To appear in: Advances in Conceptual Modelling, Databases, and CASE, pages 163-182.
P. Loucopoulos and R. Zicari (eds.), John Wiley, New York, 1991

Extensions to the Relational Data Model

Marc H. Scholl

ETH Zürich, Department of Computer Science

ETH Zentrum, CH–8092 Zürich, Switzerland, e-mail: scholl@inf.ethz.ch

1 Introduction: Three Ways of Extending Relations

In this paper we give an overview of research on extensions of relational database technology. In order to systematically classify different ways to extend the data model we take a programming language point of view of data models: a data model consists of a set of basic (predefined) types, a set of type constructors (or structuring primitives), and a set of operators (for the predefined as well as constructed types). Extensions to each of of these data model constituents are possible and have indeed been investigated in the past. Our presentation focuses on extensions to the type system (primitives and constructors) and those extensions to the operators that are implied by them.

During the 1980's, there has been a significant trend in database research addressing the problem of supporting non-traditional database applications. Though relational database systems (RDBMSs) entered the commercial marketplace in the early eighties, it seemed clear that, at least without major enhancements, they would not be appropriate for non-business applications. Several research groups started out to either enhance RDBMS technology in several ways or to develop completely different models and systems. Throughout this paper we limit our scope to those investigations that tried to keep some of the characteristics of the relational model and/or systems. Attempts to make semantic data models operational, for instance, Entity-Relationship models, have already been discussed in this volume before. Also, extensions in query languages' expressive power to deal with recursion, will be surveyed in a subsequent series of articles, as will the object-oriented approaches. Therefore, we will take a more "conservative" approach, that is, stay closer within the original relational framework.

The relational data model can be described as a type system with predefined, generic operators (cf. [102, 103]). There are, of course, many other ways to describe a data model, e.g., from a mathematical (logic, algebraic) point of view or from a semantic data modeling perspective. We chose the programming language approach, since it provides us with a systematic classification of different ways of extending the relational model. In general, a data model is viewed (from the programming language perspective) as consisting of a set of primitive (or atomic) predefined types, a set of type constructors, and a set of operators working on predefined and/or constructed types. Formally, this could be regarded as a many-sorted algebra approach. Obviously, from a certain level of abstraction, the same constituents make up a programming language. The difference between a data model and a

programming language lies in the kinds of features provided in each of these categories. Typical data models come with a set constructor, for instance, whereas programming languages typically have arrays. Correspondingly, there is a big difference in the operational paradigm between the two fields: in databases the emphasis is on descriptiveness, set-orientation, and optimizability. Programming languages operate navigationally. Data models tend to have few data types (primitives as well as constructors), programming languages provide more flexibility. With the emergence of the object-oriented paradigm in programming languages there was a slight shift in major concerns towards data structuring, while in the database community the object-oriented approach shifts focus towards operational aspects. This is a clear indication for the benefits that can be expected from the current merge of these fields.

The relational data model is described according to our systematics by three components. Consequently, extensions of the relational model can be characterized according to whether and how they extend each one of these

Primitive Types: Typically, only a small number of atomic data types (or attribute domains, in the relational jargon) are offered for numbers (integers, reals) and strings. Some systems include further types, such as date, money, time.

Type Constructors: The essence of the relational model is that it provides exactly one such constructor: **relation** (which is a combined set-of-tuple constructor). The relation constructor can be applied to primitive types only (which then become the domains of tuple attributes).

Operators: Typical formal languages for relations include relational algebra and relational calculus. Practical languages are almost exclusively in the SQL style today. We will use relational algebra as our reference language, since it fits best with our systematics. Relational algebra has five independent operators: Set Union ($\cup$), Set Difference ($-$), Relational Product ($\times$), Relational Selection (σ), and Relational Projection (π). Almost all theoretical work on the relational model excluded operations on primitive types, except for comparisons ($=, <$). But in general, operators can be categorized into a collection of operators for each primitive type and one for each type constructor.

In the next section, we will outline research work in each of these directions: Section 2.1 describes extensions to the set of primitive (base) types; Section 2.2 gives a first overview of extensions to the structuring capabilities (i.e., the type constructors); Section 2.3 briefly points to work on query language extensions that are not related to extensions of the type system (these are covered in more depth in a separate chapter of this volume). The main body of the paper concentrates on the family of nested relational or complex object models (Section 3). We discuss the rationale behind these extensions and survey theoretical as well as practical work, including implementation efforts. Main emphasis is on the impacts of the enhanced modeling capabilities on the query language design. Section 4 gives an outlook on how this work is carried over to the area of object-oriented data models by integrating concepts of semantic data models or AI knowledge representation languages. Finally, we conclude with a summary.

2 Survey of Relational Extensions

2.1 Extending Base Types

The first class of extensions to the relational model addresses the primitive data types. There are two major concerns: one is to include (more) operations on the attribute level. This is implemented in most commercial systems. In SQL we can express, for example, arithmetic operations to numeric attributes as in: "`SELECT name, 13*salary FROM employees`". Formal languages such as algebras or calculi typically exclude such operations (because they do not want to consider function symbols). The second concern is to enrich the collection of base types. Again, database theory typically has just numbers and strings, whereas systems offer other types, such as date and money. More challenging, however, is to offer an *extensible* set of base types: Each installation of the DBMS can add its own, specialized data types. The archetype for such application-driven demands are certainly all kinds of geometric data: points, polylines, polygons, and the like. Other examples include text, or such 'simple' extensions as banking dates (360 days a year).

Obviously, as far as the data model is concerned, these extensions can be handled fairly straightforward: what we need is a good, orthogonal design of the database (query) language, such that operations on the attribute level can be used in selection predicates (i.e., SQL where-clauses) as well as in projection lists (SQL select-clauses). Furthermore, in case of extensibility, we need some syntax for declaring new types and operators.

The critical question is how to provide an extensible database system architecture. Several projects are well-known for their efforts in this direction: Probe [79, 35], Exodus [22], Starburst [68, 50], DASDBS [86, 95], to name just a few. Central problems besides general architectural considerations are how to provide access path support and special storage structures for such user-defined data types [119, 131, 132], and how to make the operations, their optimization rules, and execution strategies and costs known to the optimizer [46, 70, 51, 47, 13]. Some of the ideas have made it into commercial products already. For example, the Ingres Release 6.3 "Object Management Extension" [90] allows applications to include new data types and operations on them into the system. Access path support (by B+ trees) can be used, if the new type has an order relation ($<$).

As this kind of extension of relations is more concerned with architectural issues than with the data model itself, we will not elaborate in more detail, but rather refer the interested reader to the references given above and the article on the Genesis project elsewhere in this volume. Extensible systems such as Starburst are full database systems in the sense that they provide a skeleton architecture that can be extended in several, but predetermined ways. "Kernel" systems, such as EXODUS and DASDBS, can be extended mainly by building frontends on top (even though some degree of extensibility might also be possible within the kernel). In that sense, they are "powerful file systems" rather than complete DBMSs. Lastly, the toolkit approach of systems such as Genesis aims at providing a kind of DBMS development framework with predefined interfaces and libraries of useful code, akin to the graphical user interface (GUI) toolkits for the X-Windows system.

2.2 Extending Type Constructors

In this section we survey extensions of the relational data model that aimed at increasing the structural modeling power. Relations are, due to the First Normal Form condition, "flat tables": all attribute values have to be atomic. If we consider the relational model as a type system, this means that there is only one type constructor, **relation**. Furthermore, this type constructor can be applied exactly once, all component types have to be base types. The background of the relational model is mathematical set theory, and in fact, a relation is often formally defined as a subset of a Cartesian product (of the attribute domains). In programming language terms, we would say a relation is a **set of** (labeled) **record**s. The two type constructors (set, record) come in a pair, no recursive application is allowed, all record fields are of base types.

We can distinguish three approaches to extending the type system as far as constructors are concerned: (1) we can stick with the one type constructor, relation, but use it repeatedly: this yields *nested relations*; (2) we can separate the two parts of the constructor relation and have separate **set** and **tuple** constructors, that are also applied orthogonally: then we have "*Complex Objects*" (today, in view of object-orientation, we would rather say Complex Values); (3) finally, several proposals have been made to include other type constructors (such as lists, multisets, arrays, ...). Figure 1 gives a graphical summary of (1NF) relations, nested relations, and "Complex Objects" using the notation from [56].

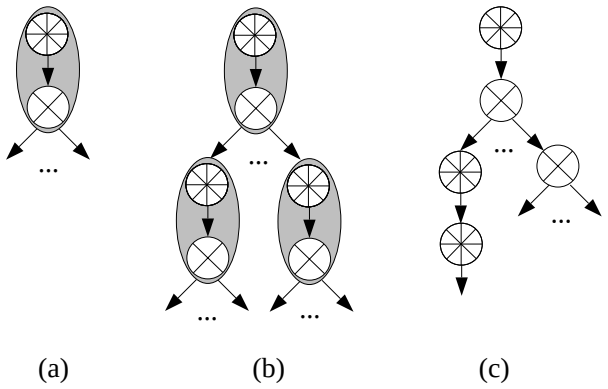


Figure 1: Flat Relations (a), Nested Relations (b), and "Complex Objects" (c)

Example 1 The following is an example of three schema definitions in an imaginary syntax using **record** and **set** as constructors. In (a) we see the traditional flat relations describing departments and employees, respectively. Part (b) shows one Nested Relation, where the employee tuples are grouped into a subrelation for each department. Notice that in order to obtain a nested relational structure the type definitions of relations have to use a strictly alternating tuple-set sequence and, more importantly, they must be non-recursive [44, 112].[1] This restriction leads to a purely hierarchical model. Due

[1] Furthermore, if we allow a record type to occur within the definition of more than one superordinate record types the usual interpretation taken over from classical programming languages is that the two relations share the *schema* of a subrelation, but the *values* of them are independent from each other. There is no subtuple sharing!

(a) flat relations

Empl(eno,name,salary,dno)
Dept(dno,dname,budget)

```
type Emptup = record
               eno:  int,
               name:  chrstr,
               sal:  real,
               dno:  int
              end;
     Emprel = set of Emptup;
     Deptup = record
               dno:  int,
               dname:  chrstr,
               budget:  real,
              end;
     Deprel = set of Deptup;

var  Empl:  Emprel,
     Dept:  Deprel;
```

(b) nested relations

Dept(dno,dname,budget,Empl)
Empl=(eno,name,salary)

```
type Deptup = record
               dno:  int,
               dname:  chrstr,
               budget:  real,
               Empl:  Emprel
              end;
     Deprel = set of Deptup;
     Emptup = record
               eno:  int,
               name:  chrstr,
               sal:  real
              end;
     Emprel = set of Emptup;

var  Dept:  Deprel;
```

(c) "recursively nested relations"

Belongs=Dept(dno,dname,budget,Staff)
Staff=Empl(eno,name,salary,Belongs)

```
type Emptup = record
               eno:  int,
               name:  chrstr,
               sal:  real,
               Belongs:  Deprel
              end;
     Emprel = set of Emptup;
     Deptup = record
               dno:  int,
               dname:  chrstr,
               budget:  real,
               Staff:  Emprel
              end;
     Deprel = set of Deptup;

var  Empl:  Emprel,
     Dept:  Deprel;
```

to the lack of modeling constructs for many-to-many or recursive relationships, it might not be considered general enough as a logical data model. We will come back to this point later in this section, where we will see how the recursive type definitions in (c) can be interpreted. ◇

Multiset (bag) has been proposed as an additional type constructor in order to formally deal with duplicates in relations. Since relational DBMSs typically do not guarantee the set property (duplicate elimination) of tables, a correct formalization has to use the concept of a multiset [34, 32, 128, 61]. Also, if SQL's aggregate operations shall be formalized, multisets are necessary. Alternatively, or in addition, one can introduce lists to cope with ordering (for example in order to express the physical storage sequence of tuples, to formalize the ORDER BY clause of SQL, or as a conceptual tool for modeling order structures). Examples for this can be found in [88, 89, 48]. Further extensions could be made to include arrays, variants, or the like.

In any case, we are not interested in extensions to the type system per se: we expect each constructor to contribute a collection of generic operators for access and manipulation of instances of the constructed type to the language. Furthermore, if several type constructors are included, we will often need conversion functions (such as from set to multiset, from list to set, and so on). A good design of the type system with these generic operators will always be *orthogonal* such that structures as well as operations can be obtained by combining these building blocks appropriately.

Throughout the rest of this paper, we will not elaborate on type constructors other than **set** and **record**, to avoid too many operators. The principal objective of how to design a query language for an orthogonal type system can be illustrated with these two constructors as well.

2.3 Extending Query Languages

The concept of relational completeness of a database language has been set up be Codd in [27]. A relational query language is said to be (relationally) complete if it is at least as powerful as relational algebra (or, equivalently, relational calculus). However, for all practical purposes this is not enough. Virtually all implemented relational DBMSs provide languages with more expressive power. Examples are SQL's aggregates, built-in functions, ordering, grouping, duplicates, and last, but not least updates. Most of these aspects have been neglected in database research (with some exceptions concerning SQL's retrieval features, see the previous section). Particularly, updates have often been excluded from formal work until recently (see, for example [1]).

According to our systematics, we will discuss those extensions of query languages that result from increased flexibility in the type system together with these extensions. Other enhancements, that are independent from the type system, usually concern recursion. Since this volume contains separate articles on logic and databases, we will not detail this aspect here. Some approaches to recursive query facilities that are not based on rule languages and take advantage of some form of nested relations or other similar structures are described in [114, 60, 31, 69], for example.

3 Nested Relations and Complex Objects

In this section we go into some detail of nested relational databases. We will discuss the rationale behind this (family of) data model extensions, mention some implementation efforts, and summarize the theoretical work that has been done in this context. The focal point of our presentation, however, will be the consequences for query languages.

Nested relations are an attempt to extend the structuring capabilities without introducing new concepts, rather the known concepts are exploited in more depth. So nested relations are a very conservative extension. Essentially, the only change as compared to traditional relations is that the one type constructor **relation** can now be applied repeatedly. In other words, the First Normal Form restriction is dropped, attributes need no longer be atomic (i.e., of a base type), they can also be structured. However, all structured attributes are again relations (relation-valued attributes). The original names expressed this fact: "unnormalized" or "non-first-normal-form (NF^2,$\neg$1NF)" relations. Therefore, as far as structuring goes, nested relations re-introduce the hierarchical data model. The essential difference, though, lies in the style of operators that come with the model: relational algebra, relational calculus, or SQL are extended in several ways to deal with these hierarchical structures.

There have been practical as well as theoretical reasons for investigating nested relations:

- the increased flexibility of data structures allows a more direct mapping of applications onto the database (improved *logical design*);
- the hierarchical structures of nested relations can be used to describe a wide variety of *physical database designs*;
- nested relational query languages can nicely express some of the "awkward" SQL-features, such as grouping;

- the conceptual gap between relational and nested relational models is minimal, so this extension should be acceptable for users;
- when extending a formal system, it is best to move forward in small steps; to stick with one concept promised to be able to carry over most of the established theoretical and technological background (such as design guidelines, normal forms, optimization and implementation techniques, and expressive power/completeness of query languages);
- other extensions to the model, if orthogonal, can be added later.

In the sequel we highlight some of the research that has been carried out in the context of nested relations.

3.1 Normal Forms, Design Theory

Among the earliest mentions of extended relational models is [72], in the context of normal forms. Later, work in several groups has taken up this subject for nested relations [43, 82, 91, 83, 55]. Partitioned Normal Form (PNF) [93] and Nested Normal Form (NNF) [82] have been proposed as preferable relational structures. Partitioned normal form basically requires the key of a nested relation to consist of atomic attributes only and, recursively, that the same holds for all subrelations. The effect is that no subtuples are replicated within several superordinate tuples, since the atomic attributes can be used to partition the set of all subtuples. The department-employee relation in Example 1(b) is supposedly in PNF, since the key of departments is usually *dno*, an atomic attribute, and the key of the subrelation employees is usually *eno*, another atomic attribute. The nested normal form of relations basically states that no redundancy is introduced by the nesting structure, a criterion that is formalized using MVDs and FDs [83]. Algorithms for obtaining NNF relations are also given in these papers.

As an informal summary of this work, we conclude that nested relations provide a way of directly representing certain multi-valued dependencies (MVDs) by nesting. Therefore, during logical database design, fewer relations have to be split into smaller pieces (decomposition). Instead, often the dependent attributes can be grouped into a subrelation, e.g., employee attributes can be grouped per department instead of decomposing into two relations. While relational database design often ends up in a large number of supplementary tables (due to extensive need for decomposition in case of set-valued "attributes"), nested relational databases can keep the information closer together. Joins that are necessary to recollect all data in relational databases become unnecessary with nested relations, since more data are contained in one tuple. On the one hand, this makes query formulation easier for the user, on the other hand, this observation spawned another research direction.

3.2 Nested Relations as Storage Structures

The possibility of internally materializing frequent joins was the rationale for using nested relations in physical database design. Regardless of the kind of logical data model, nested relations are perfectly suited to describe internal database layouts [97]. All kinds of (schema-driven, static) clustering strategies can be modeled using appropriate nested relations [109].

Therefore, the Verso and DASDBS projects aimed at implementing nested relations as storage structures [11, 105, 37, 86, 95]. When physical database design is described by such a high-level abstract model, the transformation of queries from the logical to the physical level can be performed by formal manipulation within a (nested) relational algebra. Algebraic query optimization techniques can be used to transform and optimize queries into expressions of some nested algebra that are directly used as executable query plans. The major objective in the transformation is to recognize and eliminate those joins from the users' queries, that are redundant since they are internally materialized [18, 106, 19].

For an illustration, let use refer back to the relations from Example 1: let us assume a standard relational interface to the database, that is, at the logical level, the user sees the relations from Example 1(a). We assume that users will often join the two relations (on the *dno*-column), so, internally, a good physical design would be to materialize this join (assuming a relatively low update frequency). A materialization in form of a nested relation does not introduce any redundancy in case of a $(1:n)$-relationship as in our example, so the database administrator may indeed decide to store the nested relation from Example 1(b). Now we expect a significant performance gain since the system no longer has to compute the join at query execution time, rather it can just retrieve the precomputed join (or parts thereof).

The interesting research topic that had to be solved in this context was: How can the query optimizer transform the join query given in terms of logical relations to the single-table query on the internal relation? And, can we utilize the large body of relational query optimization knowledge that had been developed before? The latter aspect reflects our overall goal with nested relations, namely to extend the scope of relational theory with only minimal effort. Both attempts, [106] and [19], succeeded in solving this optimization problem with mostly relational techniques. The approach of [106] is to realize the query transformation in the same way as views are normally implemented: by query rewriting. That is, we define the logical-level relations by queries over the internal relations (projections and unnesting are needed in case of hierarchical join materialization). When a user query is mapped to the internal level, these queries are simply substituted for the names of the logical-level relations in the query expression. The result of this textual substitution is a query expression in terms of internal relation. However, it needs to be optimized. Let us exemplify the approach and the ensuing problems by continuing Example 1:

Example 2 Omitting some details of dealing with null values that are necessary for an information-preserving transformation, the mapping between the flat relations in Example 1(a), as the logical-level relations, and the nested relation from Example 1(b), as the internal database layout, can be described by the following expressions in the nested relational algebra of [98]:

$$\begin{aligned} IDept &\equiv \nu_{IEmpl=(eno,name,salary)}(LEmpl \bowtie LDept) \\ LDept &\equiv \pi_{dno,dname,budget}(IDept) \\ LEmpl &\equiv \mu_{IEmpl}(\pi_{dno,IEmpl}(IDept)) \end{aligned}$$

Notice that we prepended an "L" to the logical-level relation names and an "I" to the internal-level names.

Now, let us consider a query asking for part of that join, as expressed by the user in terms of logical relations; also shown is the same query transformed to the internal level by simply substituting the expressions defining logical relations in terms of internal ones:

$$
\begin{array}{rcl}
LQ & = & \pi_{dname}\Big(\sigma_{name=\text{“Joe''}}(\quad\quad LEmpl \quad\quad) \bowtie \quad\quad LDept \quad\quad\Big) \\
& \Downarrow & \\
IQ & = & \pi_{dname}\Big(\sigma_{name=\text{“Joe''}}(\overbrace{\mu_{IEmpl}(\pi_{dno,IEmpl}(IDept))}) \bowtie \overbrace{\pi_{dno,dname,budget}(IDept)}\Big)
\end{array}
$$

Obviously, this query needs further optimization, if the join should actually be removed. In [106] we showed, that the query can be simplified, the redundant join can be eliminated using (flat) relational techniques, such as tableaux. The necessary extensions to the query optimization rules include (i) dealing with null values and outer joins, and (ii) after elimination of the join transforming the resulting expression into one where unnesting appears as late a possible. We will not give the detailed steps here, the final result for our example query would be:

$$IQOpt \quad = \quad \pi_{dname}(\sigma_{\sigma_{name=\text{“Joe''}}(IEmpl)\neq\emptyset}(Idept))$$

In this nested relational query against the internal relation, a simple nested selection followed by a standard projection computes the query result. ◇

In general, we could show that any select-project-join query on the logical level, where all the joins are internally materialized, map to one single-pass query. This class of nested relational queries can be evaluated with linear complexity [106, 94]. A main result of our work was that the transformation and optimization problem is solvable using mainly known techniques, no significant complexity is added by using nested relations instead of flat ones.

To conclude this aspect, we summarize that nested relations are an elegant way of describing physical database layouts, since they can express arbitrary schema-driven clustering techniques.[2] The major strong point of doing so is that query transformation and optimization can then be completely solved within the formal framework of an algebra. Similar results have been obtained in [19] for the Verso project using a purely tableaux-based approach. Summaries of the experiences with the Verso and DASDBS projects have been presented in [105] and [95], respectively.

3.3 Other Implementations of Nested Relations

Besides the Verso and DASDBS projects, who aimed at directly implementing nested relations in a one-to-one mapping to storage structures, there have been a number of further projects investigating other implementation choices. In the AIM-P project [29, 30] with its extended nested relational model, the main emphasis was on the user query language (HDBL,

[2] In order to deal with indexes, link fields, and $(n : m)$-relationships, reference attributes have to be introduced on the internal level that model tuple addresses, see [109].

see Section 3.6). Their implementation of nested relations uses an internal structure that does not necessarily reflect the nesting in the physical page layout (see [36] for a comparison with the DASDBS approach). A rather elaborate directory concepts was developed instead, that also allows for flexible navigational access (nested cursors).

Another implementation of nested relations, ANDA, completely separated the atomic values occurring in tuples (in a structure called VALTREE) and the hierarchical structure of tuples (in a structure called RECLIST) [38, 39]. The idea behind these two tightly coupled structures was to separately support value-driven operations, such as selections and joins, and structure driven operations, such as projections and unnesting, by the two specially optimized mechanisms.

A number of further projects implemented nested relational interfaces on top of some other DBMS, usually relational ones. These implementations can serve as a testbed for query language research. Performancewise, they suffer from the fact that all subrelations have to be mapped to separate tables. Therefore, the large number of joins is still necessary (on the internal level). Unless the underlying relational system provides (or is modified to provide) some access path or clustering support for accelerating joins, such as join indices [125] or ORACLE-style clustering [78], these "on-top" implementations are not likely to show good performance. One example of such an implementation on top of a research relational DBMS is described in [52].

Before we turn to query languages, let us comment on the idea of "Using QUEL as a data type" [121]. This idea finally led to the POSTGRES approach of implementing complex objects by attributes, whose values are query language expressions that dynamically retrieve the "subobjects". If we stick with relations and relational QUEL as the query language with this nesting capability, then we obtain exactly nested relations. If a relation R has a 'QUEL-valued' attribute $R.A$, then this relation R is in fact a *nested relation*. POSTGRES [122, 118] adds other features, such as abstract data type domains or rules to the standard repertoire of a relational DBMS, but in this respect, it is *very* similar to nested relational systems.

3.4 Query Languages

An enormous amount of work on nested relations was devoted to query languages. Among the proposals are languages following each of the following paradigms: algebraic, calculus, SQL-style, and rule-based. Before we go into details, let us characterize the overall approach. From the (flat) relational model, we have a number of query languages following the different paradigms, that is, there are well-understood ways of querying (and manipulating) relations. The advantage of the pure nested relations approach to an extended data model is obvious: in principle, we need no new query language at all! All we need is the freedom to apply the constructs of any one of the relational languages wherever relations occur. More formally, we need a language design that is fully *orthogonal*: since relations may now occur not only on the outermost level, but also nested within tuples (as relation-valued attributes), relational operators should be applicable on the attribute level, too.

Another approach to a nested relational query language, that was pursued only in algebraic languages, consists in the addition of two operators, Nest (ν) and Unnest (μ). Actually,

the initial papers on nested relations had just these operators and did not discuss the other extension of applying the algebra on the attribute level [96, 59, 44]. The idea was that whenever relation-valued attributes are to be manipulated, one could first unnest, apply the standard relational operators and finally renest to obtain the desired result. However, it was clear from the beginning that (i) this cannot work in general, since unnesting may not be reversible by nesting [59, 44], and (ii) this will not be an efficient way of computing the results. These two observations have spawned two streams of investigations in nested relational query languages. One was more of a theoretical nature, namely to investigate the expressive power of nest/unnest-extended algebras and comparing them with nested relational calculi. The other aimed at languages that were suitable as the basis for efficient implementations. In this case, "deep" algebras have been proposed that allow for the manipulation of subrelations without unnesting them first.

The minimal extension of just adding two new operators, nest and unnest, to the standard algebra depends on the nested relation to satisfy certain constraints, such that unnesting can be undone by nesting. In [59] weak multi-valued dependencies were used to characterize such nested relations, and a third new operator, Keying (κ), was introduced to allow arbitrary nested relations to be losslessly unnested. Essentially, what is required is that the relation has a key consisting of only atomic attributes, that is, be in Partitioned Normal Form (see above). So, in addition to database design, this kind of query languages was another reason to study normal forms for nested relations. An analysis was presented in [45], to detect whether a nested relation can be obtained from a flat one by a sequence of nest operators (in this case, the relation can also be unnested losslessly). Investigations concerning the completeness of such languages have been presented in [127, 84], where it was shown that such an algebra is "BP-complete".

"Deeply nested" algebras, that is, those that allow operations on the attribute level have been proposed in [97, 4, 58, 98, 40, 81, 93, 3, 28, 126]. Some of them require the relations to be in partitioned normal form. The algebras of [93, 28] provide operators, such as the set operators, union and difference, that apply recursively to all subrelations, therefore they need the PNF restriction. The other algebras provide explicit nesting of algebraic expressions into the arguments of their operators. We will discuss some examples using the syntax of [98] and the nested department-employee relation from Example 1(b): $Dept(\underline{dno}, dname, budget, Empl(\underline{eno}, name, salary))$.

Example 3 In the flat relational algebra there are only two places where attribute names occur in expressions: in selection predicates and in projection lists. Consequently, these are the points where the nested relational algebra allow the application of algebraic expressions to relation-valued attributes.

We may select only those departments having at least one employee named John by the following selection query Q_1:[3]

$$Q_1 := \sigma[\,\emptyset \neq \sigma[name = \text{“John”}](Empl)\,](Dept).$$

[3]Here and in the sequel, we write selection predicates and projection lists in square brackets after the operator symbol instead of as a subscript.

Every *Dept*-Tuple has a (relation-valued) attribute *Empl*, so we may apply a predicate on the *Empl*-value for the outer selection. Since, for each *Dept*-tuple, *Empl* is a relation we can select from this relation the set of tuples satisfying the inner predicate ($name$ = "John"). Whenever this nested subquery returns a nonempty result for a *Dept*-tuple, we retain this *Dept*-tuple in the query result, otherwise we discard it. Notice that if a department qualifies, it is contained in the query result as a whole, particularly with *all* employees, not only those named John. This is because the inner selection just serves for predicate testing.

Now consider the case where we want to see, for all departments, only those employees named John. Here we have to retrieve a modified subrelation for each *Dept*-tuple. This is achieved by nesting a selection (with predicate $name$ = "John") into a projection on *Dept*:

$$Q_2 := \pi[\, dno, dname, budget,\, \sigma[name = \text{“John”}](Empl)\,](Dept).$$

The result will contain one tuple per *Dept*-tuple, each with for components: *dno*, *dname*, *budget*, and a subrelation *Empl* containing only those employees of that department who are named John. Notice, that this subrelation will be empty for all departments without John's among their staff. If we want to discard those departments (without John's), we could apply a simple selection afterwards:

$$Q_3 := \sigma[\, Empl \neq \emptyset\,](Q_2).$$

Notice that this query Q_3 is the nested relational equivalent of the flat relational select-join query. ◇

All relational operators can be nested into selections or projections. The valid operands of nested operators are determined by a scoping rule ("dynamic constants" in [98]). Essentially, not only the subrelations at each nesting level can be used, but also other top-level relations. For deeply nested relations, also subrelations that lie along the path from the current point in the hierarchy up to the root are valid operands. We will not go into details here, they can be found in [107, 108]. However, it turns out that this kind of nesting algebra operators into others is quite powerful: Joins (Products), Difference, and Nesting can be simulated by (nested) selection and projection alone.

Other results concerning the expressive power of nested algebras are due to [3, 15], where a similar algebra and a calculus are defined and it is shown that the algebra is equivalent to the "strictly" save calculus and the algebra with an additional powerset operator is equivalent to the save calculus.

In the previous Section 3.2 we have already seen that such a "deeply" nested algebra can express efficient query evaluation plans. In contrast to "nest/unnest"-extended algebras, a query processor can directly execute a nested algebra expression, such as the ones shown above, without the need for devising another strategy (unless additional indexes are present, that is, if the query is to be executed as a scan).

One other comment is in place: the POSTGRES extension of supporting attributes whose 'value' is a query language expression can be realized by nested relational algebra

too. Suppose a relation R has a 'QUEL-valued' attribute $R.A$, with a value of some query expression *expr*. Then the retrieval of R-tuples together with their A-values is expressible in the nested algebra by $\pi[\ldots, expr, \ldots](R)$. The scoping rule mentioned above is the same in POSTQUEL: the query expression may refer to other relations and to attributes of R.

There have also been a number of SQL-style approaches to nested relational query languages [88, 89, 92, 67]. Their principal idea is the same as in the algebras: SQL-Select clauses (cf. projection lists) can now contain nested Select-From-Where (SFW-) blocks as may the Where clause. With standard SQL, Where clauses could already contain nested SFW-blocks, but these had to refer to relations, now they can refer to subrelations too. One of the major problems with SQL, its lack of orthogonality, had to be attacked, in order to clean up syntax and semantics of the SQL-based approaches. To our taste, [67] succeeded best in this respect. SQL/NF [92] is based on PNF relations, HDBL [88, 89] operates on "extended" NF^2-relations that contain other type constructors in addition to relations. Therefore, its syntax is necessarily more complex since it has to distinguish tuples, sets, and lists (see under Section 3.6 for more details on HDBL).

Example 4 The query Q_1 (nested selection) from above would be expressed in SQL/NF [92] as:

```
select *
from   Dept
where  exists ( select *
        from   Empl
        where  name = 'John' ).
```

Query Q_2 (the nested projection) would be:

```
select ( select *
         from   Empl
         where  name = 'John' )
from   Dept.
```

Without the opportunity to nest queries into the From clause, i.e., without query composition, query Q_3 from above would have to duplicate the subquery:

```
select ( select *
         from   Empl
         where  name = 'John' )
from   Dept
where  exists ( select *
        from   Empl
        where  name = 'John' ).
```

Therefore, SQL/NF can name a subquery within the Select clause using a key word "as". HDBL [88, 89] introduces a statement "let" to define subquery-expressions once and use it multiple times in a nested SQL statement. SQL/W [67] adds a "with" clause for the same purpose.

```
select ( select *
         from   Empl
         where  name = 'John' ) as Johns
from   Dept
where  exists Johns.
```

In addition, SQL/W allows SFW-blocks in the From clause (i.e., composite queries), such that the query could also be formulated as:

```
select *
from   ( select ( select *
            from   Empl
                   where  name = 'John' )
         from Dept )
where  exists Empl.
```

Notice that here the predicate "exists Empl" refers to the result of the inner selection that has been executed before. Actually, SQL/W allows a number of syntactic simplifications that make query expressions shorter.

```
Dept.Johns
where exists Johns
with Johns = ( Empl where name = 'John' ).
```

Here "Select * From" has been omitted, and "Select Johns From Dept" has been replaced by "Dept.John". The "with" construct is the same as "let" in [88, 89] and "as" in [92]. The most compact expression in SQL/W is not much different from an algebra expression since most syntactic sugar can be left out:

```
( Dept.( Empl where name = 'John' ) ) where exists Empl.
```

◇

Besides algebras, calculi, and SQL-style languages there has also been some work in the direction of extending rule-based languages (DATALOG) to deal with complex structures. In [3] a rule language has been presented and compared with the algebra and the safe calculus. LDL has also been extended to deal with set valued attributes ([17], see also the chapter of this volume). Another calculus for complex objects has been presented in [10]. Recent research dealing with set values in logic-based languages, such as [6, 2, 7, 25, 54, 62, 64, 71, 21] was carried out in the framework of object-oriented models, but the impacts of set-values are the same in an object or tuple component. Namely that the nesting of formulae into attribute positions in logical terms makes the language second-order (at least syntactically), since this leads to quantification over set values. In order to stay within first-order calculus, such languages have to be given a first-order semantical interpretation. F-logic [62, 63] may be considered a state-of-the-art language doing this.

3.5 "Complex Objects"

By Complex Objects in the sense of [3] we mean data structures that are obtained by some arbitrary application of record and set constructors over a collection of base types (cf. Figure 1(c)). That is, in contrast to nested relations with their combined type constructor relation, complex objects have two separate constructors for records (tuples) and sets. This separation does not introduce much more modeling power, but it add to the flexibility. Further, we use this model to illustrate the general idea of how each type constructor should support its own collection of generic query (and update) operators.

Recall the operators of the relational algebra: among the five independent operators, namely union, difference, product, select, and project, the first two are the standard set operators. That is, they depend in no way on the fact, that the elements of relations are *tuples*. Relational Product is different from standard set theoretic product since it "collapses" the resulting pair of tuples into one tuple. This is because tuples of tuples would be outside the relational model. With the flexibility of Complex Objects, we could use the standard Cartesian Product. The next operator, select, is also independent from the fact that relation elements are tuples: it selects those elements of the input set that fulfill the given predicate. It is the predicate that has to "match" the structure of the set's elements, tuples in case of relations. Selection as such need not be changed if other set elements are permitted, but other kinds of predicates may become necessary or useful. Finally, projection seems like a real relational, that is, set-of-tuple, operation. But already the nested relational extension of nesting algebraic expressions into a projection list shades another light onto the project operator: it is now considered as a (descriptive) set-iterator combined with a tuple generator. This means, we interpret projection as iterating over a set and for each element of that set applying a tuple construction, where each component of the resulting tuple is obtained from evaluating one of the expressions in the projection list.

This way of interpreting the relational operators is very similar to the functional data model approach [117, 74, 35]. In PDM, for instance, nested relational projection with embedded algebraic expressions would be simulated using the "apply_append" operator (see also the Complex Object algebra of [3]). The benefit of these considerations is that we now have a better understanding of how query languages should be designed such that new type constructors can be added without a need for redoing the query language design: each type constructor (record, set, bag, list, array, ...) should come with a collection of generic operators. If the query language is designed orthogonally, then extensions are straightforward. The query languages presented in [3] obey to this principle. Evidently, all the results obtained there apply to nested relations too, since they are a special case. Conversely, most of the work on optimization or implementation of nested relations carries over to the generalized case of complex objects, since we are still working with sets and tuples, just in a more flexible way.

3.6 Other Type Constructors

Some of the work on extending the relational model departed further than the nested relational/complex object approaches by introducing other type constructors, such as multisets (bags), lists, or arrays. Also, some approaches started from Entity/Relationship-style mod-

els and designed query facilities for these. They can also be viewed as providing other type constructors (entity, relationship). The extended NF^2-model of [88, 89] has already been mentioned above. There, in addition to sets and tuples, lists (that is, ordered multisets) are provided. These three type constructors can be used orthogonally, such that sets are not privileged as the 'outermost' constructor. The query language, HDBL, is in the nested relational spirit, but includes explicit constructors for sets, tuples, and lists. Since there is no default, queries tend to become a little cumbersome. The ERC model of [85], the EER model of [42], the MAD model of [75, 76], the FAD language [9], the NST-algebra [48] and the Algres system [24], the DBPL language [104], and the EXTRA model of [23], are other examples exhibiting the same query language characteristics: in the same way that type constructors can be nested, query languages have to provide nesting of expressions.

Many of the complex object models included identifiers (reference attributes, surrogates) [94, 89, 88, 75, 9, 109, 23, 67] to support non-hierarchical structures. Whereas hierarchical structures are directly represented by nesting, more general structures need some form of "pointers". With the inclusion of references, the complex object models start becoming object-oriented. If we take a closer look at many of the recently proposed object-oriented data models, we will find almost all of the nested relational/complex objects query language concepts. This is no surprise, since these new models also typically include ways of nesting objects into others.

Some of the models and query languages are either direct followups of the work on nested relations/complex objects or at least have been developed in the same research group (such as [16, 41, 112, 111, 49, 33]). Others, even though developed independently, show the same similarities (for example [80, 12, 130, 116, 8, 124, 73]). This is a clear indication that the results on complexity, expressive power, optimization, and implementation issues obtained in the context of nested relations/complex objects can be utilized in the new field of object-oriented databases. For example, most of the equivalences stated for object algebras in [80, 115, 123] are the same as their nested relational counterparts [106, 19, 40, 28]. In the next subsection, we briefly sketch one approach to make direct use of nested relations in the context of object models.

4 From Relational to Object Models

The work on nested relations that has been carried out in the DASDBS project is continued in the COSMOS project [100, 101]. Particularly, the COCOON subproject works on an object-oriented data model and its implementation that is based on nested relations. The model is obtained as an evolution [112, 99] from relational/nested relational models. It has been presented in detail in [111], here we will just outline the underlying ideas. The key issue in the project is to design an object-oriented query language that is "almost" a (nested) relational algebra.

In Example 1 we have seen flat (a) and nested (b) relational type definitions using the record and set type constructors. Also shown was a collection of type definitions that was recursive (c). It was not a priori clear, what such recursive type definitions should mean, and the nested relational approaches that use equations for the definition of relational schemas explicitly rule out recursion [44, 93]. Also, the generic semantic data model of [56] does not

permit recursive use of the set and/or tuple constructors (association and aggregation): the construction shown in Figure 2(a) is not allowed. The situation has to be modeled using at least one *function* between types. In Figure 2(b) we have shown a valid modeling using two functions between the "recursive" types.

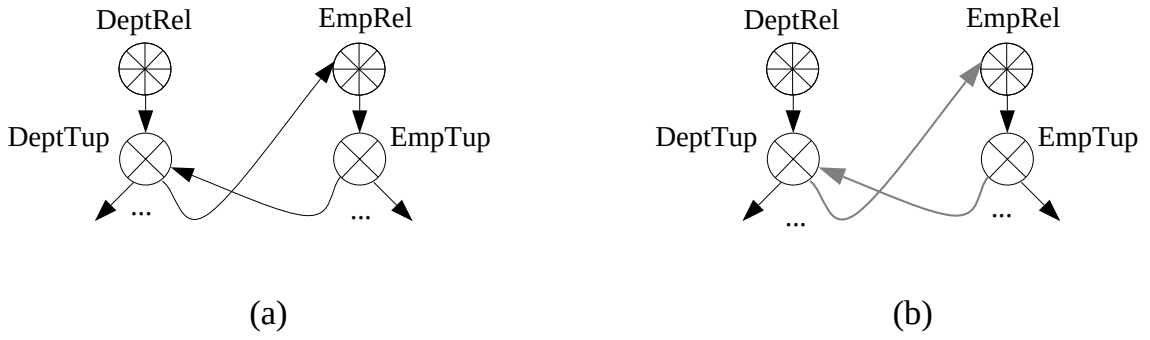


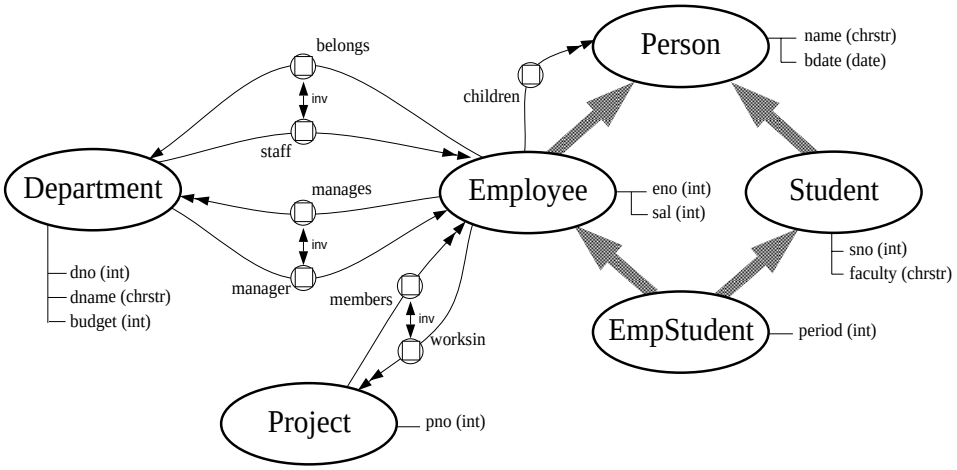
Figure 2: Example of a forbidden type construction (a) and a solution using functions (b)

This solution originates from functional data modeling [117] and has been taken up in many of the recent object-oriented approaches [130, 33, 16, 77, 111, 73]. Since by introducing functions instead of just composite structures the objects and the "relationships" between them (expressed by functions) are separated, the recursion is broken. Pure object-function models go one step further: once there is the need to support functions (in order to brake recursion), we can dispense with the tuple constructor and model all tuple-component-relationships by functions. This is not a necessary consequence, but just a way to keep the number of essential constructs as small as possible. The COCOON project follows this approach.

The idea is to allow the recursive type definitions shown in Example 1 by interpreting them as an object-function model. Each of the type definitions introduces an abstract object type together with some functions. What used to be regarded as attributes in the relational model, is now considered a function whose domain is the type in which the function appears and whose range is the type occurring after the function name. Figure 3 gives an example of a graphical and linear schema description for an example database. Besides the "recursive" structure, the example also shows inheritance (is-a).

The graphical representation of the COCOON schema suggests another interpretation of the model: it can be viewed as a linearized syntax for the definition of the generic concepts of a KL-ONE network [20]. That way, we have built a bridge between databases and knowledge representation. It is not just the form of graphics that are used to illustrate COCOON schemas; also the semantics of the model, particularly its update semantics [66], is based on automatic classification, the key issue in KL-ONE systems [113, 110].

The interesting point about this model for our presentation here is that even though the structural part of the model is substantially different from nested relations, the query language can essentially be a nested relational one. The principal idea is that, for each query, we take a hierarchical view onto the object network and formulate a query against that hierarchical view. For different tasks we take different hierarchical views, so the network




```
type DEPARTMENT is-a OBJECT =
                    (DNO:     int,
                     DNAME:   chrstr,
                     BUDGET:  int,
                     MANAGER: EMPLOYEE inverse MANAGES,
                     STAFF:   setof EMPLOYEE inverse BELONGS);

type EMPLOYEE   is-a PERSON =
                    (ENO:     int,
                     SAL:     int,
                     CHILDREN:setof PERSON,
                     WORKSIN: setof PROJECT inverse MEMBERS,
                     BELONGS: DEPARTMENT inverse STAFF,
                     MANAGES: setof DEPARTMENT inverse MANAGER);

type PERSON     is-a OBJECT =
                    (NAME:    chrstr,
                     BDATE:   date);

type STUDENT    is-a PERSON =
                    (SNO:     int,
                     FACULTY: chrstr);

type EMPSTUDENT is-a STUDENT, EMPLOYEE =
                    (PERIOD:  int);

type PROJECT    is-a OBJECT =
                    (PNO:     int,
                     MEMBERS: setof EMPLOYEE inverse WORKSIN);
```

Figure 3: Example database schema as a KL-ONE network and as recursively defined types

is used as a symmetric representation of many-to-many relationships among objects where queries can be expressed in any of the contained hierarchical views.

For example, one query could see projects as a "subrelation", WORKSIN, of employees. Therefore, this query can be formulated as if the database contained a nested relation $EMPLOYEES(eno, name, \ldots, WORKSIN(pno, \ldots))$. On the other hand, another query could prefer the opposite structure and be expressed as if the database had a nested relation $PROJECTS(pno, \ldots, MEMBERS(eno, name, \ldots))$. That is, it could see the employees as a "subrelation", MEMBERS, of projects. Since this idea works in general, we can in fact use a nested relational query language over a non-hierarchical database. This explains why all of the object-oriented query algebras proposed so far show such strong similarities with nested relational/complex object languages. Further, it points a way of how object-oriented databases can take advantage of the research results obtained for these languages, such as complexity analysis, completeness considerations, query optimization, and efficient execution plans.

Example 5 Let us consider one query for each of the "hierarchical views" mentioned above. Query Q_1 likes to see projects subordinate to employees, since this is the natural choice for expressing the selection condition:

Q_1: Who is working in projects named "OODB" that started in 1988?

Query Q_2, on the other hand, is naturally expressed using the other direction of the many-to-many relationship for symmetrical reasons:

Q_2: What projects involve employees named "Smith" living in "Zurich"?

Below we give one possible formulation of each of the two queries in COOL [111], RELOOP [26], and OODAPLEX [33].

The COOL language we developed in the COCOON project is basically our nested algebra applied to the COCOON object model:

$$Q_1^{\text{COOL}} = \textbf{select}[\emptyset \neq \textbf{select}[pname = \text{“OODB”} \wedge start = 1988](Worksin)](Employees)$$
$$Q_2^{\text{COOL}} = \textbf{select}[\emptyset \neq \textbf{select}[name = \text{“Smith”} \wedge livingin = \text{“Zurich”}](Members)](Projects)$$

The RELOOP language of the O_2 project is an SQL extension for the O_2 object model. It uses nested select-from-where blocks (the queries shown here look the same in OSQL of IRIS [14]):

```
Q1^RELOOP =  select e
             from e in Employees
             where exists ( select p
                            from p in Worksin(e)
                            where pname(p) = "OODB"
```

```
                                  and start(p) = 1988 )
```

Q_2^{RELOOP} =

```
select p
from p in Projects
where exists ( select e
                 from e in Members(p)
                 where name(e) = "Smith"
                   and livingin(e) = "Zurich" )
```

The language OODAPLEX is an object-oriented extension of DAPLEX, a functional query language:

Q_1^{OODAPLEX} =

```
e in Employees where
exists ( p in Worksin(e)
           where pname(p) = "OODB"
             and start(p) = 1988 )
```

Q_2^{OODAPLEX} =

```
p in Projects where
exists ( e in Members(p)
           where name(e) = "Smith"
             and livingin(e) = "Zurich" )
```

◇

Even though we did not discuss the syntax and/or semantics of object query languages in detail, the three examples given indicate two things: first, nesting of query expressions (in the sense of nested relational/complex object languages) will be useful to deal with the structural aspects of objects. Secondly, the differences between individual object query languages are, to a large extent, syntactical. A vast amount of concepts (and hence implementation techniques) can be shared between them. Some problems, e.g., whether or not set comparisons are allowed in selection predicates or the way restructuring is expressed, have been attacked differently in complex object languages. Currently, the same is true for the object languages that have been proposed. This evolutionary aspect of query language design for object models is particularly stressed in [14, 16, 33, 53, 26, 120, 99]).

5 Conclusion

We gave an overview of research directions that have been, and continue to be, pursued in the field of extended relational databases. We have classified extensions to the relational model according to a programming language oriented view of data models. In this view, a data model is characterized by three components: (i) the collection of base (or primitive, atomic) types provided, (ii) the type constructors available for defining constructed types, and (iii) the operators of its query and update language. The purpose of this systematics is to show that with a good, that is, orthogonal, design of the data model, all operators are

tied to some type of its type system. Operators for base types are often ignored in database models (except for $=, <$ in selection predicates). However, *extensible* database management systems, by allowing the definition of new application-specific, user-defined base types, have to deal with such "attribute-level" operations. A typical example are geometric data types for spatial applications.

Our main interest has been on extensions to the collection of type constructors. The relational model has only one such constructor, relation (a combination of record and set). The restriction of the relational model to require that each relation be in First Normal Form does nor permit repeated use of the type constructor, since all attributes have to be atomic. Several directions for extensions are obvious: one can add other constructors (which might result in a model far beyond relations), split the constructor relation into its two parts, set and tuple (and use these orthogonally and repeatedly), or just stay with that one constructor, but allow repeated application. This last direction is the most stringent one, the resulting model is that of *nested relations*. Since no new types or type constructors are introduced, the query language can in principal remain unchanged, except the fact that it should now be applicable across multiple levels of nesting: whenever there is a "relation-valued" attribute, the language has to permit relational operators. This kind of orthogonality of a language, while being standard in programming languages, is new to database languages, since with flat relations there was no need for it.

A slightly more flexible class of data models is obtained when set and record are separated: *Complex Object* models. Here, it becomes clear that not only each of the base types has to accompanied by an appropriate set of operators, but also type constructors have to provide their generic operators for access and manipulation. Once this fact has been realized, it is clear how other type constructors can be added to a data model: they have to be defined together with their operators, such that all we have to require from the database language is orthogonality. The new constructor and its operators can be added rather easily. The increased flexibility of the type system allows for the direct representation of complex structures. These are crucial in any of the new application domains for database systems: engineering and design, regardless in which discipline such application tasks arise (electronic chips, software development, document preparation, natural sciences, and so on).

Finally, we have shown that research on nested relations and complex object models can also contribute to object-oriented data models, since there the same query language characteristics are needed. Also, many object models include orthogonal record and/or set constructors, such that object algebras are necessarily similar to nested relational ones. The benefits that can be expected are the use of optimization techniques, expressiveness and complexity results, and implementation strategies. Therefore, even though we concentrated on the data model and query language aspects of relational extensions, we also gave some remarks and hints on implementation work. Object-oriented databases offer even more flexibility for the design of new application systems, since the strict separation between the data management subsystem and the "real" application system is given up (the well-known impedance mismatch problem can be solved, or, at least, alleviated). In the form of "methods" or "foreign functions" (parts of) application algorithms can be imported into the (low-level) DBMS code, such that they execute 'closer to the data' and hence avoid massive data movement.

Many of the projects that have been mentioned in this survey have tried to evaluate their ideas and prototypes in a wide variety of applications. It is beyond the scope of this paper to report experiences gained there. In our own research in the DASDBS and COCOON projects we have been looking into geoscientific, office document filing, CIM, medical information systems, and architectural applications. Some of the experiences as well as future prospects are reported in [132, 129, 87, 133, 95, 100].

Acknowledgement. I owe thanks to Hans-Jörg Schek for encouragement and cooperation during the many years of our joint work on the subjects reported here. I am also grateful to all the past and present members of the DASDBS/COSMOS team. Thanks to Hans-Jörg Schek and the editor(s) for useful hints on improving a draft version.

References

[1] S. Abiteboul. Updates, a new frontier. In M. Gyssens, J. Paredaens, and D. van Gucht, editors, *ICDT '88: 2nd Int. Conf. on Database Theory*, pages 1–18, Bruges, Belgium, September 1988. LNCS 326, Springer Verlag, Heidelberg.

[2] S. Abiteboul. Towards a deductive object-oriented database language. In Kim et al. [65], pages 419–438.

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

[4] 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.

[5] S. Abiteboul, P. C. Fischer, and H.-J. Schek, editors. *Nested Relations and Complex Objects in Databases.* LNCS 361, Springer Verlag, Heidelberg, 1989.

[6] S. Abiteboul and Stéphane Grumbach. COL: A logic-based language for complex objects. Technical Report 714, INRIA, Paris, France, September 1987.

[7] 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.

[8] A.M. Alashqur, S.Y.W. Su, and H. Lam. OQL: A query language for manipulating object-oriented databases. In *Proc. Int. Conf. on Very Large Databases*, pages 433–442, Amsterdam, August 1989.

[9] 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.

[10] F. Bancilhon and S. Khoshafian. A calculus for complex objects. *Journal of Computer and System Sciences*, 38:326–340, 1989.

[11] 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.

[12] 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.

[13] L. Becker and R.H. Güting. Rule-based optimization and query processing in an extensible geometric database system. Computer Science Tech. Report 97, University of Hagen, September 1990. to appear in ACM TODS.

[14] 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*. LNCS 303, Springer Verlag, Heidelberg, March 1988.

[15] C. Beeri. Data models and languages for databases. In M. Gyssens, J. Paredaens, and D. van Gucht, editors, *ICDT '88: 2nd Int. Conf. on Database Theory*, pages 19–40, Bruges, Belgium, September 1988. LNCS 326, Springer Verlag, Heidelberg.

[16] C. Beeri. Formal models for object-oriented databases. In Kim et al. [65], pages 370–395. Revised version appeared in "Data & Knowledge Engineering", Vol. 5, North-Holland.

[17] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur. Sets and negation in a logic database language (LDL1). In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 21–37, San Diego, March 1987. ACM, New York.

[18] N. Bidoit. Efficient evaluation of queries using nested relations. Technical report, INRIA, Paris, 1985.

[19] N. Bidoit. The Verso algebra or how to answer queries with fewer joins. *Journal of Computer and System Sciences*, 35(3):321–364, December 1987.

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

[21] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented modelling with a rule-based programming paradigm. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 225–236, Atlantic City, NJ, May 1990. ACM, New York.

[22] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proc. Int. Conf. on Very Large Databases*, Kyoto, 1986.

[23] 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.

[24] S. Ceri, S. Crespi-Reghizzi, G. Lamperti, L. Lavazza, and R. Zicari. Algres: An advanced database system for complex applications. *IEEE Software*, 23, July 1990.

[25] W. Chen, M. Kifer, and D.S. Warren. HiLog as a platform for database programming languages. In Hull et al. [57], pages 315–329.

[26] S. Cluet, C. Delobel, C. Lécluse, and P. Richard. RELOOP, an algebra based query language for an object-oriented database system. In Kim et al. [65], pages 294–313.

[27] E. F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[28] L.S. Colby. A recursive algebra and query optimization for nested relations. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 273–283, Portland, OR, June 1989. ACM, New York.

[29] 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.

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

[31] S. Dar and R. Agrawal. Extending SQL with generalized transitive closure functionality. Tech. Memo. AT&T, 1990.

[32] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proc. Int. Conf. on Very Large Databases*, pages 197–208, Brighton, September 1987. Morgan Kaufmann, Los Altos, Ca.

[33] U. Dayal. Queries and views in an object-oriented data model. In Hull et al. [57], pages 80–102.

[34] U. Dayal, N. Goodman, and R. H. Katz. An extended relational algebra with control over duplicate elimination. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, Los Angeles, mar 1982. ACM, New York.

[35] 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. IFB 136, Springer Verlag, Heidelberg.

[36] U. Deppisch, J. Günauer, and G. Walch. Storage structures and addressing techniques for the complex objects of the NF^2 relational model. In A. Blaser and P. Pistor, editors, *Proc. GI Conf. on Database Systems for Office, Engineering, and Scientific Applications*, pages 441–459, Karlsruhe, 1985. IFB 94, Springer Verlag, Heidelberg. (in German).

[37] U. Deppisch, H.-B. Paul, and H.-J. Schek. A storage system for complex objects. In *Proc. Int. Workshop on Object-Oriented Database Systems*, pages 183–195, Pacific Grove, September 1986.

[38] A. Deshpande and D. van Gucht. An implementation of nested relations. In *Proc. Int. Conf. on Very Large Databases*, pages 76–87, Los Angeles, August 1988. Morgan Kaufmann, Los Altos, Ca.

[39] A. Deshpande and D. van Gucht. A storage structure for nested relational databases. In Abiteboul et al. [5], pages 69–83.

[40] V. Deshpande and P. Å. Larson. An algebra for nested relations. Research Report CS–87–65, University of Waterloo, Waterloo, Ontario, dec 1987.

[41] 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.

[42] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H.-D. Ehrich. Conceptual modelling of database applications using an extended er model. Technical Report 90–05, Dept of Computer Science, Techn. University of Braunschweig, Germany, December 1990.

[43] P. C. Fischer, L. V. Saxton, S. J. Thomas, and D. van Gucht. Interactions between dependencies and nested relational structures. *Journal of Computer and System Sciences*, 31(3):343–354, December 1985.

[44] 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.

[45] P. C. Fischer and D. van Gucht. Determining when a structure is a nested relation. In *Proc. Int. Conf. on Very Large Databases*, pages 171–180, Stockholm, 1985.

[46] J. C. Freytag. A rule-based view of query optimization. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 173–180, San Francisco, May 1987. ACM.

[47] 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.

[48] R.H. Güting, R. Zicari, and D.M. Choy. An algebra for structured office documents. *ACM Transactions on Office Information Systems*, 7(2):123–157, April 1989.

[49] M. Gyssens, J. Paredaens, and D. van Gucht. A graph-oriented object database model. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 417–424, Nashville, TN, April 1990. ACM.

[50] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M.J. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):143–160, March 1990. Special issue on Database Prototype Systems.

[51] 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.

[52] A. Heuer. A data model for complex objects based on a semantic data model. In Abiteboul et al. [5], pages 297–312.

[53] A. Heuer, J. Fuchs, and U. Wiebking. OSCAR: An object-oriented database system with a nested relational kernel. In *Proc. Int'l Conf. on Entity-Relationship Approach*, Lausanne, Switzerland, October 1990. North-Holland. to appear.

[54] A. Heuer and P. Sander. Semantics and evaluation of rules over complex objects. In Kim et al. [65], pages 439–458.

[55] G. Hulin. On restructuring nested relations in partitioned normal form. In *Proc. Int. Conf. on Very Large Databases*, pages 626–637, Brisbane, August 1990. Morgan Kaufmann, Los Altos, Ca.

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

[57] R. Hull, R. Morrison, and D. Stemple, editors. *2nd Int'l Workshop on Database Programming Languages*, Oregon Coast, June 1989. Morgan Kaufmann, San Mateo, Ca.

[58] G. Jaeschke. Recursive algebra for relations with relation valued attributes. Technical Report TR 85.03.002, IBM Heidelberg Scientific Centre, 1985.

[59] 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.

[60] B. Jiang. A suitable algorithm for computing partial transitive closures in databases. In *Proc. IEEE Conf. on Data Engineering*, Los Angeles, CA., 1990.

[61] U. Karge and M. Gogolla. Formal semantics of sql queries. Technical Report 90–01, Dept. of Computer Science, TU Braunschweig, Germany, 1990.

[62] 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.

[63] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical Report 3/1990, Dept. of Mathematics and Computer Science, University of Mannheim, Germany, June 1990.

[64] M. Kifer and J. Wu. A logic for object-oriented logic programming (Maier's O-Logic revisited). In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 379–393, Philadelphia, March 1989. ACM.

[65] W. Kim, J.-M. Nicolas, and S. Nishio, editors. *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases*, Kyoto, December 1989. North-Holland.

[66] C. Laasch and M.H. Scholl. Generic update operations keeping object-oriented databases consistent. Submitted for publication, July 1991.

[67] 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.

[68] B. Lindsay, J. McPherson, and H. Pirahesh. A data management extension architecture. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 220–226, San Francisco, May 1987. ACM, New York.

[69] V. Linnemann. Recursive functions in a database language for complex objects. *Information Systems*, 15(6):627–645, 1990.

[70] 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.

[71] D. Maier. A logic for objects. Technical Report CS/E-86-012, Orgeon CGraduate Center, 1989.

[72] A. Makinouchi. A consideration on normal form of not-necessarily-normalized relations in the relational data model. In *Proc. Int. Conf. on Very Large Databases*, Tokyo, 1977.

[73] M.V. Mannino, I.J. Choi, and D.S. Batory. The object-oriented functional data language. *IEEE Transactions on Software Engineering*, 16(11):1258–1272, November 1990.

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

[75] 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. IFB 136, Springer Verlag, Heidelberg. (in German).

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

[77] A. Ohori. Representing object identity in a pure functional language. In S. Abiteboul and P.C. Kanellakis, editors, *ICDT '90 – Proc. Int'l Conf. on Database Theory*, pages 41–55, Paris, France, December 1990. LNCS 470, Springer Verlag, Heidelberg.

[78] Oracle Corporation. *ORACLE User Manual*.

[79] J. A. Orenstein and F. A. Manola. Spatial data modeling and query processing in PROBE. Technical Report CCA–86–05, CCA, Cambridge, 1986.

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

[81] G. Ozsoyoglu, Z. M. Ozsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems*, 12(4):566–592, December 1987.

[82] Z. M. Ozsoyoglu and L. Y. Yuan. A normal form for nested relations. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 251–260, Portland, March 1985. ACM, New York.

[83] Z.M. Ozsoyoglu and L.Y. Yuan. On the normalization in nested relational databases. In Abiteboul et al. [5], pages 243–271.

[84] J. Paredaens and D. van Gucht. Possibilities and limitations of using flat operators in nested algebra expressions. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 29–38, Austin, March 1988. ACM, New York.

[85] C. Parent and S. Spaccapietra. An algebra for a generalized entity-relationship model. *IEEE Transactions on Software Engineering*, SE–11(7):634–643, 1985.

[86] 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.

[87] H.-B. Paul, A. Söder, H.-J. Schek, and G. Weikum. Supporting the Office Filing Service by a database kernel system. In H.-J. Schek, editor, *Proc. GI Conf. on Database Systems for Office, Engineering, and Scientific Applications*, pages 196–211, Darmstadt, 1987. IFB 136, Springer Verlag, Heidelberg. (in German).

[88] P. Pistor and F. Andersen. Designing a generalized NF^2 model with an SQL-type language interface. In *Proc. Int. Conf. on Very Large Databases*, pages 278–285, Kyoto, August 1986.

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

[90] Relational Technology Inc., Alameda, Ca. *INGRES Object Management Extension User Guide, Release 6.3*, November 1989.

[91] M. A. Roth and H. F. Korth. The design of ¬1NF relational databases into nested normal form. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 143–159, San Francisco, May 1987. ACM, New York.

[92] 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.

[93] 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.

[94] H.-J. Schek. Towards a basic relational NF^2 algebra processor. In *Proc. Int. Conf. on Foundations of Data Organization (FODO)*, pages 173–182, Kyoto, May 1985.

[95] 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 Database Prototype Systems.

[96] H.-J. Schek and P. Pistor. Data structures for an integrated database management and information retrieval system. In *Proc. Int. Conf. on Very Large Databases*, pages 197–207, Mexico, 1982.

[97] 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 Verlag, Heidelberg, 1983. (in German).

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

[99] H.-J. Schek and M.H. Scholl. Evolution of data models. In A. Blaser, editor, *Database Systems for the 90's, Proc. Int'l Symposium*, pages 135–153, Müggelsee, Berlin, November 1990. LNCS 466, Springer Verlag, Heidelberg.

[100] H.-J. Schek, M.H. Scholl, and G. Weikum. From the KERNEL to the COSMOS: The database research group at ETH Zürich. Technical Report 136, ETH Zürich, Dept. of Computer Science, July 1990.

[101] H.-J. Schek, M.H. Scholl, and G. Weikum. The background of the DASDBS & COSMOS projects. In *Proc. Int'l Conf. on Mathematical Foundations of Database Systems (MFDBS)*, Rostock, Germany, May 1991. LNCS, Springer Verlag, Heidelberg.

[102] J.W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(1):247–267, March 1977.

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

[104] J.W. Schmidt, H. Eckhardt, and F. Matthes. Extensions to DBPL: Towards a type-complete database programming language. ESPRIT report 892, University of Frankfurt, 1988.

[105] M. Scholl, S. Abiteboul, F. Bancilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, and A. Verroust. VERSO: A database machine based on nested relations. In Abiteboul et al. [5], pages 27–49.

[106] 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 Verlag, Heidelberg.

[107] 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).

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

[109] 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. MK.

[110] M.H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. Technical Report 150, ETH Zürich, Dept. of Computer Science, December 1990. To appear in Proc. DOOD Conf., Munich, Dec. 1991.

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

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

[113] M.H. Scholl and H.-J. Schek. Supporting views in object-oriented databases. *IEEE Database Engineering Bulletin*, 14(2):43–47, June 1991. Special Issue on Foundations of Object-Oriented Database Systems.

[114] H. Schöning. Integrating complex objects and recursion. In Kim et al. [65], pages 535–554.

[115] G.M. Shaw and S.B. Zdonik. Object-oriented queries: Equivalence and optimization. In Kim et al. [65], pages 264–278.

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

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

[118] 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.

[119] M. R. Stonebraker, B. Rubenstein, and A. Guttman. Application of abstract data types and abstract indices to CAD databases. In *Proc. IEEE Conf. on Engineering Design Applications, Database Week*, pages 107–113, San Jose, 1986.

[120] M.R. Stonebraker. The 3rd generation database system manifesto. In *Proc. IFIP TC2 Conf. on Object Oriented Databases – Analysis, Design & Construction (DS–4)*, Windermere, UK, July 1990.

[121] M.R. Stonebraker, E. Anderson, E. Hanson, and B. Rubinstein. QUEL as a data type. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 208–214, Boston, MA, June 1984. ACM, New York.

[122] 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.

[123] 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.

[124] D.D. Straube and M.T. Özsu. Queries and query processing in object-oriented databases. Technical Report TR 90–11, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada, April 1990. To appear in ACM TOIS.

[125] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.

[126] J. van den Bussche. Evaluation and optimization of complex object selections. Technical Report 91–17, University of Antwerp, Belgium, March 1991.

[127] D. van Gucht. On the expressive power of the extended relational algebra for the unnormalized relational model. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 302–312, San Diego, March 1987. ACM, New York.

[128] G. von Bültingsloewen. Translating and optimizing SQL queries having aggregates. In *Proc. Int. Conf. on Very Large Databases*, pages 235–243, Brighton, September 1987. Morgan Kaufmann, Los Altos, Ca.

[129] W. Waterfeld and H.-J. Schek. The DASDBS geokernel – an extensible database system for GIS. In K. Turner, editor, *Three-Dimensional Modeling with Geoscientific Information Systems*, Santa Barbara, Ca., 1990. Kluwer Academic Publishers. Proc. NATO Advanced research Workshop.

[130] 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.

[131] P. F. Wilms, P. M. Schwartz, H.-J. Schek, and L. M. Haas. Incorporating data types in an extensible database architecture. In C. Beeri and U. Dayal, editors, *Proc. Int. Conf. on Data and Knowledge Bases: Improving Usability and Responsiveness*, Jerusalem, June 1988. Morgan Kaufmann, Los Altos, Ca.

[132] A. Wolf. The DASDBS-Geo Kernel: Concepts, experiences, and the second step. In *Proc. Int. Symp. on the Design and Implementation of Large Spatial Databases*, Santa Barbara, Ca., 1989. LNCS 409, Springer Verlag, Heidelberg.

[133] P. Zabback, H.-B. Paul, and U. Deppisch. Office documents on a databse kernel – filing, retrieval, and archiving. In *Proc. ACM Conf. on Office Information Systems*, pages 261–270, Cambridge, Ma., April 1990.