Evolution of Data Models

Hans-Jörg Schek **Marc H. Scholl**

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

Abstract

Relational, complex object, and object-oriented data models seem to establish three diverging directions and it appears hopeless to arrive at a new higher-level unifying data model as a new platform for the non-traditional applications as well as for the classical ones. In this paper, however, we give some evidence that recent object models can be considered as an evolution from classical ones. Specifically, we describe a first evolutionary path from relations through nested relations and complex objects to object networks. A second evolutionary path starts from the DBTG network model and shows how this is turned into an object network again. Similar observations apply for other models, such as semantic data models or knowledge representation languages. Thus, we conclude that the next generation object-oriented data models should integrate concepts from these predecessors. This enables compatibility, coexistence, and cooperation between the different systems as well as a possible migration path, and may lead to a new overall data model platform for application development.
Keywords: Data Models, Complex Objects, Nested Relations, Object-Orientation, Query Language

1 Motivation

Relational data models with SQL as data definition and data manipulation language are a quasi-standard for commercially available data base systems, but research in data models and interfaces to database systems seems to diverge into different directions rather than converging to a new, higher-level object model which could serve as a new platform for the non-traditional applications as well as for the classical ones. Object-oriented databases often start from object-oriented programming languages and add persistency as well as other database functions such as transaction management. Objects (instances of abstract data types) are encapsulated and only "accessible" through a well-defined interface. Object manipulation is by invoking type-specific interface functions (methods). The advantages are twofold: first, the structure of objects is hidden (which provides a higher level of abstraction) and second, the methods can implement integrity checks that are specific to the object type. Typically, OODBMS have a complete programming language environment, but provide little means of descriptive set-operations [GM88]. This reflects their origin from the programming language realm. As there are many programming languages, we expect as many database systems and the integration aspect of databases across several applications written in different languages seems to be lost or at least more difficult.

Complex Objects have evolved from relations in that they are constructed by repeated application of tuple and set constructors. Nested Relations as a special case of Complex Objects have been studied

in detail in their theoretical as well as practical issues during the past few years [AFS89]. One of the major strong points is the fact that they preserve the descriptive set-oriented style of relational query languages. Like in many other models, a variety of query languages have been proposed for nested relations and complex objects, such that it might not seem justified to consider this direction as *one* approach. However, the essential ingredients in all of them are actually the same. Therefore, like in the classical relational model, we can assume one — may be hypothetical — algebra as their common basis. Furthermore, it seems that shared subobjects or recursively structured objects are incompatible with the hierarchical organization and hence can not be brought under the same unifying umbrella.

Adding generalization or specialization known from semantic data models [HK87] or similar notions from knowledge representation schemes ultimately seems to increase the divergence. While such mechanisms facilitate the definition of new types and gives rise to inheritance of attributes, methods, or functions, making query formulation easier, they also need new constructs in queries, such as type predicates and update operations for changing the type of an object. Again, many proposals exist that look pretty incompatible. Furthermore, a number of problems with the formal foundations still lack solutions, for instance how to handle polymorphic types, and how to deal with the higher-order syntactic constructs in a first-order logical framework [Bee89, KL89].

Therefore, in view of all this it appears hopeless to obtain a higher-level unifying data model as a new platform which contains the existing data models as special (degenerate) cases. However, in this paper we will put together some evidence that the seemingly different concepts are not really incompatible by their very nature. While there exist some differences with respect to formal definitions and in the terminology, we try to show that the essential features are very similar with regard to their behaviour for a user. We think that the differences are not exactly " ... just silly exercises in surface syntax" [SRH90], but we do argue that, apart from the diverging development, we can — in retrospect — interpret them in a consistent way. In order to explain this we describe two evolutionary ways: the first starts with the relational data model and encompasses nested relations, complex objects and object networks ("structural" object-orientation [Dit86]). The second one starts with the DBTG network model and leads directly to the structural part of object networks.

As a result of this exercise we see how recent object models can be obtained as a synthesis of well-established concepts, namely (1) set-oriented, descriptive query and update languages from relations and nested relations, (2) recursive schema definitions from the network model to allow for object sharing, (3) behavioral abstraction from abstract data types providing encapsulation (methods), and (4) inheritance through the specialization of types. Notice that object methods also provide extensibility and adaptation to application classes. The practical aspect of such a synthesis is that classical data models are contained as special cases. For example standard SQL can be extended to SQL for complex objects. This, in turn, can be generalized to SQL for object networks, and opened to object-specific methods. Thus, a kind of upward-compatibility is achievable which facilitates cooperation between different systems.

We do not describe a specific object model here, although we use our notation of [SS90b] and [SS90a] where appropriate, for explanation purposes and as an example. Our principal goal, however, is to point out how the salient features of recent research proposals are obtained as an evolution from well-known classical ones, and what features have been added to them. In fact, it turns out that apart from all terminological and syntactical differences, a substantial body of core concepts can be found in all of them. The research proposals we have in mind include the ORION query language described in [BKK88, Kim89], the OSQL language of the IRIS project [Bee88, WLH90], the PROBE language [MD86], the

object-oriented extension OODAPLEX of the Functional Data Model [Day89], the EXCESS language and EXTRA model of EXODUS [CDV88], the data model used in the O_2 project [D^{+}90, LR89], the LauRel project [Lar88], the MAD model [Mit87] with its MQL query language, and the HDBL language [PT86] of the AIM prototype system. Previous work on evolutionary aspects includes [Bee88] and [SS90b]. Theoretical work on the foundations of OODB models [AK89, Bee89] shows strong similarities to nested relational/Complex Object models. The same is true for query languages/algebras developed for OO models [Kim89, SZ89].

In the following Section 2 we describe the evolution starting with relations and Section 3 contains the evolution when the DBTG network model is used as a starting point. In Section 4 we show that similar results could be obtained when starting from models such as the Functional Data Model, various semantic data models, or knowledge representation languages. This section also contains some remarks on the inclusion of (retrieval and update) methods, that is, "behavioral" object-orientation. Finally we conclude in Section 5 with a summary.

2 Extending Relations to Objects

2.1 From Relations to Nested Relations and Complex Objects

When we consider the relational model as a language for defining data types, that is, given some set of base types (the atomic domains) we construct relations from these, we find that the only type constructor available is **relation**. Relations are sets of tuples, however, in contrast to the **set** ($\circledast$) and **tuple** ($\otimes$) constructors found in semantic data models [HK87], the relational model does not offer them as separate constructors. Further, the first normal form condition requires that the domains of attributes (that is, the tuple-components) be *atomic*. Therefore, this **relation** type constructor can only be applied *once* per constructed type (relation).

The nested relational model [SS86] is obtained from the flat relational one by allowing relations as values of attributes. That is, we allow the single type constructor **relation** to be applied *any number of times* in a constructed type (nested relation). If we further relaxed the restriction that **set** and **tuple** constructors have to strictly alternate, then we end up with what is usually called "Complex Objects" [AB88] or extended nested relations in [PT86]. Figure 1 gives an overview of (1NF-) relations, nested relations, and complex objects considered as type constructors in the graphical notation from [HK87].

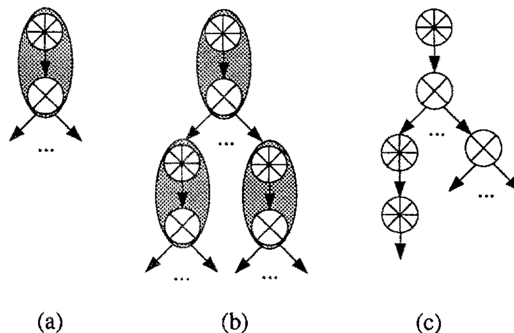

Figure 1. Examples for Relations (a), Nested Relations (b), and Complex Objects (c) as type constructors

If we write these type definitions down in a linear syntax, like it is usually done in programming languages, see also [Sch77], then we might obtain for a sample nested relation *Dept(dno,name,budget,...,Staff(ssec#,name,sal,...))*:

```
type DeptRel = relation                    type EmpRel  = relation
                 dno:      integer                          ssec#:    integer,
                 name:     string,                          name:     string,
                 budget:   real,                            sal:      real,
                 ...,                                       ...
                 Staff:    EmpRel                         end;
               end;

persistent var Dept: DeptRel;
```

Here, we used the statement "`persistent var Dept: DeptRel;`" as the declaration of the relation *Dept* over the corresponding type, which is, in fact, the declaration of a persistent variable. If we want more flexibility, for instance in the sense of complex objects, we could have defined tuple types "DeptTup" and "EmpTup" first, and then defined "DeptRel" and "EmpRel" of type "set of DeptTup" and "set of EmpTup", respectively.

Essentially, nested relations or complex objects do not really extend the data model with new concepts, they just permit a more deliberate use of the type constructors already available. Hence, rather than just "flat files", we can now model quite complex hierarchical (!) structures. In a very similar, and likewise obvious way, we can extend any of the known relational query languages, no matter whether they are algebraic, calculus, or SQL style. In fact, the only extension is to use query language operators whereever relations occur. Therefore, for instance, if tuples in a relation contain a relation-valued attribute (a "subrelation"), we can apply relational operations to it nested into a projection list or a selection predicate. Such nested relational query languages have been discussed in detail in several publications [JS82, FT83, SS83, AB84, SS86, PT86, RKB87, RKS88]. Therefore we will not elaborate on them in much detail. A simple example in an extended SQL language refering to the above nested department relation is the following:

Example 2.1: We like to see for all departments their names and the names of their employees making more than 40k:

```
Select name, ( Select name
               From   Staff
               Where  sal < 40000 )
From   Dept
```

2.2 From Relations to a Relations Graph

An important point about nested relations and complex objects is that they are incapable of directly representing non-hierarchical or recursive relationships. Nested relations have proven useful for both theoretical as well as practical investigations, for instance, several prototype systems have been implemented [BRS82, DKA+86, DL89, PSS+87, SPSW90] and it has been shown that nested relations provide a sound formal basis for the description of internal storage structures [SPS87]. As a data model to be used at the logical (that is, user interface) level, however, we will need even more complex modeling capabilities that include support for many-to-many relationships and recursive relationships.

When looking at the type definitions for nested relational schemas or complex object schemas shown above, we notice that, in order to be well defined, these type definitions have to be *non-recursive*. That

is, we could not define the "EmpTup" type to include a component "worksfor" of type "DeptTup", for instance. While this "component" would certainly be nice to have, so as to provide a fast access from an employee to his/her department, the type definitions would be recursive and thus their semantics would not be known. In semantic data models, such as those discussed in [HK87], the situation is exactly the same. Figure 2 shows a construction that is forbidden and how it has to be modelled instead. (Solid lines connect constructed types to their components, the dotted lines denote functions.)

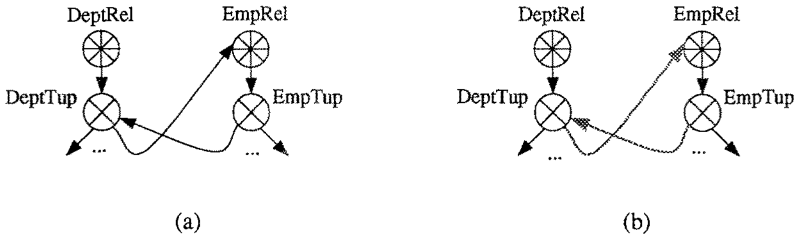

Figure 2. Example of a forbidden type construction and solution using functions

The solution is quite obvious: the *recursion* in the type definitions is broken up by introducing functions in addition to tuple components. In other contexts, this would be called "pointers" (classical programming languages), "reference semantics" [Mey88], "ref_to" attributes [PT86, Lar88], or "Object-IDs" (various object-oriented database models). We like the notion of functions mapping instances of the domain type to instances of the range type best, because it is both, a known concept from mathematics, and it provides a clean conceptual abstraction of implementation level details, such as using pointers or surrogates. This change to a functional paradigm is the essential part. In order to deal with only a few concepts, it seems elegant to view attributes (tuple components) also as (stored) functions. Furthermore, user-defined functions (that is, ADTs as base types) are then easily integrated. The data model of the IRIS system [Bee88, WLH90] and OODAPLEX [Day89] use the same kind of object-function-model. The effect of this is that what has been restricted to a hierarchy of relations is now extended to become a general graph of relations.

For a somewhat more complete example to be used in the sequel, we show the schema of a database containing information about companies, employees, and automobiles (see [Kim89]) in Figure 3. The graphical notation here is drawn from KL-ONE, a knowledge representation model [BS85]. The ovals denote object types, the edges functions between them. Double headed arrows indicate set-valued functions. The small circles represent atomic data types, such as integers or strings.

A linear syntax for these type definitions is:

```
type Company =
        name:          string,
        budget:        integer,
        president:     Employee,
        staff:         set of Employee,
        location:      set of City,
        produces:      set of Vehicle;

type Vehicle =
        id:            string,
        color:         string,
        owner:         Employee,
        manufacturer:  Company;

type City =
        population:    integer,
        name:          string,
        zipcode:       string,
        has_comp:      set of Company;

type Employee =
        name:          string,
        bdate:         date,
        hiredate:      date,
        ssec#:         integer,
        salary:        integer,
        address:       City,
        works_for:     Company,
        owns:          set of Vehicle;
```

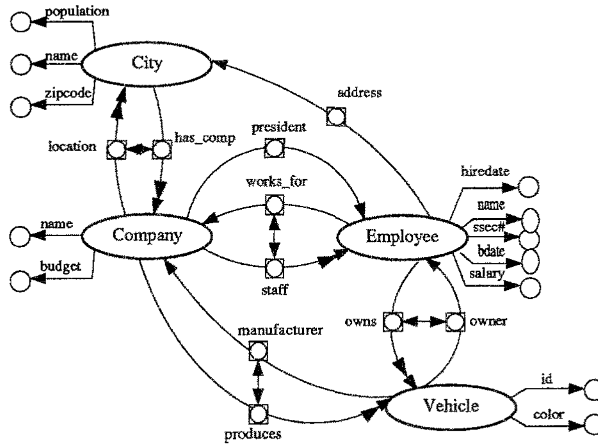

Figure 3. A relation graph

The "components" in a type definition are now interpreted to be functions from that type to the corresponding range type. For instance, "owns" is a function from type "Employee" to a set of objects with the element type "Vehicle", and so forth. The two-sided arrows connecting, for instance, the "owns" and the "owner" functions in Figure 3, indicate that the functions are inverses of each other. In our context of set-valued functions that means, for instance, if some car c is contained in the set returned by the "owns" function applied to some employee e, then e is the object returned by "owner" applied to c. Before we discuss query language issues, we add generalization/specialization that is typically found in both, semantic data models and object-oriented programming languages. For update operations, however, we want to emphasize already the need to distinguish object **creation/destruction** from **adding/removing** objects to/from sets (such as set-valued functions). This aspect is further elaborated in Section 2.4

2.3 Adding Generalization/Specialization

Defining a new type as a subtype of an existing type is a convenient way of reusing type definitions. For instance, as employees are special persons, we can define the type "Employee" as a subtype of "Person". As a result, the subtype *inherits* the description of the supertype, that is, automatically employees have names, which are strings, birthdates that are dates, and an address (a city). Furthermore, "Employee" is a refined type, since it has more functions, defined on its instances. For example, employees have an employee number, a social security number, work for a company, and so on. On the instance level, we expect to have a similar effect: when we talk about all persons, we want to see all instances of both "Person" *and* "Employee", since the latter are also persons (this is termed "inclusion semantics" of subtyping).

In the following Figure 4, we added generalization edges to the object schema known from Figure 3. Persons have been added, two specializations of vehicles, automobiles and trucks, and a special kind of companies, namely autocompanies, that is, those that produce automobiles. We also showed a "restricts" arrow connecting two functions. This is a kind of specialization among functions. In fact, the produces function on the autocompany type is restricted (specialized) to return only special vehicles, namely automobiles.

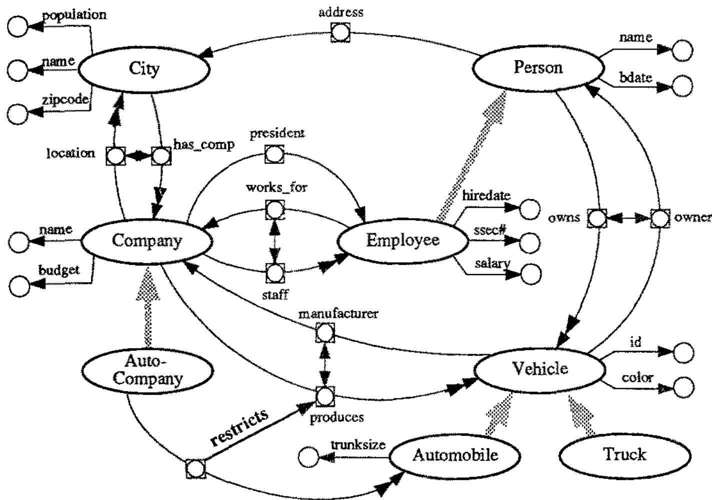

Figure 4. Relation graph with generalizations

In a linear syntax, this example database schema could be defined as follows:

```
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...};
```

Now we have type definitions and a database definition that can rightly be called object-oriented. With the appropriate interpretations of the recursive type definitions, we will also see that nested relational query languages can still be applied.

2.4 Essentials of an Object-Oriented Query Language

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. In the previous subsection we showed how to extend the data structures gradually from flat via nested relations to an object-oriented model. As for the operations, the path is exactly the same. When moving to nested relations, relational query languages have been extended to be applied in a nested fashion (see above). We have shown in [SS90b, SS90a] that these hierarchically nested query languages can still serve as appropriate languages for network schemas. The key idea is to take, for each particular query, a hierarchical view onto the object net. That is, for one query it might be most convenient to start with the Company class and consider Cities as subobjects (through the set-valued "location" function). The query can thus be formulated as if the database contained a nested relation *Company(..., location (...))*, where the subrelation comprises the information on all cities where the individual companies have a branch. Another query, however, might be best expressed on the converse hierarchical view, that is, start with Cities and treat Companies as subobjects (through the set-valued function "has_comp"). Any hierarchical view is implicitely contained in the object net, without implications on redundancy.

In order to deal with generalization/specialization in the query language we need type test predicates, such that we can, for instance, test whether a person is also of the more special type employee. Similarly, since objects evolve over time, and may well change their type, we need operations that manipulate the type-instance relationship between types and objects. In our example, such operations are needed when hiring a person as an employee or when an employee retires.

Further extensions to the update capabilities of an object-oriented language as compared to a relational one are that we need to distinguish between object creation and insertion of an existing object into a class (**insert** vs. **add** in [SS90b, SS90a]), and similarly between deletion and removal from a class (**delete** vs. **remove**). The **add** and **remove** operations are also used to manipulate the values of set-valued functions.

As discussed in [Kim89, SS90b, SS90a] it is crucial in an object-oriented data model to be precise about what the semantics of queries are. Particularly, the closure property of a query language is important: in the relational (and nested relational/complex object) model, the input to a query as well as its output is a relation (or two in case of binary operators). This enables complex queries to be composed of simpler operators. It is this very feature that led to the success of relational query languages, since only a handful of basic operators have to be understood to issue complex queries. Furthermore, also for the implementation of a DBMS, the advantage is that the query processor can concentrate on the efficient implementation of these basic operators and a mechanism to combine them. In an object-oriented data model, however, the situation is less obvious. Since objects are encapsulated, objects can never be delivered to a user as the response to query, rather, *data* about objects has to be extracted from the objectbase [SS90b]. This corresponds to queries returning relations or nested relations as their result (the **extract** operator of [SS90b], also see [AK89, Bee89, JSS90]). This is in obvious contradiction with the closure property. On the other hand, in order to use queries for view definitions [HZ88, SS90a] or as the specification part of (set-oriented) update operations (that is, to identify the objects that should be subjected to the update), a query semantics is needed that returns objects. Two alternatives immediately come to mind: either the resulting objects are newly created during query execution [Kim89, SZ89] ("*object-generating*" semantics) or they are preexisting ones [SS90a] ("*object-preserving*" semantics). The first alternative has two major disadvantages. ObjectIDs would have to be created for the new

objects, which may be costly and introduce additional problems. For instance, if, in a recursive query, we invent new object IDs in the iteration step, we might never reach a fixpoint, that is, the iteration might never stop. Furthermore, queries resulting in new objects could not be used to define views, if we want them to be updateable. So, object-preserving query semantics seems to be a necessary condition for an object-oriented data model to meet all the requirements typically imposed on a reasonable query language.

One important addition to the query and update language of an object-oriented data model is the introduction of *variables*. Since we do not talk about references, ref_to-attributes, or ObjectIDs in the model, we need some other way to (at least temporarily) refer to objects in the language. A very common solution to this is to use variables as some kind of temporary name for objects (or object sets). The interested reader is refered to [SS90b, SS90a] for a detailed discussion of the query language issues. In the following we conclude by giving some example queries that illustrate the basic ideas, and finally show a tabular summary of this section. The queries shown refer to the example given in Figure 4

Example 2.2: Insert a new company with two new employees.

```
var c1:      Company,
    e1,e2:   Employee;

    c1 := insert into Company
                     (name := "Shooting Star, Inc.",
                      budget := 100000 );

    e1 := insert into Employee
                     (ssec# := 1234,
                      name  := "John",
                      works_for := c1);

    e2 := insert into Employee
                     (ssec# := 4321,
                      name  := "Mary",
                      works_for := c1);
```

Here we used variables to hold the results of the object creations, viz. the new objects. Later, we can use these variables to refer to the objects, for example, to assign them to some function defined on other objects.

Example 2.3: Delete companies with a budget under 1100k and fire their employees.

```
var C: set of Company

C := (filter x in Company with budget(x) < 1100000);

remove e from ( filter y in Employee with works_for(y) in C );

delete(C);
```

Like in nested relational query languages, set inclusion predicates are added. We introduced explicit object variables for clarity ("filter" is our selection operation). Notice that we applied the update operation "remove" to the result of a (selection) query.

Example 2.4: Let us extract the budget of all Companies producing Porsches. Next, we extract a complex data structure which shows for each Company its employees and for each employee the cars owned by her.

```
P  := filter a in Automobile with name(a) = "Porsche";
PC := filter c in Company with P in produces(c);

Q1 := extract budget(x) from x in PC;


Q2 := extract name(c),
             extract ssec#(e), name(e), sal(e),
                    extract name(a)
                    from a in owns(e)
             from e in staff(c)

      from c in Company;
```

This last example shows the **extract** operator, which turns objects into (nested) relations suitable for output purposes. That is, Q1 is actually a flat relation with schema *Q1(budget)*, and Q2 is a nested relation *Q2(name, staff(ssec#,name,sal, owns(name)))*. The result of the **extract** operation is no longer a part of the objectbase. Rather it is a set of tuples derived from objects. Thus, **extract** is a means of transfering data into another environment, for instance, a relational DBMS or a record-oriented application tool.

To summarize, Table 1 shows the correspondence between nested relational and object models. Such correspondences have already been observed between relations and objects, see e.g. [Bee88, Wie86]. 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 [AK89, Bee89, Kim89, SZ89].

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 abstract type
domains ... finite, defined bottom-up	"domains" ... infinite, not defined explicitly

Table 1. Nested relational versus object model concepts

3 From CODASYL to Semantic/Object-Oriented Models

In this chapter we add a second evolutionary way to currently discussed new data models starting from the network model according to the CODASYL proposals, shortly called the network data model (NWM) in the following. While others also have pointed out strong relationships between the network model and object-oriented models, e.g. [Ull88] we will present the network model in such a way that the similarities, but also differences and additional features can be seen easily.

3.1 A Reinterpretation of the NWM Data Definitions

The data definition language of the NWM (CODASYL DDL) allows to define record types and NWM-set types. Roughly, records are used for the description of objects and the NWM-sets represent 1:n relationships. In Figure 5 a small network is shown in the usual graphical representation (Bachman diagram) along with schema definitions

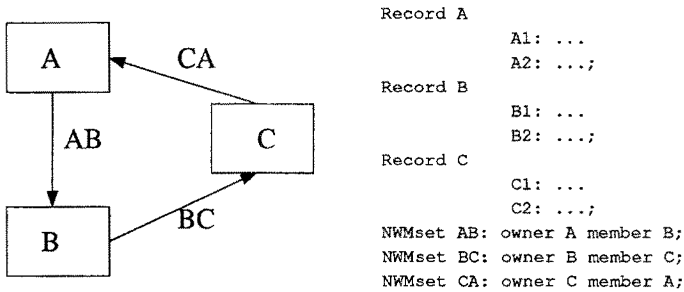

Figure 5. A sample NWM data definition

The presentation above shows the principle only. We did not include further specifications about referential integrity in form of the insertion or the retention clause. Nor did we specify the order of members or the set selection clause. They are important to understand the operations of the NWM in detail, which, however, is not our intention in this paper. Here we will concentrate on the general aspects:

The above DBTG syntax clearly is only one possibility from many other equivalent ones. A less "fateful" one could have been the following version:

```
Record A                Record B               Record C
        A1: ...                 B1: ...                C1: ...
        A2: ...                 B2: ...                C2: ...
        AB: set of B;           BC: set of C;          CA: set of A;
```

Obviously, we defined the NWMsets within the record definition of the related owner. For instance, AB is defined within A because AB has the owner A. Similarily, BC and CA are defined within B and C, respectively. It is evident that we could produce the original syntactical version from the new one and vice versa, and we can perform such a transformation for any given network schema definition.

The only exciting observation now is that in the new notation, the NWM schema definitions look pretty much like the definitions of objects we had in the previous chapter: A NWMset appears as a set-valued function returning "member" objects. Any other field can be interpreted as a function returning a primitive object. Note that we do not define the inverse function to a NWMset, i.e., we do not introduce a name for it. But nevertheless this function exists implicitly and is called "owner-xy" for each NWMset xy. It is characteristic for the network model that a member has a unique owner, i.e., the inverse (owner) function must not return a set, but a single object only. Of course, we could define another — independent ! — NWMset in the opposite direction, but the DBMS could not guarantee that it is in fact the *inverse* of the former.

In our new schema notation we also see much clearer that the object (record) type definitions are recursive: A is based on some primitive objects and defined in terms of B, then B is defined in terms of C, and C, in turn, needs the definition of A. Therefore, we should not be surprised to find references or

pointers called currency indicators when the semantics of operations is explained. In that aspect the NWM is much closer to object models in that object identity is already introduced in terms of the database key.

Seen from this perspective, a NWM is a special case of an object model where the functions for object types are either single-valued returning a primitive object only or set-valued returning a set of objects under the restriction that the inverse of each function is either single-valued or undefined, but not set-valued. An important notion is missing in the NWM: there is no **is_a** relationship between types and so, no notion of inheritance of functions is supported. As a further restriction, the NWM disallows the use of recursive NWMsets, i.e., a set-valued function defined on an object type A returning members of that same type A. While the latter has been discussed heavily during the CODASYL developments and was deemed unnecessarily restrictive already a long time ago [Oll78], the other more serious ones have never been questioned. Therefore, already from the data definition point of view, recent object models have introduced significant improvements in that

- functions may not only return a primitive object, but also a non-primitive one;
- functions and their inverses may return *sets* of objects (thus represent *n:m* relationships);
- direct recursion is allowed;
- generalization/specialization is supported.

3.2 Evolution of NWM Operations

In order to define operations, one has to understand the currency indicator concept. Seen from the new perspective, considering the NWM schema as a set of abstract data types, we must know how to create, change, and delete these instances and how to observe their behaviour. This is what the currency indicators are for (following the nice presentation of these aspects in [Sch87]): They are predefined (pointer) variables for object instances with — unfortunately — fixed names. The most prominent variable is the "current-of-run-unit" (CRU), a pointer with a type which is the union of all record types from the NWM schema. During the execution of a navigational CODASYL program, the CRU variable holds the most recently visited record. All update operations and the GET apply to the record pointed to by CRU. Also, the NWMset variables for each "current-of-set" are polymorphic, they can hold either the owner or one of the member records. Just the record variables ("current-of-record") have only a single type.

Compared to the object model of the previous section, we notice that, w.r.t. typing, the NWM-operations are less consistent than they could have been. Nevertheless, we find the basic principles of object models already here: We have to distinguish between the "object base", i.e., the set of NWM instances, which can only be manipulated by changing the pointer variables mentioned above, and the visible part of the objects, i.e., the primitive objects for which we have agreed on a standard representation for the communication with the outside (called "user working area" in the NWM).

Let us shortly summarize the operations manipulating the internal object base first: For the creation of NWM records we have INSERT, which establishes a new instance and assigns some primitive object values from the outside to the related functions. In order to identify instances later, several kinds of FIND are used. The result of any FIND operation is a single object which is returned in the CRU variable. Via this variable, an object can be CONNECTed to or DISCONNECTed from a NWMset in a subsequent update operation. Also, another form of the FIND statement can be used (for relative positioning within NWMsets). However, FINDs cannot be nested. The CRU variable is the handle for DELETE and for UPDATEs of primitive values.

When we want to get something out of the object base, we have to apply one or several FINDs first in order to set the CRU variable to the desired instance. A subsequent GET dereferences the CRU (pointer) variable and provides us with the primitive object values in a representation we understand, i.e., in the user working area with its predefined buffers. If we want to see more objects, e.g., those satisfying the same condition, or all members of a NWMset, we must repeat the above FIND/GET procedure for every instance. Again, there is no nesting of operations, and no set-oriented mode of operation.

This shows that some of the essential features of object models can be found in the NWM: NWMrecords are instances of abstract types manipulated by a limited set of functions (called FINDs), mostly for navigational access. CONNECT and DISCONNECT are used to add or remove objects to or from relationships. Finally, GET retrieves data about the objects into a predefined communication area.

It is obvious that the object model described in the previous section is an evolution from the NWM that improves it in the following main aspects:

- An arbitrary number of object variables can be used and we can freely give names to them, instead of using just the predefined pointer variables.
- Functions returning a single object can be nested. Nesting of set-valued functions requires an iterator to be type-consistent.
- Sets of objects can be identified and referred to via set variables.
- There is one uniform iterator over sets, whether the set is defined by a predicate or whether it is the value of a set-valued function.
- Instead of the GET operation returning a single record data structure, **extract** returns sets of data records or even nested relations/complex objects

Let us look into these differences by returning to our example from Section 2 (cf. Figure 4. The central part of the example is represented in our new NWM data definition syntax as follows:

```
Record Company
        name:       string,
        president:  Employee,
        staff:      set of Employee,
        produces:   set of Vehicles;

Record Vehicle
        id:         string,
        color:      string;

Record Employee
        name:       string,
        bdate:      date,
        hiredate:   date,
        ssec#:      integer,
        salary:     integer,
        owns:       set of Vehicle;
```

We did not model the **is_a** relationships, e.g., between Person and Employee (by a single-valued NWMset), nor did we define the Company-City relationship. In order to model the one-to-one president relationship from Company to Employee we must also use a NWMset. As already mentioned, the NWM implicitly gives standard names to the inverses of the NWMsets. Therefore, we have the "functions" owner_president, owner_staff defined on Employee, owner_produces on Vehicle, and owner-owns on Employee. In the following, we give some examples of queries expressed in the NWM-style and compare them with expressions in the object model language.

Example 3.1: Determine all employees who own a vehicle which is produced by the company the employee works for: **NWM:**

```
Outer Loop over Employee:
    Find any/next Employee;
    Get; Keep ssec#;
    Find owner_staff;
    Inner Loop over produces members:
        Find first/next produces;
        Find owner_owns;
        Get; Compare this ssec#
             with the kept one;
             If same: output ssec#;
    End inner;
End outer.
```

OM:

```
SE := filter e in Employee with
       exists ( filter v in produces(works_for(e)) with e=owner(v) );
extract ssec#(x) from x in SE;
```

Apart from being much more compact, the OM formulation is clearer in that it allows to iterate over a set of objects, regardless whether it is the set of all employees or it is the set of vehicles returned by the composition of functions *produces(works_for(e))* for the employee variable *e*. Common to both solutions is the usage of the two owner functions, but the OM expression allows to compare objects directly. Note that already quite some time ago we saw high-level query languages for the NWM, such as CQLF [MP82], NUL [DH76], or IQS [IQS86], that tried to introduce a higher level of abstraction by allowing set-oriented retrieval, or even some graphical query support, as in FORAL-LP [Sen77]. More recent developments in this area even compare their languages to nested relational ones [Kle89, Hou82].

We could have avoided the use of the set variable SE in the OM expression above. A textual substitution of SE into the **extract** expression leads to a single expression without a variable. On the other hand, using variables often increases readability, as is shown by the following expression in the OM for our example:

```
SE := filter e in Employee
        with exists EV intersect CV
        using EV = owns(e),
              CV = produces(works_for(e));
```

Like the object variable *e*, the set variables *EV* and *CV* are local within the expression and not defined outside. Textual substitution of *EV* and *CV* would lead to another correct formulation. When new objects are generated and shall be connected to the rest of the objectbase, it is not only convenient but necessary to use variables. In the following example, textually substituting for the variables would create two distinct new objects instead of one!

Example 3.2: A new employee with name "bigboss" is inserted who not only works for the "xyz" company, but becomes the president of it right away.

NWM:

```
find "xyz" company;        /* current-of-set for staff and president */
insert Employee "bigboss"; /* the CRU now is the new employee */
connect to staff;
connect to president;
```

OM:

```
bb := insert into Employee ( name := "bigboss" );
update x in ( filter c in Company with name(c)="xyz" ):
       add bb to staff(x),
       set president(x) := bb;
```

While in this example almost the same steps are performed in either language, the use of explicit variables increases readability, and helps avoiding what has been called "currency confusion", that is, erroneously connecting the CRU record to the wrong NWMset occurence.

The NWM has some further interesting features, such as integrity checks by external methods on records, or "virtual" attributes. The latter are a means of defining derived functions from stored ones. For instance, assume we want to have for each employee record the name of the company the employee works for. We can define a virtual attribute "cname", whose source is the "cname" field of the owner w.r.t. the NWMset "staff". Languages for object models have similar capabilities, but in more generality. Suppose we want to see the cities where the employer of an employee has a branch as a new (derived) function "comp_city" on the employee type:

```
extend e in Employee by comp_city := location(works_for(e));
```

After this definition, the function comp_city can be used in retrieval statements in exactly the same way as any other function that was previously defined.

In general, we can define new functions by arbitrarily complex query expressions (cf. derived functions in IRIS [WLH90]).

4 Further Aspects

4.1 Other Data Modeling Approaches

Besides the two evolutionary paths discussed above, there are many more. Some of them are just possible directions, others have actually been pursued in research projects. Among the latter is, for instance, the Functional Data Model [Shi81]. In this model, functions, functional composition, and tight database-programming language integration have been the main focus. Now, where methods (that is, functions on objects) became a hot topic, these considerations regain attraction. In fact, in [Day89], an object-oriented extension, called OODAPLEX, is introduced together with an algebraic and an SQL-style language. Not surpringly, the model is very similar to the one we presented in Section 2.

Many projects tried to extend semantic data models of all flavors so as to include a query (and update) language, such that they could directly be used as a DBMS interface rather than being just a conceptual tool for database design. Of course, when accounting for the freedom of constructing arbitrarily complex structures from some given base types [HK87], these query languages resemble those developed in the field of nested relations and complex objects. EXTREM, for instance, is such a model that puts main emphasis on building on a (nested) relational semantics [Heu89].

Our own model is based on KL-ONE, a representative of knowledge representation models used in AI [BS85]. In the BACK system [PSKQ89], a "pure" KL-ONE system, an interesting conclusion after several years in development of a new A-Box language (the part of KL-ONE that deals with instances of the schema) was that some kind of SQL-like query facility is crucially needed to be able to manipulate knowledge bases with a large amount of instance-level information. Similar results have been reported from investigations on the use of Frames as a data model [RS89]. Whether one starts from AI techniques and integrates DB functionality or vice versa, the need for an integration of DB and AI techniques in the field of data/object/knowledge modeling and querying is clearly recognized by researchers from both fields. Common areas of interest have been identified before, but mostly on the dynamic side, that is, resursive query processing.

4.2 Behavioral Abstraction — Methods

The inclusion of behavioral abstraction mechanisms, that is, methods, into any of the advanced data or object models discussed above, is still a largely open issue. It is useful to distinguish retrieval methods ("accessors" in the terminology of [Mey88]) from update methods ("manipulators"). The former can quite naturally be interpreted as functions attached to a class, that is, whose first argument is distinguished from the other arguments as being the "recipient" of the function call (that is, the message). In [Bee89], for instance, (multi-argument) methods in this sense are considered n-ary functions, the attachment to the class of the first argument is nothing else than interpreting the n-ary function as a functional, that is, a unary function that maps the first argument to an $(n\text{-}1)$-ary function in the remaining arguments. With this interpretation, retrieval methods fit into the known framework of all models supporting functions.

In order to adequately deal with update methods, however, traditional database theory has to be extended so as to cope with dynamics. Formal foundations for update operations, even in the relational context, are still an open research problem. Attempts presented so far include temporal logic, or algebraic specification techniques known from the programming language area (for the defintion of Abstract Data Types).

5 Conclusions

We have shown that recent proposals for new (object-oriented) data models can be considered as an evolution from many proposals and enable some degree of compatibility between the interfaces of today's databases with the expected ones in the future.

Starting with the relational data model we have only mentioned briefly that giving up the first-normal-form condition and allowing relations again as attributes leads to nested relations. If, in addition, tuples are allowed as attributes we receive the general model of complex objects. These models nicely describe hierarchical data structures. This is a consequence of constructive, non-recursive schema definitions. As in the classical relational model, foreign keys are used to describe non-hierarchical relationships. Languages for these nested objects are available as a direct evolution from (flat) SQL.

An essential step is the move from hierarchies to networks. This can be achieved either by the explicit introduction of reference attributes (comparable to pointers in programming languages), or by allowing recursive schema definitions along with the move to reference semantics instead of copy semantics for the assignment of objects to relationships in possibly different roles. Objects in this context are instances of abstract data types whose representation is not known to us and not of interest. Nevertheless, the retrieval part of languages for networks is almost identical to the previous nested SQL languages. The update part however needs add/remove operations in addition to insert/delete and needs variables in order to refer to objects.

A second essential ingredient is the support of generalization and spezialiation by maintaining a type lattice. This feature needs also the extension of the retrieval part of the language by type test predicates. Add/remove operations are again necessary for changing the type of an object by moving it up or down the type lattice.

A similar evolution can be observed when the classical network data model is used as a starting point. A simple syntactic transformation shows that the data definition language expresses similar structures as recent proposals. The currency indicators are predefined object variables to be manipulated by finds, insert/delete, and connect/ disconnect. In that aspect the network model is closer to recent object models.

However, again, generalization/specialization must be added, and the language must be brought up to a set-oriented, less procedural, and clean higher level. Fortunately, due to the influence of the relational model and its extensions together with the functional data model we know how this can be achieved.

It turns out that recent developements have brought much movement into a seemingly stagnating field caused by the dominance of the relational model. However we will not see a revolution. Smooth transitions from the standard we have today to higher-level models tomorrow seem possible.

We did not discuss the interesting synthesis of programming languages and databases into persistent database programming languages. This is partly because these proceedings contain a paper devoted to this, and partly because we believe that database systems in a multi-lingual environment will be necessary also in the future.

We are convinced that an object network model along the lines of what we presented can play a unifying role and can be regarded as a new higher level data model standard. This optimistic view is supported mainly by the following facts:

- It is a model obtained by relational and network data model ingredients. These models form the "quasi standard" for databases of today.
- It combines the essential features of many research proposals for future database interfaces (as mentioned in the introduction).
- It facilitates extensibility by its functional paradigm.

Challenging research questions in closely related areas include logical object network design and schema evolution, physical storage structure design, and optimization of the execution of object operations.

References

[AB84] S. Abiteboul and N. Bidoit. Non first normal form relations to represent hierarchically organized data. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 191–200, Waterloo, 1984. ACM, New York.

[AB88] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. Technical Report 846, INRIA, Paris, May 1988.

[AFS89] 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.

[AK89] 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.

[Bee88] 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.

[Bee89] 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.

[BKK88] J. Banerjee, W. Kim, and K.-C. Kim. Queries in object-oriented databases. In *Proc. IEEE Int. Conf. on Data Engineering*, pages 31–38, Los Angeles, CA, February 1988.

[BRS82] F. Bancilhon, P. Richard, and M. Scholl. On line processing of compacted relations. In *Proc. Int. Conf. on Very Large Databases*, pages 263–269, Mexico, 1982.

[BS85] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[CDV88] 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.

[D+90] 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.

[Day89] 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.

[DH76] C. Deheneffe and H. Hennebert. NUL: A navigational user's language for a network-structured database. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 135–142, Washington, D.C., 1976. ACM, New York.

[Dit86] K. R. Dittrich. Object-oriented database systems: The notions and the issues. In *Proc. Int. Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[DKA+86] P. Dadam, K. Küspert, F. Andersen, H. Blanken, R. Erbe, J. Günauer, V. Lum, P. Pistor, and G. Walch. A DBMS prototype to support extended NF^2 relations: An integrated view on flat tables and hierarchies. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 356–366, Washington, 1986. ACM, New York.

[DL89] P. Dadam and V. Linnemann. Advanced information management (AIM): Database technology for integrated applications. *IBM Systems Journal*, 28(4):661–681, 1989.

[FT83] P. C. Fischer and S. J. Thomas. Operators for non-first-normal-form relations. In *Proc. IEEE Computer Software and Applications Conf.*, pages 464–475, 1983.

[GM88] 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.

[Heu89] A. Heuer. A data model for complex objects based on a semantic data model. In S. Abiteboul, P. C. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects in Databases*, pages 297–312. Springer, Lecture Notes in Computer Science 361, Heidelberg, 1989.

[HK87] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[Hou82] B.C. Housel. QUEST: A high-level query language for network, hierarchical, and relational databases. In P. Scheuermann, editor, *Proc. 2nd Int. Conf. on Databases — Improving Database Usability and Responsiveness*, pages 95–119, Jerusalem, 1982.

[HZ88] 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.

[IQS86] Siemens AG, Munich. *UDS Dialog System IQS. User Manual*, 1986. (in German).

[JS82] G. Jaeschke and H.-J. Schek. Remarks on the algebra of non-first-normal-form relations. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 124–138, Los Angeles, March 1982. ACM, New York.

[JSS90] R. Jungclaus, G. Saake, and C. Sernadas. Using active objects for query processing. In *Proc. IFIP TC2 Conf. on Object Oriented Databases – Analysis, Design & Construction (DS-4)*, Windermere, U.K., July 1990. to appear.

[Kim89] W. Kim. A model of queries for object-oriented databases. In *Proc. Int. Conf. on Very Large Databases*, pages 423–432, Amsterdam, August 1989.

[KL89] M. Kifer and G. Lausen. F-Logic: A higher order language for reasoning about objects, inheritance, and scheme. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 134–146, Portland, OR, May 1989. ACM, New York.

[Kle89] H.-J. Klein. Pragmatics and semantics of NQL, a descriptive query language for network databases. *Information Systems*, 14(1):29–45, March 1989.

[Lar88] 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.

[LR89] 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.

[MD86] F. A. Manola and U. Dayal. PDM: An object-oriented data model. In *Proc. Int. Workshop on Object-Oriented Database Systems*, Pacific Grove, 1986.

[Mey88] B. Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, 1988.

[Mit87] B. Mitschang. The Molecule-Atom data model. In H.-J. Schek, editor, *Proc. GI Conf. on Database Systems for Office, Engineering and Scientific Applications*, Darmstadt, April 1987. Springer IFB 136, Heidelberg. (in German).

[MP82] F. Manola and A. Pirotte. CQLF—a query language for CODASYL-type databases. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 94–103, Orlando, Fl., 1982. ACM, New York.

[Oll78] T. W. Olle. *The CODASYL Approach to Data Base Management*. J. Wiley & Sons, Chichester, 1978.

[PSKQ89] C. Peltason, A. Schmiedel, C. Kindermann, and J. Quantz. The BACK system revisited. Technical Report KIT-Report 75, Technical University of Berlin, Berlin, Germany, September 1989.

[PSS+87] 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.

[PT86] P. Pistor and R. Traunmüller. A data base language for sets, lists, and tables. *Information Systems*, 11(4):323–336, December 1986.

[RKB87] 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.

[RKS88] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13(4):389–417, December 1988.

[RS89] U. Reimer and H.-J. Schek. A frame-based knowledge representation model and its mapping to nested relations. *Data & Knowledge Engineering*, 4:321–352, 1989.

[Sch77] J.W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2:247–267, 1977.

[Sch87] J.W. Schmidt. Data models. In P.C. Lockemann and J.W. Schmidt, editors, *Database Handbook*. Springer, Heidelberg, 1987. Chapter 1.

[Sen77] M.E. Senko. DIAM II with FORAL-LP: Making pointed queries with light pen. In *Information Processing '77*, page 635ff., Amsterdam, 1977. North-Holland.

[Shi81] D. Shipman. The functional model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[SPS87] 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.

[SPSW90] 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.

[SRH90] M. Stonebraker, L.A. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):125–142, March 1990. Special Issue on Prototype Systems.

[SS83] H.-J. Schek and M. H. Scholl. The NF^2 relational algebra for a uniform manipulation of external, conceptual, and internal data structures. In J.W. Schmidt, editor, *Sprachen für Datenbanken*, pages 113–133. IFB 72, Springer, Berlin, Heidelberg, 1983. (in German).

[SS86] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, June 1986.

[SS90a] M.H. Scholl and H.-J. Schek. A relational object model. In *Proc. Int'l. Conf. on Database Theory*, Paris, December 1990. to appear.

[SS90b] 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.

[SZ89] 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.

[Ull88] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Rockville, MD, 1988.

[Wie86] G. Wiederhold. Views, objects, and databases. *IEEE Computer*, December 1986.

[WLH90] 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.