.

[30] H.-B. Paul, H.-J. Schek, M. H. Scholl, G. Weikum, and U. Deppisch. Architecture and implementation of the Darmstadt database kernel system. In *Proc. ACM SIGMOD Conf. on Management of Data*, San Francisco, 1987. ACM, New York.

[31] P. Pistor and R. Traunmüller. A data base language for sets, lists, and tables. *Information Systems*, 11(4):323–336, December 1986.

[32] M. A. Roth, H. F. Korth, and D. S. Batory. SQL/NF: A query language for ¬1NF relational databases. *Information Systems*, 12(1):99–114, March 1987.

[33] H.-J. Schek, H.-B. Paul, M.H. Scholl, and G. Weikum. The DASDBS project: Objectives, experiences and future prospects. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):25–43, March 1990. Special Issue on Prototype Systems.

[34] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, June 1986.

[35] 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*. Springer LNCS 361, Heidelberg, 1989.

[36] M. H. Scholl. Theoretical foundation of algebraic optimization utilizing unnormalized relations. In *ICDT '86: Int. Conf. on Database Theory*, pages 380–396, Rome, Italy, September 1986. LNCS 243, Springer, Berlin, Heidelberg.

[37] M. H. Scholl. Towards a minimal set of operations for nested relations. In M. H. Scholl and H.-J. Schek, editors, *Handout Int. Workshop on Theory and Applications of Nested Relations and Complex Objects*, Darmstadt, April 1987. (Position paper).

[38] M. H. Scholl. *The Nested Relational Model — Efficient Support for a Relational Database Interface*. PhD thesis, Department of Computer Science, Technical University of Darmstadt, 1988. (in German).

[39] M. H. Scholl, H.-B. Paul, and H.-J. Schek. Supporting flat relations by a nested relational kernel. In *Proc. Int. Conf. on Very Large Databases*, pages 137–146, Brighton, September 1987. Morgan Kaufmann, Los Altos, Ca.

[40] M.H. Scholl, C. Laasch, and M. Tresch. Views in object-oriented databases. submitted for publication, July 1990.

[41] M.H. Scholl and H.-J. Schek. A synthesis of complex objects and object-orientation. In *Proc. IFIP TC2 Conf. on Object Oriented Databases – Analysis, Design & Construction (DS–4)*, Windermere, UK, July 1990. North-Holland. to appear.

[42] G.M. Shaw and S.B. Zdonik. An object-oriented query algebra. *IEEE Data Engineering*, 12(3):29–36, September 1989. Special Issue on Database Programming Languages.

[43] M.R. Stonebraker and L.A. Rowe. The design of POSTGRES. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 340–355, Washington, D.C., May 1986. ACM, New York.

[44] D.D. Straube and M.T. Özsu. Query transformation rules for an object algebra. Technical Report TR 89–23, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada, August 1989.

[45] D.C. Tsichritzis and O.M. Nierstrasz. Fitting round objects into square databases. In S. Gjessing and K. Nygaard, editors, *Proc. European Conf. on Object-Oriented Programming*, pages 283–299, Oslo, August 1988. LNCS 322, Springer Verlag, Heidelberg.

[46] G. Wiederhold. Views, objects, and databases. *IEEE Computer*, December 1986.

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

[48] N. Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, June 1988.