The Two Roles of Nested Relations in the DASDBS Project

H.-J. Schek, Marc H. Scholl*

Abstract

The paper gives an overview of the nested relational model and its two roles in the **D**armstadt **D**atabase System (DASDBS) project, which was started in 1982 to develop an extensible database architecture supporting a variety of application specific front-ends with a common kernel system. In its first role the nested relational model serves as a model for the kernel interface describing hierarchical storage clusters. In its second role the nested relational algebra appears as a basic language for a KL-ONE-oriented semantic data model at an object-oriented layer upon the kernel.

1 DASDBS: A Project Beyond Relational Databases

In the beginning of the 1980s a significant direction in database research was established: various extensions to the relational model were proposed to enrich the rather poor semantics of the model [Cod79], and to open the model to better meet the requirements imposed by a new type of applications, the so-called "Non-Standard Database Applications".

Among the proposals for the enhancement of relational systems towards the new requirements are on-top solutions, like e.g. [HL82, LKM$^+$85], where explicit links between relations were introduced, and also several attempts at a more fundamental extension of the model. The latter approaches are characterized by the preservation of the relation as the basic data structure, while relaxing the first normal form condition. Traditionally, relations are required to obey 1NF which means that the values of all attributes of a tuple must be *atomic*, i.e. undecomposable by the DBMS. Already Makinouchi in [Mak77] observed that the assumption of 1NF is not a neccessary precondition for the relational theory to be applied. In the early 80s a number of groups came up with "unnormalized", "non-first-normal-form (NF^2)", or "not necessarily normalized" relations [FT83, BRS82, JS82, SP82]. In the sequel we will use the term *Nested Relations* for the characterization of the extended relational model, since it is both rather precise and neutral.

The motivation behind these proposals has been twofold: first, from the modelling point of view, simple tables of atomic components were considered inappropriate for the

*Authors' address: Dept. of Computer Science, Technical University of Darmstadt, Alexanderstr. 24, D-6100 Darmstadt, F.R.G.; E-mail:schek@ddadvsl.bitnet, mscholl@ddadvsl.bitnet representation of e.g. text [SP82]. Exploiting the capabilities of the concept "relation" by applying it repeatedly—"relations with relation-valued attributes" [SS86]—was the rationale behind this. Secondly, originating from the poor performance of the first relational systems, some efficient storage structures for relations were needed. For instance, VERSO relations were defined as a *model* in [AB84], while earlier they have been used in the database machine project as a storage structure [BRS82]. Also [FT83] considered unnormalized relations as promising internal structures for a conceptually flat relational database. Our own motivation for the development of the model comprised both aspects (cf. [SP82] on the modelling aspect and [SS83] on the other).

The Darmstadt Database System (DASDBS) project at the Technical University of Darmstadt was started in late 1982. It was expected that no single DBMS could be developed that covers all the different needs of the various new applications looking for DBMS support. Rather, a *kernel* that integrates the common features of a rather low-level storage component, but allows efficient and flexible *front-ends* tailored to specific application classes, was pursued. Thus the kernel with its several front-ends forms a *family* of database systems. The current members of the family are shown in figure 1. For details of the DASDBS project and system architecture the reader is refered to [DOP+85, SW86, PSS+87].

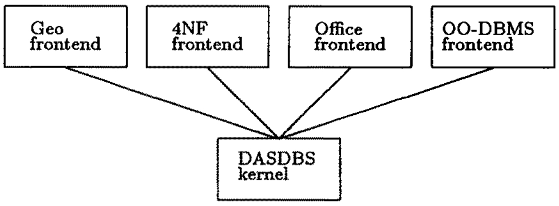


Figure 1: The DASDBS family of database systems

While the implementation of the DASDBS kernel [PSS+87, SPS87] clearly aimed at the efficiency issues related to more powerful storage structures, a novel approach to the data modeling aspect is covered in this paper as the second role of nested relations: it is shown how the step from flat relations with their simple but powerful query languages to nested relations can be carried over even one step further to general network structures. Nested algebras or SQL-type languages can be applied to non-hierarchical data too. We discuss this extension using KL-ONE as an example of a semantic net.

In this paper we first summarize the nested relational model as introduced in [SS83, SS86] in section 2. The first role of nested relations as a model for storage structures is only briefly described in section 3, while the new aspect of allowing recursive nested relations—and so supporting main constructs from semantic data modeling—is explained in some more detail in section 4. This is the second role of nested relations in DASDBS. A "user-friendly" version of nested SQL is also presented in section 4 and the idea of how to utilize this language for recursive queries is sketched.

2 The Nested Relational Model

2.1 Data Structure

Disregarding the first normal form condition for relations allows for a variety of extensions. We could have chosen to allow general structures known from programming languages as attribute values, for instance **records**, **arrays**, **lists**. While some proposals in fact allow such arbitrary domains (cf. [PA86, PT86]), the nested relational model in its pure sense is restricted to the constructor "relation", i.e. **set of tuples**. When describing the modelling power of the relational model by means of programming language constructs with a Pascal-like syntax, we could use **set of record** as shown in figure 2 where each of the *atomic-type*$_i$'s is a basic type like **integer**, **real**, or **string**.

```
type relation-type = set of record
                         attr-name$_1$ : atomic-type$_1$;
                                     ⋮
                         attr-name$_n$ : atomic-type$_n$
                         end;
```

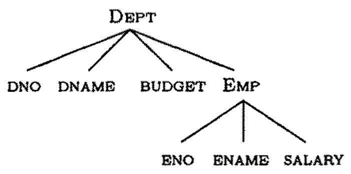
Figure 2: Flat relation as a Pascal-like type

The idea of the nested relational model is simply to allow relations at any place where attributes occur. Hence, attribute values may either be atomic or relations. This (hierarchical) nesting of relations may be repeated for an arbitrary (but fixed, see below) number of levels. Again applying the Pascal-like syntax, an example is given in figure 3 where the nesting in *attr-name*$_j$ may continue to an arbitrarily deep level.

```
type nested-relation-type = set of record
                                        ⋮
                            attr-name$_i$ : atomic-type$_i$;
                                        ⋮
                            attr-name$_j$ : set of record
                                               ⋮
                                              end;
                                        ⋮
                            end;
```

Figure 3: Nested relation as a Pascal-like type

From a semantic modelling point of view (see e.g. [LS87]), the construct "relation" corresponds to the application of *one aggregation* (tuple constructor) followed by *one association* (set constructor) to a collection of primitives (the underlying atomic domains). Thus nested relations can be constructed by repeatedly applying the sequence *aggregation–association* to a collection of primitives or composite objects obtained by previous applications of this rule.



DEPT					
			EMP		
DNO	DNAME	BUDGET	ENO	ENAME	SALARY
		⋮		⋮	
20	Computer Science	≈ 0		⋮	
			324193	M. Scholl	little
				⋮	
		⋮		⋮	

Figure 4: Sample nested relation: schema tree and value table

Since formal definitions of nested relations can be found in [SS86], we will only summarize our notations in this paper. The key issue in our formalization is a two-part view of a relation: *schema* and *value*. The former is a description of the structure, that gives the *names of the attributes* and comprises the assignment of *domains* to these attributes—also called "intension" of the relation—, while the latter is the *set of tuples* belonging to the "extension" of the relation at a certain point in time. Usually schemata are given by a parenthesized list of attribute names preceeded by the relation name. For instance,

$$\text{EMPLOYEE}(\text{ENO}, \text{ENAME}, \text{SALARY}, \text{DNO})$$

is the schema of the famous employee relation having four attributes, viz. employee number (the *key*), employee name, salary, and department number. In most cases an explicit assignment of domains to the attributes is omitted. Values are shown as tables having attribute names as column headings and tuples one in a row.

When defining *nested* relations where attributes may be relation valued, obviously the 'flat' definitions become recursive. A schema now is a set of (attribute) *descriptions* each of which consists of an attribute *name* and an attribute schema. Iff an attribute schema is empty (as a set), then this attribute is an atomic one, otherwise the schema describes the relations contained in the attribute values. Accordingly, the *domain* of an attribute is either a set of primitives (for an atomic attribute) or the powerset of the Cartesian product of the domains of the subattributes. Hence, every valid value of a non-atomic attribute is a set of tuples over the corresponding domains, which reflects the idea of "nested" relations. Schemata of nested relations can be represented by trees and values by nested tables. A linear notation of the schema in figure 4 is

$$\text{DEPT}(\text{DNO}, \text{DNAME}, \text{BUDGET}, \text{EMP}(\text{ENO}, \text{ENAME}, \text{SALARY}))$$

As can be seen in the example, the usual notion of a key can easily be applied in the nested case. Unlike other authors [AB84, FT83, RKS84], we do not require the existence of atomic keys, or even atomic attributes on each level of a nested relation, nor the disjointness of subrelations. Most of these criteria seem to be suitable for "good" nested relational databases, but they can be expressed as additional constraints, leading into the area of normal forms for nested relations [OY85, RK87].

2.2 Operations

In the design of a language for the manipulation of nested relations we followed a simple idea: whenever a relation is encountered at some place in the language, we wanted to allow the application of queries expressed in that language. As our focus was not on user interfaces but rather conceptual and implementation-oriented, we decided not to look into 'high level languages', like a nested relational SQL [PA86, PT86, RKB87]. The relational algebra provides a flexible formal language that is better suited for optimization issues, for instance. Thus, we extended the algebra (see, e.g. [Mai83, Ull82]) to cope with nested relations.

The Flat Relational Algebra

The power of the algebra is obtained by five basic operators: union($\cup$), difference($\setminus$), relational product($\times$), projection(π), selection(σ), and the ability to *compose* expressions from these operators. Every operator can be regarded as a (generic) function mapping one (or two) relation(s) into another relation. Hence, complex queries can be formulated by functional composition of algebraic operators. The following recursive rules define the relational algebra:

1. If R is a relation (name), then it is an algebraic expression.
2. If E_1, E_2 are algebraic expressions, then so are $E_1 \cup E_2$, $E_1 \setminus E_2$, and $E_1 \times E_2$.[1]
3. If E is an algebraic expression, then so are $\pi[L](E)$ and $\sigma[F](E)$.
   L, the *projection-list*, is a list of attribute names contained in the schema of E, and F, the *selection-formula*, is a formula made up from logical connections ($\wedge$,$\vee$,$\neg$) of (arithmetic) comparisons between attributes in the schema of E and constants.

Nest and Unnest

The first two operations that came along with the first publications on nested relations deal with the transformation between flat and nested relations and vice versa: Nest (ν) and Unnest(μ) [FT83, JS82]. Nesting achieves an effect similar to what is intended by the SQL "GROUP BY"-clause, except that the SQL-clause goes beyond the relational model, while nest, of course, stays within the (nested relational) model. The effect of nesting can be undone by unnesting. While unnest is always inverse to nest, the opposite is not true in general [JS82, FT83]. For the schema part, an example of nest and unnest is given below:

$$\begin{aligned}
&\textsc{NestedDept}(\textsc{dno}, \textsc{dname}, \textsc{budget}, \textsc{Emp}(\textsc{eno}, \textsc{ename}, \textsc{salary})) := \\
&\quad \nu[(\textsc{eno}, \textsc{ename}, \textsc{salary}) = \textsc{Emp}] \\
&\qquad\qquad (\textsc{FlatDept}(\textsc{eno}, \textsc{ename}, \textsc{salary}, \textsc{dno}, \textsc{dname}, \textsc{budget}))
\end{aligned}$$

$$\begin{aligned}
&\textsc{FlatDept}(\textsc{eno}, \textsc{ename}, \textsc{salary}, \textsc{dno}, \textsc{dname}, \textsc{budget}) := \\
&\quad \mu[\textsc{Emp}](\textsc{NestedDept}(\textsc{dno}, \textsc{dname}, \textsc{budget}, \textsc{Emp}(\textsc{eno}, \textsc{ename}, \textsc{salary})))
\end{aligned}$$

[1]for the set operations ($\cup$ and $\setminus$) to apply, E_1 and E_2 must be "union compatible", i.e. have identical schemata—at least up to renaming.

Notice that nesting, besides decomposition (projection), is a way of obtaining "better" relational schemata during database design, because it also removes redundancy. This is the reason why nested relations can be considered appropriate in the design of a database schema in some "normal form", see e.g. [OY85, RK87].

The Nested Relational Algebra

For the data structure of nested relations we allowed nesting relations into relations in place of attributes. Similarly, we can nest algebraic expressions into algebraic expressions in place of attributes. This way, we obtain a doubly nested algebra: the first way of nesting is the usual one, viz. functional composition (see above), the second one is new. At any place where attributes occur in the flat algebra, i.e. in the projection-list and the selection-formula, we can now use algebraic expressions. For instance, if in the result ND we want to see only the department names and employee names from the above nested DEPT relation, we apply a projection to DEPT (we only want DNAME and something from EMP) and inside this projection another projection on EMP retaining only ENAME, yielding NE:

$$\mathrm{ND} := \pi[\mathrm{DNAME}, \mathrm{NE} := \pi[\mathrm{ENAME}](\mathrm{EMP})](\mathrm{DEPT}).$$

The schema of the resulting relation is ND(DNAME, NE(ENAME)).

To obtain a characterization of our nested algebra from the flat algebra described above, we only have to modify two points:

- L, the projection list:
  may now contain *algebraic expressions*, $N_i := E_i$, instead of just attribute names. A subrelation N_i of the result of the projection is defined to take as value the result of evaluating E_i.

- F, the selection formula:
  may now involve set comparison predicates Θ and algebraic expressions $E_i \Theta E_j$.

Examples of such nested algebra expressions are the following queries:

1. Give all departments, that have 'Computer Science' as their name and at least one employee making more than 30,000.-:

$$\sigma[\mathrm{DNAME} = \text{“Comp. Sc.”} \wedge \sigma[\mathrm{SALARY} > 30000](\mathrm{EMP}) \neq \emptyset](\mathrm{DEPT}).$$

2. Give all departments with name "Computer Science", and from these departments all employees making more than 30,000.-, called RE ("rich employees"):

$$\pi[\mathrm{DNO}, \mathrm{DNAME}, \mathrm{BUDGET}, \mathrm{RE} := \sigma[\mathrm{SALARY} > 30000](\mathrm{EMP})]$$
$$(\sigma[\mathrm{DNAME} = \text{“Comp. Sc.”}](\mathrm{DEPT})$$

3. Similar to the previous one, but now ("CS"-) department tuples should be discarded, if there is no employee making more than 30,000.-:

$$\sigma[\mathrm{RE} \neq \emptyset](\pi[\mathrm{DNO}, \mathrm{DNAME}, \mathrm{BUDGET}, \mathrm{RE} := \sigma[\mathrm{SALARY} > 30000](\mathrm{EMP})]$$
$$(\sigma[\mathrm{DNAME} = \text{“Comp. Sc.”}](\mathrm{DEPT}))$$

Notice, that the additional selection ("$\sigma[\text{RE} \neq \emptyset]$") refers to the *result of the inner selection.*

It is interesting to notice the correspondence between the third nested relational query and the flat relational select-project-join query

$$\sigma[\text{DNAME} = \text{"Comp. Sc."}](\text{DEPT}) \bowtie \sigma[\text{SALARY} > 30000](\text{EMP})$$

on the equivalent flat relational schemata. There the departments without employees making more than 30,000.- are discarded automatically by the join. The equivalent flat expression for the second nested query can be obtained by using an *outer join* instead of the natural join.

The above examples should illustrate that in typical nested relational algebra queries, in general, arbitrary expressions (not only selections and projections) can be nested inside a projection (or selection). The crucial point is to define what the valid arguments to such "inner" operations are. To understand the idea of our solution, we start with the usual algebraic expressions: such an expression is a (composite) *function applied to the database*. Thus, valid arguments to the function are the objects in the database, viz. the relations. The relations known within the database form the *scope* of the expressions on the outermost level. Now consider an expression applied within a projection list: the outer relations of the database are still within the scope, but in addition, the attributes of the expression to which we apply the projection are added to the scope. Hence, *outer relations and the attributes* may become arguments to the inner expression. In general, descending one level in the nesting hierarchy of a relation enlarges the scope of expressions by the attributes of this new level.

The phenomenon of using higher level (in the schema tree) attributes within deeply nested algebra expressions has been called "Dynamic Constants" in [SS86]. This concept has proven very powerful, since nesting, difference, and relational product (and thus join) can be defined using only nested selection and projection with dynamic constants [Sch87, Sch88]. Consider a very simple example: selecting from the subrelation employees for each department tuple the manager's subtuple (assuming an attribute MNO of DEPT):

$$\pi[\text{DNO}, \text{DNAME}, \sigma[\text{ENO} = \text{MNO}](\text{EMP})](\text{DEPT}).$$

Here MNO is a "dynamic constant" in the scope of the inner selection applied to EMP.

We have also defined *update* operations for nested relations in a notation similar to the query algebra, see [Sch85b, Sch88]. They are also nested. We can, for instance, *change* a tuple by *inserting* a subtuple into a subrelation, e.g. change a department by hiring a new employee.

Thus the algebra provides a powerful set of operations on the hierarchical structure of nested relations. In order to manipulate subrelations deep down in the hierarchy of a nested relation, the corresponding algebraic expression is nested inside projections and/or selections. In particular, this algebra avoids having to unnest a relation, apply flat algebraic operations and renesting in such cases. Therefore it is more than a theoretical tool, it may serve as the operational interface of an efficient DBMS kernel providing nested relational storage structures.

3 Static Nested Relations as Storage Structures in the DASDBS Kernel

3.1 Hierarchical Clustering in the DASDBS Kernel

The kernel has two main layers: the Stable Memory Manager (SMM) is a set-of-pages oriented layer that includes buffering and supports classical transaction management for page-level operations. The next layer supports hierarchically structured "Complex Records" as primitive Complex Objects. Complex records are the implementation of nested tuples according to the nested relational model. A Complex Record is stored in as few pages as possible. The set of pages occupied by one Complex Record is called "Storage Cluster". Sets of Complex Records can be retrieved, inserted, updated, or deleted, thus set-orientation is achieved on the record and the page level. All kinds of higher level objects, including geo-objects, flat relations, or frames-like objects are mapped to these kernel objects by the corresponding front-ends. The kernel structures are used by the front-ends as storage clusters, i.e. selecting a data representation on the kernel level corresponds to physical database design.

The following two propositions motivated our choice of the nested relational model for the description of such internal database layouts:

1. one can do better than just storing tuples of a relation one by one as records into a file; i.e. flat relations as storage structures are too poor,

2. external storage devices provide a (virtual) block-structured linear address space. Hierarchies are the most general structures that can be linearized (without introducing redundancy or using "pointers"), i.e. nested relations are general enough.

Of course, one can apply sophisticated storage schemes for flat relations too, our point is just that then these structures cannot be described by that model. Rather the physical schema is usually determined by a collection of parameters on a different (lower) level of abstraction. With nested relations on the other hand, all relevant physical design techniques can be described by means of the model. Denormalization [SS80, SS81] by joins, nested denormalizations by a nested relations (join plus nest), pointers by projections of "address attributes" that can be used as a reference mechanism in other relations (access paths or tuples from joining relations, "link fields"). A detailed discussion of the various alternatives is contained in [SPS87], [Sch88] investigates their description by our nested relational algebra, and [Pau88] analyzes the cost of the variants in the context of the DASDBS imlementation. Figure 5 gives a summary of alternatives for the internal representation of many-to-many relationships. Obviously, those cannot be implemented symmetrically by nested relations without introducing redundancy. Thus either we use only references (pointers, addresses) to support these relationships or we decide to maintain (even more, viz. in the actual data) redundant storage structures. The choice, however, has to be guided by the workload that is to be optimized, i.e. mainly retrieval-to-update ratios in this example.

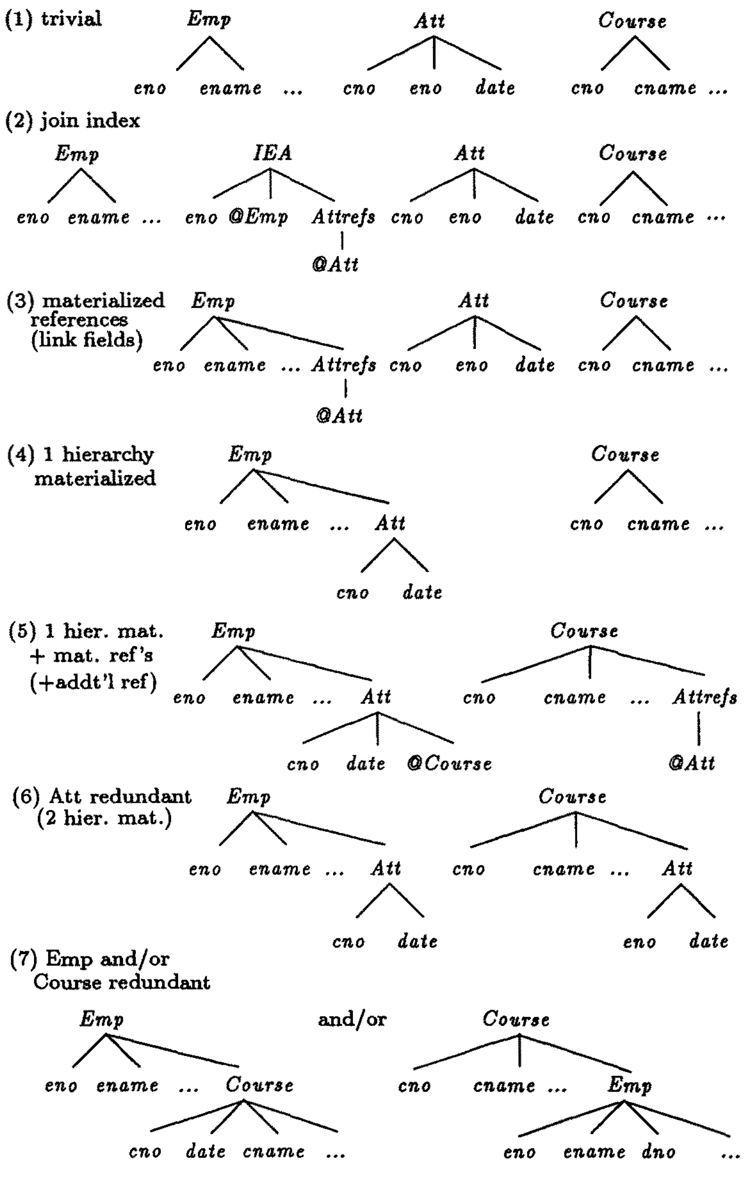


Figure 5: Some alternative storage schemes for $n : m$-relationships

3.2 Kernel Interface: Single Scan Operations on Nested Relations

An important problem in the implementation of DASDBS was the decision, how much of the complex query facilities of the nested algebra to implement within the kernel. A coarse characterization is: allow as powerful operations as possible, but keep them *linear*. Linear means that a kernel query must be evaluable in a *single hierarchical scan* over the data. This criterion obviously excludes joins, for instance. However, in our case of nested expressions single scan processible is harder to define than just requiring "single table" queries like in the RSS component of the System R prototype [ABC+76]: on every hierachical nesting level of a single scan query there may only occur projections and selections. The selections must not include other set comparisons than "$\langle something \rangle \langle cop \rangle \emptyset$", with $\langle cop \rangle$ "$=$" or "$\neq$". Other set comparisons, e.g. equality of two sets (of subtuples), would introduce join complexity. The notion of single scan operations is discussed in [Sch85a, Sch86, PSS+87] and defined formally in [Sch88]. Interestingly, a similar notion and characterization has been found in the VERSO database machine project [BRS82, A+86].

3.3 Algebraic Optimization in the Flat Relational Front-End

One of the applications for a nested relational kernel system that has guided our project from the beginning and that also had a significant influence on the design of the algebra is a flat relational front-end. The underlying idea is performance-oriented: by using nested relations as *internal* representation for a conceptually flat relational (e.g. 4NF) database, we can precompute some of the most frequently issued join operations and store the result of the join without introducing redundancy. This idea is based on a proposal by Schkolnick [SS80, SS81] to "denormalize" or "materialize" joins in order to save query processing time. Of course, the differing conceptual (flat) and internal (nested) database layout must be hidden from the user. This can be achieved by a transformation step controlled by the DBMS.

The key issue of the flat relational front-end is algebraic optimization: as both levels are described (nested) relationally, the transformation can be defined via the nested algebra, and substituting the corresponding definitions into conceptual-level queries is a trivial way of transforming the queries to the internal level. However, to actually *remove* the redundant (join) operations from the user query, an algebraic optimization step is necessary. This problem has been solved in its theoretical aspects [Sch86], the main result is that select-project-join queries on the conceptual level, that do not require any join internally—as all joins are materialized—can efficiently be optimized and are mapped to single scan queries at the internal level. (This is another justification for the appropriateness of this subset of algebraic expressions for the kernel.) Again a similarity to the VERSO project: [Bid85] characterizes VERSO "superselections"—a kind of queries that corresponds to a combination of our nested projection and selection—by flat relational select-project-join queries.

4 Recursive Nested Relations in the DASDBS Data Model

4.1 Motivation and Example

Almost immediately after the first investigations on nested relations have been published in the years 1982 and 1983, controversial discussions started. Among the objections that were raised it was claimed that the does not support shared subobjects or $n : m$-relationships or recursive data structures.

In fact, it is true that types of hierarchically organized data are obtained if a tuple constructor and a set constructor repeatedly are allowed for the construction of complex object types. The real question, however, is what kind of operations we allow on these types. As it could be seen from the previous section we extended the high-level set-oriented operations on relations to any set-of-tuple type which is obtained by the type constructors. This is the important difference, compared to the old world of hierarchical data models.

With regard to the second concern let us consider the question of n:m relationships and recursive data. In fact, if we elevate the static view we had for the storage structure also for the data modelling layer, we are unable to model symmetrically n:m relationships, apart from introducing redundancy or using references for symmetry.

Fortunately there is no reason why we should keep a static nested relational schema definition for this layer. In fact, in the DASDBS object layer the starting point is recursive definitions of nested relational schemata with net-structured instances. This is the key to deal with n:m relationships and shared objects in a symmetric way. Static nested relations then appear quite naturally as result of operations on such nets. Such structures are called "dynamically nested relations" in [LS88]. In the following we will explain this role of nested relations by an example. Take the following type definitions in a hypothetical Pascal++ language

```
type project =  record
                   pno : integer;
                   pn : name;
                   members : set of employee;
                   produces : set of part;
                end;
type employee = record
                   eno : integer;
                   ename : name;
                   assignmts : set of projects inverse members;
                   education : set of course;
                   manages : set of employee;
                end;
type part =     record
                   pname : name;
                   produced_by : set of project inverse produces;
                end;
type course =   record
                   cname : name;
                   ⋮
                end;
```

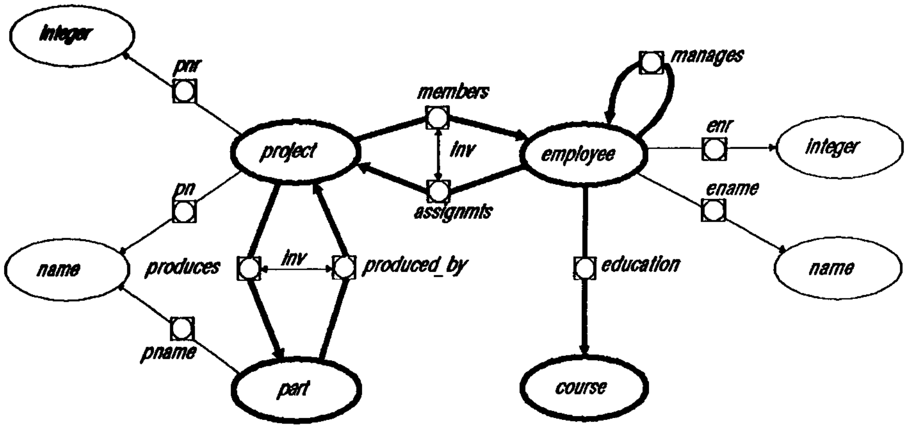


Figure 6: Example of a KL-ONE semantic network

type *projects* = **set of** *project*;
type *employees* = **set of** *employee*;
type *parts* = **set of** *part*;

Apparently this is not standard Pascal because—as mentioned—set of "structured" type is not allowed (the first +). More importantly, recursively defined types occur (the second +). Recursion not only occurs because of the manages component in employee which is defined by the employee type again in the manages component, but, more generally, due to the description of one object by means of others and vice versa. For example, project is described by its members which are employees and employees are described by their project assignments and it is required that they are inverse to the members component.

4.2 KL-ONE

Those who are familiar with Brachman's KL-ONE semantic network for knowledge representation [BS85] will have recognized that the view we have taken is the same as in KL-ONE. In fact the previous type definitions in Pascal++ are not hypothetical. They directly correspond to the definition of KL-ONE "generic concepts". Project in this terminology is a generic concept with "roles" *pno, pn, members*, and *produces*. The roles *pno* and *pn* have to be filled with at most one value from some primitive concepts while *members* is filled with a set of values from a generic concept *employee* and *produces* is filled with a set of *parts*. So everything in our previous example can be expressed in KL-ONE's terminology definition (called T-Box there). Roles and their inverses model properly the symmetry of (n:m) relationships. A corresponding graphical representation is shown in figure 6.

4.3 Examples of Nested Relational Views on the KL-ONE Net

The symmetric view taken in KL-ONE, i.e. the recursive type definitions, are the reason why we are not able to assign a static nested relational schema equivalent to the type definitions. Nevertheless we see that, basically, tuples (projects, employees, parts) appear which have relation valued attributes (components). The interesting fact is that many different nested relational schemata are contained in the previous definitions. We could

(1) *projects (pno, pn, members (eno, ename))*

or, symmetrically we may see

(2) *employees (eno, ename, assignmts (pno, pname))*

Either projects are the top-level relation with their employees as a sub-relation, or, vice-versa, we see employees at the root of the schema tree and their projects as descendants. More interestingly, we could also take a cyclic view, like

```
(3) parts (pname,
          produced-by (pno,
                       members (eno),
                       produces (pname))),
```

which gives a nested relation where for each part we find by which projects it is produced and for each such project we see the employees (by their number *eno*, in *members*) in this project and all parts which are produced there (by their name *pname* in *produces*).

In order to look into the *manages* hierarchy, we could use

(4) *employees(eno, manages (eno))*

The principle we applied in all these examples is straightforward: We decided which object type should be the root and selected a subset of its components. For each selected component which is a set type we again selected which components of the children we wanted to see and so on. Thus, after having defined recursive nested relations with our Pascal++ language, we are now looking for a (view definition) language to dynamically define nested relational views on this network structure.

4.4 Nested Algebra or Nested SQL for KL-ONE

The exciting fact now is that such a language already exists: we can apply the nested relational algebra [SS86] or nested SQL [PA86, RKB87] without any change. We are able to express all previous examples by regular nested algebra expressions or by nested SQL in a very simple way as shown by the following examples

```
(S1) select pno, pn,                                (A1) π[pno, pn,
         (select eno, ename from members)                  π[eno, ename](members)](projects)
     from projects
```

We omitted renaming here, i.e. we called the resulting objects "*projects*" with their attributes *pno, pn*, and *members*. The symmetric case, also without renaming, is

```
(S2) select eno, ename,                             (A2) π[eno, ename,
         (select pno, pn from assignmts)                   π[pno, pn](assignmts)](employees)
     from employees
```

Notice the wellknown role of the select clause (or the π in the algebra) as a type constructor. As in usual SQL the result of the select statement is a set of tuples with components as designated by the select list. The following examples are expressed as easily as the previous ones

```
(S3) select pname,                              (A3) π[pname,
        (select pno,                                    π[pno,
            (select eno from members),                      π[eno](members),
            (select pname from produces),                   π[pname](produces)]
        from produced_by)                               (produced_by)]
     from parts                                      (parts)
```

Also a recursive type is simply handled

```
(S4) select eno, (select eno from manages)   (A4) π[eno, π[eno](manages)](employees)
     from employees
```

The previous examples have shown how the schema of a nested relation is defined dynamically by a query. At the same time the instances are defined. Up to now we have applied projection which selects a subset of attribute values in each tuple as usually. But we may apply selection too. We may restrict the instances by some predicates as usually. If, for example we were interested in all projects but only the member with name 'Smith' we would write

```
(S1') select pno, pn,
            (select eno from members
            where ename='Smith')
      from projects
```

Note that an empty set would be returned if a project has no 'Smith' among its members. In order to see people reporting directly to 'John' we would write

```
(S4') select eno,
            (select eno from manages)
      from employees
      where ename='John'
```

While this already looks quite elegant there are two concerns which we will discuss in the sequel. The first one is ease-of-use, the second is the subject of recursive queries.

4.5 Sequences of Flat SQL Statements

Fortunately there is a proposal which may be an answer to both concerns at the same time. The idea is to apply simple, flat SQL-like statements and to connect them via names [LS88]. Complex queries get confusing using nested expressions and indentation as shown above. Splitting them into smaller pieces defined in several steps gives more evidence as we see by rewriting some of the previous examples.

```
(F1) begin select
            select pno, pn, M from projects
            M: select eno, ename from members
     end select

(F3) begin select
            select pname, P from parts
            P: select pno, X, Y from produced_by
            X: select eno from members
            Y: select pname from produces
     end select
```

We introduced names in the select clauses as placeholders and define them later step by step. In the second example we know that we want to see *pname* together with something we abbreviate P for every *part*. P is defined later as a set of tuples calculated from *produced-by* which is a set-valued component of *part*, the element type of *parts*. Within P, in turn, we select *pno* and two things called X and Y defined later for every element in *produced-by* which as we know is of type *project*. The value *pno* is a component of a basic type in *project* whereas X or Y are computed from the set-valued components *members* or *produces* of *project* respectively. The simple rule is that we specify by select what we want to see if a set-valued component (object or subobject) is encountered. In other words, a "**select** *" as a default for selecting everything must be applied more carefully.

This "trick" with names allows us to handle the recursively defined types in a way as simple as the nonrecursive ones. The manages example is one:

```
(F4') begin select
            select eno, Down from employees
                where ename='John'
            Down: select eno from manages
      end select
```

In a similar way we could get people reporting directly to John and those at the next lower level under John

```
(F4") begin select
            select eno, Down1 from employees where ename='John'
            Down1: select eno, Down2 from manages
            Down2: select eno from manages
      end select
```

Also aggregate functions can be used easily as in

```
begin select
    select pname, P from parts
    P: select pno, C from produced_by
    C: select count(*) from members
end select
```

which gives us for every part the *pname* and the set of tuples consisting of the project number *pno* and the number *C* of employees working in this project for all projects producing the current part.

4.6 Recursive Queries

While the previous examples still can be handled with standard operations of the nested relational model we have to extend the model for real recursive queries, e.g. if we want to see the complete hierarchy under John. The difference is that in all previous examples we can determine the type of the result, i.e. the schema of the resulting nested relation by parsing the select statements without execution. In other words, given the recursive type definition in Pascal++ and given the query specification, the type of the result can be determined at compile time just by inspecting the select-lists. For example in F4" the type is

Q(eno, down1(eno, down2(eno)))

This is no longer possible for John's complete hierarchy as we don't know in advance how many levels will be obtained by this computation. However, it is simple to express these kinds of queries. The utilization of names within SQL expressions becomes crucial now and is not only "syntactic sugar". For instance, consider the employee hierarchy below 'John':

```
(F4*) begin select
            R: select eno, Down from employees where ename='John';
            Down: select eno, Down from manages
      end select
```

As in the previous examples the result R is the set of "John" tuples consisting of *eno* and *Down*, defined in the next statement. This defines *Down* recursively: the result is of a recursive type

```
type R = set of record
                   eno: integer;
                   Down: R
                 end
```

equivalent to the following recursive 'nested relation'

R(eno, Down(R)).

Its instances may have an arbitrary depth. As soon as the manages component is empty, the result is empty, the recursion terminates, and an empty set is returned to the next higher layer indicating that we arrived at an employee who is not manager.

5 Summary

A brief overview on the nested relational model, its basic concepts, the operations, and the implementation and utilization within the Darmstadt Database System (DASDBS) project was given. The first role of nested relations is the formal description of hierarchical clustering techniques. Here a nested relational schema is the result of a physical database design process which, for instance, utilizes this structure for materializing an important (i.e. frequent) join operation. The second, and more innovative role of the nested relational model is to give up static schema definitions but rather to allow recursive ones. We showed how a nested algebra or SQL-type language can be used on such a general network shaped structure without any change. The idea basically consists in dynamically taking hierarchical views of the net with the "current" node as the root.

It should be mentioned that while this easily applies for retrieval operations on network structures, some more care has to be taken for the update operations. Variations of nested relational update operations have to reflect the fact that subobjects may now be shared between different hierarchical views. Therefore one has to distinguish, for instance, whether or not a change of such a shared subobject should be propagated to the other hierarchical views too. One of the new projects in the context of the DASDBS family will study this new direction of migrating the nested relational paradigm to this more general data model. This will also include extensibility features like externally defined basic types and attached procedures.

References

[A$^+$86] S. Abiteboul et al. VERSO, a DBMS based on non-1NF relations. Technical Report 253, INRIA, Paris, 1986.

[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.

[ABC$^+$76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffith, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[Bid85] N. Bidoit. Efficient evaluation of queries using nested relations. Technical report, INRIA, Paris, 1985.

[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.

[Cod79] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, December 1979.

[DOP$^+$85] U. Deppisch, V. Obermeit, H.-B. Paul, H.-J. Schek, M. H. Scholl, and G. Weikum. The storage component of a database kernel system. Technical ReportDVSI–1985–T1, Technical University of Darmstadt, 1985.

[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.

[HL82] R. Haskin and R. Lorie. On extending the functions of a relational database system. In *Proc. ACM SIGMOD Conf. on Management of Data*, Orlando, 1982. ACM, New York.

[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.

[LKM$^+$85] R. Lorie, W. Kim, D. McNabb, W. Plouffe, and A. Meier. Supporting complex objects in a relational system for engineering databases. In W. Kim, D. S. Reiner, and D. S. Batory, editors, *Query Processing in Database Systems*. Springer, 1985.

[LS87] G. Lausen and H.-J. Schek. Semantic specification of complex objects. In *Proc. IEEE CS Symp. on Office Automation*, Gaitherburg, 1987.

[LS88] R. Lorie and H.-J. Schek. On dynamically defined complex objects and SQL. In *Proc. 2nd Int. Workshop on Object-Oriented Database Systems*, Bad Münster, September 1988. (to appear).

[Mai83] D. Maier. *The Theory of Relational Databases.* Pitman Publishing Ltd., London, 1983.

[Mak77] 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.

[OY85] 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.

[PA86] 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.

[Pau88] H.-B. Paul. *DAS Database Kernel System for Standard and Non-standard Applications —Architecture, Implementation, Applications—*. PhD thesis, Dept. of Computer Science, Technical University of Darmstadt, 1988. (in German) in preparation.

[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.

[RK87] 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.

[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.

[RKS84] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for ¬1NF relational databases. Technical Report TR-84-36, University of Texas at Austin, Austin, TX, 1984. (revised version, January 1986).

[Sch85a] 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.

[Sch85b] M. H. Scholl. The NF^2 relational model for internal data structures. Technical NoteDVSI–1985–A8, Technical University of Darmstadt, 1985.

[Sch86] M. H. Scholl. Theoretical foundation of algebraic optimization utilizing unnormalized relations. In *ICDT '86: Int. Conf. on Database Theory, Rome*, pages 380–396. LNCS 243, Springer, Berlin, Heidelberg, 1986.

[Sch87] 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. Proceedings to be published with full papers).

[Sch88] 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).

[SP82] 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.

[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.

[SS80] M. Schkolnik and P. Sorenson. Denormalization: A performance oriented database design technique. In *Proc. AICA Conf.*, Bologna, Italy, 1980.

[SS81] M. Schkolnik and P. Sorenson. The effects of denormalization on database performance. Research Report RJ3082 (38128), IBM Research Laboratory, San Jose, CA, 1981.

[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.

[SW86] H.-J. Schek and G. Weikum. DASDBS: Concepts and architecture of a database system for advanced applications. Technical ReportDVSI-1986-T1, Technical University of Darmstadt, 1986. German Version to appear in: Informatik Forschung und Entwicklung, 1987.

[Ull82] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, MD, 2nd edition, 1982.