Supporting Flat Relations by a Nested Relational Kernel

M. H. Scholl, H.-B. Paul, H.-J. Schek
Technical University of Darmstadt, Computer Science Department
Alexanderstraße 24, D–6100 Darmstadt, West Germany

Abstract

While a variety of sophisticated physical database design techniques has been devised in research, only very limited capabilities are available in practice. This is mostly due to the fact, that designing internal layouts differing from the logical view is not transparent to the user. We present a more general two-level solution: Our database kernel offers hierarchical clustering expressed as nested (NF^2) relations. Flat relations, resulting from logical database design, are then mapped to this internal kernel interface. We show, how the various physical structuring approaches can be expressed in this model. Physical database design for a flat relational front-end can then be described formally within the (NF^2) relational model. The important aspect of join support is pursued by materializing some joins in nested relations without any redundancy. Select-project-join queries on the logical schema can be transformed to efficiently processible internal queries by applying algebraic optimization techniques, known e.g. from view optimization. Preliminary performance evaluations are reported that were carried out on commercially available systems and solicited our expectations.

1 Introduction & Problem Statement

Currently relational database management systems (RDBMS) have reached wide acceptance in commercial applications. One of the reasons certainly is that the *logical* design of a relational database is mostly understood and often supported by additional software tools. However, the *physical* database design as the major performance tuning tool is still a struggle against a bunch of interrelated parameters offered by the DBMS and/or the underlying file management system.

Physical database design techniques applicable in available RDBMSs generally include access path generation. The access path selection problem [Schk75], i.e. the decision which set of access paths should be generated to establish the best performance of queries and update transactions, has to be solved intellectually. The implicit assumption to this problem is that the relations found by the logical database design are also stored as "base" relations. While this particular design technique is available in all RDBMSs, a variety of other, sophisticated techniques have been devised and are offered by some system or the other. Examples are "clusters" in ORACLE [ORACLE], aiming at the tuple-to-pages allocation strategy, which allow to store tuples of different relations into one page according to identical values in common attributes. The main application of this feature is the efficient support of joins. Another facility for the same purpose are the "link fields" offered by the Research Storage System (RSS)[1] of System/R [As76, Ch81]—which, however are not utilized by the Relational Data System (RDS). Similarly, "coupling" in ADABAS [ADABAS] supports access from one tuple to another related by e.g. foreign keys. The "generalized access path" technique [Hä78] or the so-called "join indices" [LC86] are research proposals providing join support.

A common observation for the various approaches to physical database design is the following: The mapping from logical (tuples) to internal (records) data structures is mostly trivial. Tuples of the logical database relations are mapped one-to-one to internal records. Optimization issues like the important effect of *clustering* are deferred to the next deeper layer, namely the mapping of internal records to the pages (blocks). The overall objective of reducing the number of I/O operations neccessary to compute query results or perform updates is pursued by special techniques for this page-level database layout. ORACLE clusters, for instance, can be defined to let the DBMS allocate space for "member" tuples on the same page as their "owner" tuple (Tuples related by a key–foreign key relationship will be called owner and members in the sequel adopting these notions from the CODASYL model). Tentative TIDs can be given to the free place administration module to achieve the same effect. The distinction between a clustering and a non-clustering index is another example of such techniques.

The problem with these approaches is that query optimization, which is performed on a high level to reduce processing costs, especially algebraic optimization (cf. [Ul82, Ma83, ASU79, Scho86]) is too far from this page-level to take advantage of the clustering information. However, incorporating the influence of a specific clustering technique, e.g. in the selection of join algorithms, is an important issue in query optimization [ASK80, Ul82, WY76].

One approach which has not yet been mentioned so far is "denormalization" [SS80, SS81]. The idea, that has also influenced our direction, basically consists in storing materialized joins. However, as opposed to Schkolnick, who stays within the flat relational model to describe the materialized joins, our approach is considerably more general: we utilize (a subset of) the *nested relational model* (also called NF^2 model) for the (internal) record level. We see the following advantages: First, the various techniques applied in the physical database design can be expressed in a unique, formal manner. Second, the underlying NF^2 database kernel system (cf. [DPS86, PSSWD87, SW86]) can be exploited to efficiently manage relational applications. An additional advantage is that flat relations of course are a special case of nested ones, which allows to define the mapping between logical (flat) and internal (nested) relational schemata by means of a (nested) relational algebra. As a result, this formal transformation can be

[1]now called Data Base Storage System (DBSS) in SQL/DS

utilized in the algebraic optimization step, which then already reflects the physical database design. Thus, algebraically optimized query formulations also reflect the clustering strategy.

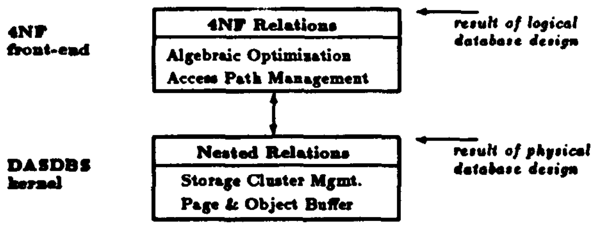

Figure 1: Supporting a 4NF front-end by an NF^2 kernel

Two related problems have to be solved when we follow this idea of utilizing our nested relational kernel DBMS "DASDBS" (Darmstadt Database System). The first one is physical database design: given a logical database schema and statistical data about the frequencies of certain types of retrieval and update operations (i.e. a "transaction mix"), find a physical database layout that guarantees optimal overall DBMS performance. Usually the number of I/Os is taken as the measure of system performance, computing time is mostly neglected. Our specific approach to this problem is to describe the physical layout by a formal data model, viz. NF^2 relations. While other projects aim at an extensibility of storage structures (e.g. STARBURST [LMP87]), i.e. special implementations of a relation can be added to the system, we show how different (wellknown) storage structures can be expressed by using nested relations as the internal data model. Defining an NF^2 relation for the internal database schema achieves a hierachical clustering strategy. All commonly used internal structures (or at least the important ones) can be represented in this model. Thus, we can in fact solve the physical design problem on this abstract level without loosing important structuring alternatives. Therefore, the second problem, to be solved at transaction processing time, viz. transformation (and optimization) of logical database schema level operations to the physical storage level, can be attacked in an algebraic fashion. The tranformation is simply performed by substituting algebraic definitions of the mapping between the two views into user operations. However, similar to a more classical setting, some sort of view optimization problem arizes. Redundant formulations of operations result from this substitution process, which can be eliminated by algebraic optimization techniques.

While preliminary ideas of our approach were described in [SS83] as a research programme for the project, this paper describes the actual results based on an implemented algebraic optimizer and on an implemented database kernel. Some of the theoretical aspects have been discussed in [Scho86].

The paper procedes by giving a summary of our kernel system, its data structures and operations in section 2. It is also shown how the kernel implements these structures on the underlying pages. In section 3 we discuss important physical structuring techniques and how these can be expressed in terms of nested relations. Section 4 describes the algebraic transformation and optimization step. It is shown that important types of queries (including joins) at the logical level can be mapped to simple operations at the internal level. The rationale behind our approach, improving performance by hierarchical clustering, is solicited by experimental evaluations reported in section 5.

2 DASDBS Kernel: Overview

2.1 Data Structures

DASDBS is a family of database systems based on a common kernel [SW86, PSSWD87]. This kernel can be considered the storage subsystem of the DBMSs in the family (cf. figure 2). Nested relations [AB84, FT83, RKS85, SS83, SS86] are the data structures available at the kernel interface. In contrast to traditional (flat, first normal form) relations, where attribute values are restricted to be atomic, i.e. undecomposable by the DBMS, *relations* are allowed as attribute values (subrelations) in this "non first normal form" (=NF^2) relational model. Thus, a hierarchical structure, viz. relations consisting of tuples with components that are relations in turn etc., is established.

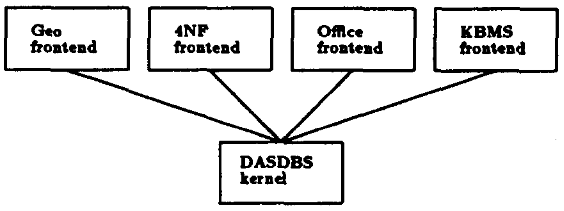

Figure 2: The DASDBS family

In the kernel, tuples of NF^2 relations—called Complex Records (CR) at this level—are implemented as *"Storage Clusters"* [DPS86]. This means, a CR is mapped to as few pages as possible. Particularly, if a CR spans pages, this set of pages is exclusively occupied by the CR. If, on the other hand, the CR is smaller than a page, several CRs may share the page. However, only CRs belonging to one (internal) relation may be stored on the same page. Thus, the definition of internal relations is our means of defining a (hierarchical) clustering strategy in a twofold sense: individual CRs are clustered on a minimal set of pages, and furthermore, all CRs belonging to a relation are stored on adjacent pages. Of course, the latter clustering effect can be disturbed by the database's dynamic behaviour, but not the former one! In contrast to GENESIS [Ba86], for instance, we apply *one unique storage technique* for Complex Records. Our argument for this decision is that we want to express different storage techniques on a higher level, namely by defining appropriate internal relations: the implementor of a DASDBS front-end can express his favourite storage technique in terms of Complex Records. In the sequel we give examples how this is done for some important structures.

The intra-record structure was designed in such a way that fast access to *complete* CRs as well as to *parts* of them (e.g. parts specified by a nested projection, see below) is acomplished. This structure guarantees getting the desired (part of the) CR into the page buffer by two I/O operations on the average. The first I/O reads the first page belonging to the CR (the "root page"). Based on the query and the information in the root page, the set of pages is determined that is needed to complete the request. A second (set-oriented, "chained") I/O operation is started to fetch this set into the buffer (see [DGW85, WNP87] for details). Furthermore, the structure applied on the page level was designed in such a way that the trivial case of CRs, namely flat tuples, is handled without overhead compared to existing relational DBMSs.

An example of an NF^2 relational schema, together with a corresponding flat relational (4NF) schema is shown in figure 3. Departments and employees are described as well as courses and the courses attended by employees. In the flat schema we marked the key-to-foreign key relationships by arrows. We will refer to this example throughout the following sections of this paper.

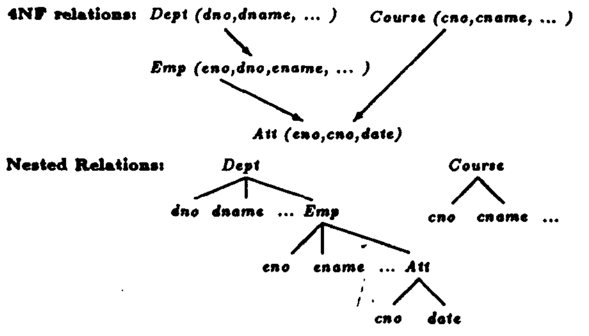

Figure 3: Example of 4NF and NF^2 relations

2.2 Operations

Subset of the nested relational algebra

Similar to the nesting of data structures (relations), nested operations in algebra/calculus [AB84, RKS85, SS86] or SQL [PA86, RKB85] style have been proposed for this model. A subset of the nested relational algebra defined in [SS86] is implemented within the kernel of DASDBS, the so-called "*single pass processible*" operations [Sche85, Scho86]. This term indicates the fact, that these operations can be computed efficiently, in a single (hierarchical) scan (i.e. with a complexity linear in the size of the relation) [PSSWD87].

The operations available at the kernel interface can be described as allowing application of relational selections and projections to every hierarchical level of a nested relation. Particularly, all single pass queries are *single table* (relation) queries. But additional restrictions have to be imposed to disallow selections with set comparisons introducing join complexity, for instance. For the schema above we can, for example, select department tuples with *dname* = 'Computer Science', project some of their attributes, e.g. *dno* and *Emp*. Furthermore, we want only the names (project on *ename*) of programmers (select *eskill* = 'Programmer'). In the nested relational algebra formulation this query would look like:

$$\pi[dno, \pi[ename](\sigma[eskill = \text{'Programmer'}](Emp))]$$
$$(\sigma[dname = \text{'Computer Science'}](Dept))$$

Notice that this query would have required a join in the corresponding flat relational schema, but is inexpensive in the nested one because of the following facts. The query is an example of a single-scan operation: the join between employees and departments is materialized and the additional conditions are checked during a single, hierarchical scan. If no access paths are available (the worst case) every department CR has to be inspected. This means, the root page of such a Complex Record has to be fetched and it is determined, which other pages have to be read in order to check the conditions. According to the storage strategy and to the header organisation [DPS86] only the minimum possible number of page accesses (without access paths) is required to execute this query. This observation is important, because it has been the driving force behind our idea of implementing a flat relational view on top of an NF^2 kernel system.

Another important observation, derived from the algebraically defined interface, is that all operations of the kernel are set-oriented. Thus *sets* of CRs are transfered to the calling program as the result of a query or given to the kernel for update operations (cf. the database portals approach of [SR84]).

Address selection

As we apply the NF^2 model to the storage structure level of our system, we extended the model by the notion of *addresses* of Complex Records. (On a higher level of abstraction we would use the term surrogates, i.e. a system provided, unique, stable identifier. On the kernel level it is known in addition that these identifiers allow very fast (direct) access.) A virtual attribute $@R$ is associated with each (internal) relational schema R. These addresses may be given outside the kernel. Thus, front-end systems can use these addresses to formulate *direct access queries* at the kernel interface. We introduced a special operation into the algebra to describe this kind of "query", the *address selection* ψ. For a given set A of addresses—obtained by previous retrieval operations—, $\psi[A](R)$ retrieves, via direct access, the set of Complex Records addressed by the set A. This mechanism may be used to construct and use access paths on top of the kernel. If, in our case, an index on *dname* were available, the ψ-operation only fetches the pages of those *Dept*-CRs that contain computer science departments.

Access costs

Our addressing scheme [DPS86] is a hierarchical one, namely a combination of the tuple identifier (TID) concept and (sub-) tuple numbers. The beginning of a Complex Record (= Storage Cluster) is addressed by a TID and any subrecord is identified by a sequence number which, in turn, is used to identify the corresponding page number in the header of the Storage Cluster. The discussion of this technique is contained in [DPS86] and shall not be repeated here. However, for the purpose of physical design the access costs listed in table 1 can be derived from the addressing scheme, whatever the length and nesting depth of a CR is.

	a	b	c	d
CR	$1+\alpha$	1	1	1
ΔSR	0	0	1	2

Table 1: Number of page accesses for the beginning of a Complex Record (CR) and additional page accesses for the beginning of any subrecord (ΔSR)

(a) is the case of a Complex Record which is shorter than one page. Here, α ($0 \leq \alpha \leq 1$) accounts for the amount of extra access due to overflow-TIDs as in the usual TID concept. In (b) we consider the case that a CR is larger than a page but the desired subrecord is also stored on the header page whereas in (c) we need one additional page access for the subrecord. In (d) we need two additional page accesses which are necessary for a very large Storage Cluster whose *header* does no fit into the root page. In this case we need one additional page access for the part of the header which contains the page number for the subrecord. A reasonable physical design of Storage Clusters, however, should avoid case (d) and preferably produce cases (a), (b), and (c). Furtheron, if not only the beginning of a whole Complex Subrecord is desired, i.e. if the whole *set of subrecords* has to be fetched, ΔSR in table 1 can be interpreted as the number of additional *page set* requests to be supported by chained I/O if offered by the operating system.

3 Physical Database Design Using Nested Relations

The overall objective of physical database design in general is to find an optimal internal representation of the logical schema w.r.t. a given (or estimated) workload of transactions. The number of I/O operations is used as the most important indicator of DBMS performance. Thus the general guideline of physical design is: data that is needed together should be stored together on one page if possible or on neighbouring pages (i.e. *clustering*). In the first case we are quite sure that the result of the query is obtained by one block access. The latter is only useful, if the mapping from database pages to blocks preserves neighbourhood and when chained I/O is exploited. Then, in both cases one I/O operation fetches the necessary data into the buffer without the need for further I/Os. An optimal performance w.r.t. queries would be achieved if all query results are internally clustered on one or a few neighbouring pages. This, however, is impossible in general without introducing redundancy, which in the case of updates causes overhead due to the maintenance of consistency among the multiple copies. Thus query and update ratios have to be considered carefully to find an optimal compromise. As commercial RDBMSs apply almost trivial mappings from logical to internal structures, the only *DBMS controlled redundancy* that can be introduced are the generated access paths. Additional redundancy can only be introduced on the logical level, which, however, gives responsibility for consistency to the user. In contrast to this, our approach allows for redundancy in the mapping to internal representations. Therefore, the DBMS can take the necessary actions to preserve consistency among the replicated data.

In the sequel we will show how to use the hierarchical structures of NF^2 relations in the physical design for a flat relational logical database schema. Because data have to be mapped to a linear (block structured) storage space of the physical device, using hierarchies for the description of the clustering strategy does not impose any restrictions. This is because hierarchies are the most general structures that can be linearized without introducing redundancy or references. Obviously, by using references to subobjects instead of subobjects themselves, non-hierarchical structures can be stored without redundancy in primary data. Only the references are redundant in some sense (cf. "key redundancy" in the relational model). In our discussion we will introduce redundant auxiliary data to represent non-hierarchical structures stepwise by using references first, and replicated data in a subsequent step. Emphasis is put on support for join operations, although we start with an easier case. We assume that the flat relations from figure 3 are given as the logical database schema. Several alternatives of internal representations for parts of this schema will be presented.

3.1 Alternatives for Single Relations

Already for the simple case of a single relation a variety of storage structures can be considered: The trivial structure, of course, is to apply an identity mapping. One tuple of the logical relation becomes one Complex Record (the special case of a "non-complex" one is included in the model). This is the usual technique applied in commercial systems. Horizontal or vertical partitioning (with or without redundancy) can be expressed by generating more than one internal relation obtained by selections or projections of the logical relation, respectively [CNW83, MS77, NCWD84]. These techniques aim at efficient support for selections (mostly in distributed database environments, where tuples of a relation get partitioned on several sites according to frequent accesses) or projections on frequently used attributes (to avoid reading attributes into the buffer that are rarely used). On the other hand, retrieving whole relations/tuples becomes less efficient, of course.

An important structuring alternative found in practice is "sorting" a relation w.r.t. certain attribute values. The sort order is preserved by an appropriate indexing structure ("*clustered index*" [SQL, ORACLE]). Sequential access in sort order as well as selections on the ordering attribute(s) are supported by this mechanism. We can model this technique by creating one Complex Record for the set of logical tuples having identical values in the ordering attribute (or values in a certain range, to be more general). The indexing part, however, is managed by the front-end not by the kernel itself.

For instance, if employee tuples shall be clustered according to identical department numbers (clustered index on *dno*), we use a *nesting operation* which yields one tuple for each department number having all employees as subtuples:

$$\begin{array}{ll} Emp & (\underline{eno}, dno, ename, \ldots) \\ & \downarrow \; IEmp := \nu[Demps = (eno, ename, \ldots)](Emp) \\ IEmp & (\underline{dno}, Demps(\underline{eno}, ename, \ldots)) \end{array}$$

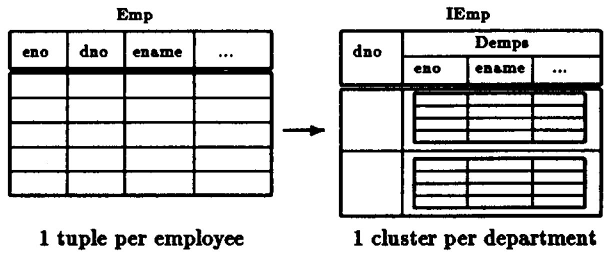

Figure 4: Clustering employees w.r.t. departments

While in a trivial representation (with unclustered index) a query like "$\sigma[dno = 42](Emp)$" would require (roughly) one I/O per matching tuple (i.e. $height(index) + N$), a clustered index needs $height(index) + n$, where n is the number of pages necessary to store all *Emp* tuples (the number of which is N) for this particular department. Obviously, $n \ll N$, thus a clustered index performs better. On the other hand, in case of updates on *dno*, tuples would have to be moved between pages to maintain the clustering. (Most systems do not move and thus clustering is not kept in a strict sense.) With our nested representation the above query performs even better, because we need the same number of I/Os for the index and the data, but the kernel can fetch the whole Complex Record in 2 I/O operations: one for the root page of the record, and one (set-oriented, chained I/O) for the rest, cf. section 2.1. Given that the operating system allows chained I/O ([PSSWD87, WNP87]) we perform better, but never worse. However, in case of updates on *dno*, we are forced to maintain clustering by deleting the empolyee subtuple in the old department's CR and insert it into the new one.

Another difference exists since we maintain the clusters: assume an index is generated on employee names which provides us with addresses. Then, in the flat storage structure $1 + \alpha$ page accesses would be necessary to fetch the required page. If we store employees according to the *IEmp* schema above, the address of an employee (also obtained by an index) is a hierarchical one: it consists of the TID of the Storage Cluster and of a (relative) subtuple number for the employee. Therefore the number of page accesses for one specific employee may vary between 1 and 3 (table 1).

Without redundancy we can only create *one* clustered index. Therefore, support for other selections can only be established by unclustered indices.

3.2 Non-Clustered Indices

Like all available DBMSs, we can use non-clustered indices in the relational frontend to the DASDBS kernel. Addresses of tuples are known outside the kernel and can be used to build access paths on top. The address attributes, however, can also be used to build "link fields" (RSS of System/R [As76, Ch81]), see below. From a systematic point of view our approach can be characterized as clustering references to Complex Records instead of the records themselves. This allows for multiple clustering strategies at little costs due to redundancy. Only the references are kept in redundant representation. In our example, we can have an index on *dno* as well as on *ename*, both having references to *IEmp* tuples (here the address attribute $@IEmp$ is stored like ordinary attributes in the index entries):

$$\begin{array}{rl} IEmp & (\underline{eno}, ename, dno, \ldots) \\ Idnoindex & (\underline{dno}, IEmprefs(@IEmp)) \\ Ienameindex & (\underline{ename}, IEmprefs(@IEmp)) \end{array}$$

The influence on performance of this type of indices is known from traditional systems. Again we can profit from the set-orientation of the kernel: given a reference list $IEmprefs$ obtained from either index, a single set-oriented I/O request (using the address selection operation ψ) can be used to fetch the addressed $IEmp$ records. While chained I/O may be not profitable in the case of non-clustered $IEmp$ records, it certainly is not worse than multiple single I/Os, thus we can only gain and not loose.

As for the drawbacks of these indices, it is clear that maintenance costs arise during updates on inverted attributes. There is no difference of our solution and the classical ones.

3.3 Join Materialization: 1:n–Relationships

Join index

Now we discuss the support of joins, notably the most expensive operation. In practical applications, access paths are generated that support all important join operations, because this is the only possibility. A special kind of "join index" has been proposed in [Hä78, LC86] and implemented in ADABAS for the "Coupling" [ADABAS] of relations related by key–foreign key relationships. There, in a single index structure (B^+-tree) on the join attribute(s), the leaf-nodes contain two kinds of references. If in our example *IDept* and *IEmp* were stored as two internal tables, a join index on these two could be

$$IDE(\underline{dno}, @IDept, IEmprefs(@IEmp))$$

Obviously, using the internal addresses stored in IDE, we can easily find a department record and all corresponding employee records.

Denormalization

In research, another technique has been proposed that influenced our project very much. "Denormalization" was proposed by Schkolnick and Sorenson [SS80, SS81] as a means of (internally) materializing joins. In addition to the logical relations, the most frequent join results are stored internally. Thus, these joins become inexpensive operations at the cost of redundancy which causes overhead for updates. Nevertheless, performance gains by factors of 3 to 4 were reported in [SS81]. This technique has already migrated into practice: the database administrator's guide of SQL/DS [SQL] recommends "storing the join of tables" as a very effective performance tuning tool. However, as the user and application programs are not shielded against such optimizations, i.e. they are performed on the *logical* schema level, all operations on the database have to be reformulated accordingly! In particular, update transactions have to take care of updating the redundant representation.

The problem of loosing "dangling tuples" in the join was the reason for storing joins *plus* the original relations and not using the join *instead* of the logical relations in [SS80, SS81]. This problem, however, can easily be solved by using *outer joins*, which introduce null values for dangling tuples. A second problem, and a more substantial one, is the redundancy in the join relation. A tuple from the one relation is repeated for every matching tuple in the other one. Both problems can be solved by using a *nested* relation for the join result. Furthermore, as the join relation is an *internal* one in our approach and the logical schema is not changed, the user and application programs are not affected, as opposed to the SQL/DS recommendation.

First consider two relations related through a foreign key (*Dept* and *Emp* in our example). Without any redundancy, we can internally store the nested relation *IDept* with *IEmp* as a subrelation.

$$IDept(\underline{dno}, dname, \ldots, IEmp(\underline{eno}, ename, \ldots))$$

Departments without employees result in a Complex Record with an empty subrelation. This quite naturally indicates that there are no employees. A slight problem occurs, if an employee is not yet assigned to any department. For all these employees we internally introduce a special null-valued department.

This stucture closely corresponds to the intention of ORACLE clusters [ORACLE], were *Emp* tuples can be stored on the same page as their *Dept* tuple. However, ORACLE neither guarantees this clustering, nor is a page-spanning neighbourship established. In SQL/DS a similar effect can only be achieved by an appropriate initial loading sequence of tuples from the owner and member relations. However, the join between *Emp* and *Dept* has to be executed if required, which results in requesting the same page several times. So, join queries benefit from this structure, because the required page is already in the buffer with a high probability.

In our approach, however, the system *knows* that all join partners (*Emp* subtuples) are contained in the Storage Cluster (*Dept* tuple). Thus, the corresponding pages are requested only once. Access paths on *dno* can be applied in addition to support selective access. On the other hand, let us consider the kind of operations that become more expensive. Sequential processing of *Dept* tuples is more costly, because only one tuple can be found on a page as opposed to clustering several of them on a page in a separate *IDept* relation without *IEmp* subrelation. If an employee is assigned to another department, this update is also more expensive as it must be implemented as (subtuple) delete plus (subtuple) insert in the new department as opposed to simply changing an attribute value (*dno* in *Emp*). Sequential processing of employees is only slightly worse than in a separate *IEmp* relation, because one *Dept* tuple shares a (set of) page(s) with a number of *Emp* tuples, which would be clustered on fewer pages otherwise.

3.4 Join Materialization: n:m–Relationships

Similar to the consideration of clustered indices vs. non-clustered ones, materializing joins can only be performed in one direction without introducing redundancy. For example, consider our relations *Emp*, *Att* and *Courses*. *Att* is related to both *Emp* and *Courses* by a foreign key. This is because an n:m (many-to-many) relationship exists between employees and courses, a *non-hierarchical* (network) structure is contained in our logical database schema. So, what is our solution to this problem? At first sight, our hierarchical kernel interface seems to be overstrained. However, let us inspect the solutions that other systems found for the representation of many-to-many relationships. Relational systems use foreign keys and three separate relations, hierarchical systems like IMS [IMS] offer several alternatives based on two or three "physical databases" with additional "virtual pointers" (in "logical databases") or have to introduce two redundant hierarchical views. Network database systems according to the CODASYL DBTG proposal [Ol78] use three record types (relations) and two set types (key–foreign key relationships) and allow several clustering alternatives (SET MODEs).

Summarising, we can state that no storage structure can be found that allows symmetrical treatment of many-to-many relationships *and* clustering without redundancy. So, our choices are either introducing redundancy to allow clustering in both hierarchical views or use references instead of objects for the clustering (which is also a restricted kind of redundancy).

As we could expect form the above analysis, no new structures can be found for the representation of many-to-many relationships. Instead, we can apply any of the alternatives discussed for the simple case in section 3.3, but now the representations of both hierarchical views (from *Emp* to *Att* and from *Courses* to *Att*) are interrelated and have to be evaluated in parallel. Determining an optimal internal structure has to take into account both views with their corresponding operational characteristics.

Thus we have a variety of alternatives, some of which are sketched in figure 5. The internal representations which are sorted in order of ascending degree of redundancy. (Symmetrical alternatives are omitted.) As for the discussion of operations that benefit from these structures and those that become more expensive, the same arguments apply as in the simple case in section 3.3. As a general guideline we can state: the more frequent update operations on the foreign keys are, the more likely are references instead of subobjects. If both directions are frequently updated, the solutions (2) or (3) in figure 5 may be good choices, because there is little overhead, but nevertheless, queries are well supported. The subrelations containing the references in (3) can be considered "declustered" access paths. A join index (2) would be clustered on its own, while here the references are clustered with the one relation. Therefore, maintenance costs in case of updates (on the foreign key) are comparable to access paths. A reference must be eliminated from one list and appended to another. Additional references may be introduced in any of the alternatives, pointing from "members" to "owners" (in addition to the "logical pointers", viz. foreign keys). This possibility is sketched in the *Att* subrelation of *Emp* in (5). Whenever one hierarchical view of the relationship is more important (i.e. frequently used) than the other alternative (4) can be selected (perhaps combined with materialized references (5)).

Clearly, if updates are very infrequent or not critical w.r.t. performance, redundant representations (6,7) are the best, because they support queries most efficiently. In practice, those many-to-many relationships are often not subject to updates. In our example the fact that an employee attended a certain course would not be updated lateron. Therefore, redundancy as an internal means of ameliorating performance should be considered carefully, not only in the special appearence of access paths.

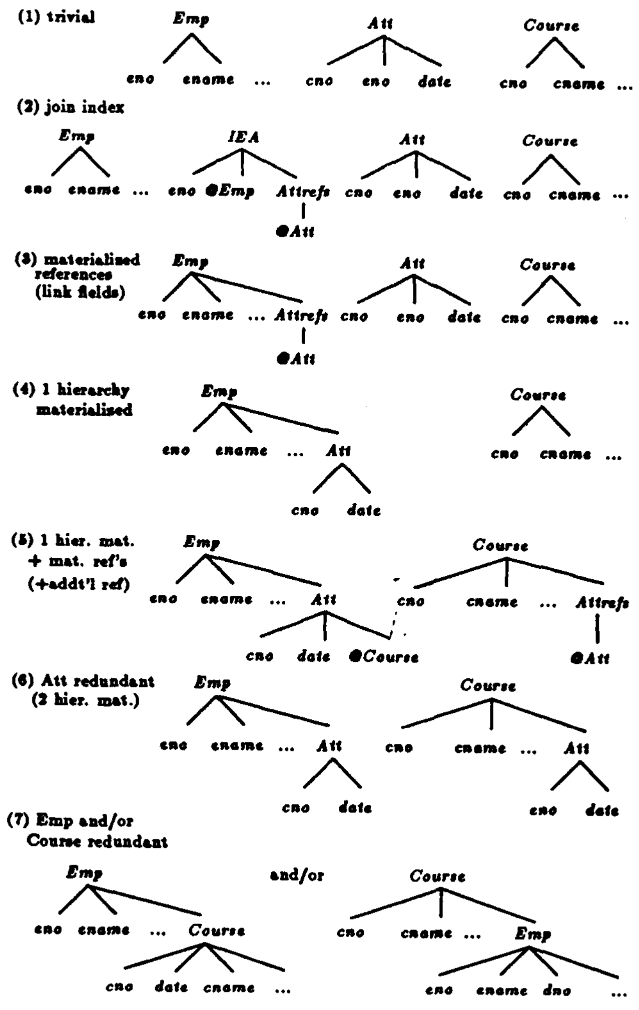
(1) trivial
Emp
cno ename ...
Att
cno eno date
Course
cno cname ...
(2) join index
IEA
cno @Emp Attrefs
@Att
(3) materialised references (link fields)
(4) 1 hierarchy materialised
(5) 1 hier. mat. + mat. ref's (+addt'l ref)
@Course
(6) Att redundant (2 hier. mat.)
(7) Emp and/or Course redundant
and/or
cno date cname ...
eno ename dno ...

Figure 5: Some storage alternatives for m:n–relationships

Summarizing we see the need for a physical design optimization supported by some piece of software. An algorithm that solves the whole problem (with reasonable effort) is not in sight, but heuristics will help in finding initial physical database layouts to start with. Currently, we are working on such heuristic approaches. Considering spanning forests of the logical schema network (relation schemata connected by key-foreign key relationships in an acyclic graph) is a starting point for these physical design aids.

If we want to support more general joins, the generalized access path technique of [Hä78] can be applied. While we only considered key–foreign key joins until now, suppose that a join on other common attributes is a frequent operation. For instance we can think of employees and departments being joined on locations (*dloc* is the location of departments and *eloc* describes where employees live, *loc* is declared on the union of both domains). Now, every index entry for a specific location would contain ***two sets of references***, one for departments and one for employees:

$$GAPloc(\underline{loc}, IEmprefs(@IEmp), IDeptrefs(@IDpet))$$

4 Query Optimization

4.1 Algebraic Optimization

The problem discussed so far, physical database design, has to be solved offline, i.e. before the database is installed, or when a reorganization is planed. Our second problem, however, is a dynamic one in the sense that it must be solved during query (and update) *execution* time. Operations issued by the user against the logical schema of the database have to be optimized and transformed to the internal schema, which has been determined by physical database design and can be quite different. The advantage of describing the internal database layout by means of our nested relational model turns out to be that the transformations between the two schemata can be defined formally using the (nested) relational algebra.

Let us reconsider the example given in figure 3 and let us assume the 4NF view as the logical (L) and the NF^2 relations as the internal (I) database schemas respectively. Then the mapping from L to I is defined by the following algebraic expressions (using the notation from [SS86], ν means nesting):

$$\begin{aligned} ICourse &:= Course \\ IDept &:= \nu[IEmp = (eno, ename, \ldots)](\nu[IAtt = (cno, date)] \\ &\quad (Dept \boxtimes Emp \boxtimes Att)) \end{aligned}$$

Notice that we used the outer join operation ($\boxtimes$) to avoid loosing "dangling tuples". In general, to preserve information during the transformation, we have to guarantee "losslessness" of the mapping. This notion, initially used in [ABU79] for project-join sequences only, has to be extended to apply to other operations too. We rather use the term "invertibility" [SS83, Scho86] to emphasize the fact that the information content has to be reproduced if we apply the inverse transformation from I back to L. Obviously, all physical design techniques have to guarantee invertibility.

For our example, the inverse transformation is defined by (μ denotes unnesting):

$$\begin{aligned} Course &= ICourse \\ Dept &= \pi[dno, dname, \ldots](IDept) \\ Emp &= \pi[dno, eno, ename, \ldots](\mu[IEmp](IDept)) \\ Att &= \pi[eno, cno, date](\mu[IAtt](\mu[IEmp](IDept))) \end{aligned}$$

These equations can now be used by the relational front-end to transform a user query from logical to internal relations by simply substituting the right-hand sides for the logical relations mentioned in the query. For our example consider a query asking for the name of Smith's department:

$$\begin{aligned} LQuery &= \pi[dname](Dept \bowtie \sigma[ename = \text{'Smith'}](Emp)) \\ &\downarrow \\ IQuery &= \pi[dname](\pi[dno, dname, \ldots](IDept) \bowtie \\ &\quad \sigma[ename = \text{'Smith'}](\pi[dno, eno, ename, \ldots] \\ &\quad (\mu[IEmp](IDept))) \end{aligned}$$

The need for algebraic optimization techniques is obvious, as simple formal substitution yields a formulation, where the join operation is still present! However, intuitively it is clear that the join in $IQuery$ is redundant, because two projections of $IDept$ are joined on dno, the key attribute. Nevertheless, from a formal point of view, join elimination criteria like those of [ASU79] are not applicable, because the unnest operation μ is not included in their criteria. In fact, we have developed a theory of algebraic optimization in this new context that proves our intuition: the join can be eliminated.

The idea of our approach is the following: the expressions obtained by the substitution of equations defining the inverse transformation are "almost flat relational", which means the only nested relational operation is μ. If we now apply a total unnest operation μ^* (cf. [FT83]) to the internal relations in all expressions defining the mapping $I \to L$ we obtain identical flat (!) relations in the expressions ($\mu^*(IDept)$ in our example) defining C-relations contained in the same I-relation.

$$\begin{aligned} IQuery &= \pi[dname](\pi[dno, dname, \ldots](\mu^*(IDept)) \bowtie \\ &\quad \sigma[ename = \text{'Smith'}](\pi[dno, eno, ename, \ldots] \\ &\quad (\mu^*(IDept))) \end{aligned}$$

Then flat relational optimization techniques become applicable. However, we have to solve another problem before: a special type of null value is introduced by the outer join operation to guarantee invertibility. The nest and unnest operations had to be redefined in order to map null values to empty sets (subrelations) and vice versa. Furthermore, we must be able to drop tuples of the totally unnested relation that contain null values in specific attributes (because the null values were not present in the logical relations): a reduction operation "ρ" was introduced for this purpose.

$$\begin{aligned} IQuery &= \pi[dname](\pi[dno, dname, \ldots](\rho[dno, dname, \ldots] \\ &\quad (\mu^*(IDept))) \bowtie \\ &\quad \sigma[ename = \text{'Smith'}](\pi[dno, eno, ename, \ldots] \\ &\quad (\rho[eno, ename, \ldots](\mu^*(IDept)))) \end{aligned}$$

Some additional algebraic equivalences incorporating this reduction operation ρ were necessary to allow application of traditional relational join elimination techniques, e.g. tableaux [ASU79]. Particularly, we can prove that select-project-join queries on the logical 4NF schema (with conjunctive selection formulae) can efficiently be optimized and transformed to the internal nested relational schema. Moreover, the queries resulting from this transformation are single pass processible (see above), iff all joins contained in the logical query are already materialized in the internal database [Scho86] (the "superselection" of VERSO [Ab86] can also express such materialized join queries):

$$IQuery = \pi[dname](\sigma[\sigma[ename = \text{'Smith'}](IEmp) \neq \emptyset](IDept))$$

4.2 Non-Materialized Joins, Access Paths

The algebraically optimized query is suitable for direct execution by the kernel, if all joins in the query are materialized. However, this is only the best case. Usually, some joins are left which must be computed on top of the kernel by repeated calls and according to a strategy which must be determined carefully. Notice, that references (addresses) to join partners may only be materialized instead of the partners. As an example consider a query

$$\pi[ename](Emp \bowtie \sigma[dname = \text{'Computer Science'}](Dept))$$

and assume an internal representation that contains materialized references to $IEmp$ in $IDept$:

$IDept$	$(\underline{dno}, dname, \ldots, IEmprefs(@IEmp))$
$Idnameindex$	$(\underline{dname}, IDeptrefs(@IDept))$
$IEmp$	$(\underline{eno}, ename, dno, \ldots)$

In addition we may have separate access paths ($Idnameindex$) or join indices which have been generated to support joins. Even if the algebraic optimization could eliminate all joins there is still the problem of how to process the resulting single-relation and single-scan operation by the kernel if one or several access paths exist on that relation!

Thus we still have to select access paths to process queries, i.e. to generate access plans. We have not yet found exciting new ideas to solve this old problem. Our strategy is the following: every scan of a kernel relation can be restricted to a subset of records by a predetermined set of addresses (as an input for the address selection). The worst case is a (full) *relation scan*, i.e. we never need a segment scan. However, we can spend some effort in obtaining a smaller set of addresses by considering access paths. Obviously, the amount of I/O for the construction of such address sets should be small compared to a full scan. In our example above the strategy would be to use the index on *dname* first to find the references to *IDept*. Then a (direct access) query fetches the corresponding *Idept* tuple(s). The materialized references contained therein can be used to retrieve the *IEmp* tuples in a third step (again a direct access query, but now set-oriented, i.e. all matching *IEmp* tuples are obtained by one kernel call):

1. Use access path:
   $DREFS := \sigma[dname = \text{'Computer Science'}](Idnameindex)$
2. Get department tuple with embedded references:
   $EREFS := \pi[IEmprefs](\psi[DREFS](IDept))$
3. Retrieve employee tuples:
   $RESULT := \pi[ename](\psi[EREFS](IEmp))$

In contrast to the approach of System/R and SQL/DS, where only *one* index could be selected to open an "index scan" or *none* to open a "segment scan", our system performs merging operations on sets of addresses from various sources (embedded references, references in join indices, access paths) to construct minimal sets for the address selection.

One negative result of our investigations should also be reported here: When we started our project in this direction [SS83] we saw a chance to combine the execution plan generation with the algebraic optimization in one step. The motivation was that all data including indices are represented as nested relations and the operations (including direct access via addresses) are described by the (nested) relational algebra. The result of such a combined step therefore could have been an optimized sequence of nested relational algebra expressions to be executed directly by the kernel. Obviously, this has not been achieved by the optimizer described in the previous section. The reason is that the algebraic optimizer *eliminates redundant* operations, whereas the access path selection afterwards *introduces additional* operations, namely those on the indices. From an algebraic point of view, these additional operations are redundant. Therefore, a combined optimizer must consider *sequences* of algebraic expressions that are dependent from each other. A single cost function has to be found that combines the costs of algebraic operations and the transfer of data between subsequent operations in a unique fashion. While there is still a hope that an elegant combined solution can be found, we have decided to follow the classical separate approach.

Thus, the architecture of the flat relational DBMS front-end for the DASDBS kernel consists of three main modules (cf. figure 6): an algebraic Transformation and Optimization Processor (TOP), a Multi Pass query (and update) Processor (MPP). Here access path selection is performed and join processing is realized by repeatedly calling the underlying Single Pass Processor (SPP). The difference between the SPP and the kernel is, that access paths can be evaluated and maintained by the SPP, while access paths are just ordinary internal relations for the kernel interface. Hence,

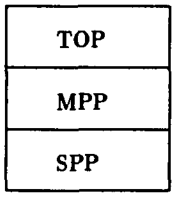

Figure 6: Architecture of the DASDBS relational front-end

query processing using access paths is separated in a part located at the MPP module, namely finding the strategy, and the operational part within the SPP for single pass queries. In the case of multi pass operations, the SPP is used by the MPP in a nested loop like manner.

5 Evaluation

As the very first version of our nested relational DASDBS kernel came up running just a few days ago, we are currently unable to report on performance evaluations of our own system. However, previous practical experiments can be taken as strong indications that the direction we chose is a promising one: previous work on join elimination for relational DBMSs (System/R) was reported in [SS80, SS81]. Results achieved there indicate performance gains of factor 3 with a transaction mix containing not only those queries that benefit from denormalization, but also some that perform worse and updates [SS81]. The evaluations were based on the cost estimates of the System/R optimizer [ASK80]. These evaluations did not include the algebraic optimization step as the denormalized relations were visible in the logical schema. Join elimination (algebraic optimization) *and* execution of the queries was investigated in [OH85]. The figures obtained there are very drastical: factors of 500 to 1000 were achieved in elapsed time for single queries involving joins as opposed to optimized queries with 2 redundant joins removed.

In order to interpret and understand these previous results we started a series of own evaluations, also based on SQL/DS. The objective of our experiments was to gain insight into the importance of clustering and/or access paths for join support. To a certain extent we tried to simulate the effect of a nested relational join materialization by a suitable loading sequence of tuples from the involved relations and by creating indices. The following observations, however, must be kept in mind:

- Even if an owner and its member tuples are clustered within a page in the SQL/DS sense, we still have to create appropriate access paths in order to get direct access to the members in a join. Otherwise the system would not know that the join partners are already there and would search for them.
- The optimizer decides whether an index scan or segment scan is opened. If two indices are available, at most one is used.

One might think of a more sophisticated access plan generation, but our approach would move some of these problems into the algebraic optimization step.

For the flat relational schema from figure 3 we generated three SQL/DS databases each with 25,000 employees, 1,000 departments, 200 courses and 75,000 attended courses. The physical layout of these databases was differing in the clustering strategy achieved by the loading sequences:

4NF-DB: *Emp*, *Dept* and *Course* were loaded in ascending key sequence, i.e. clustered, unique indices were available on *eno*, *dno* and *cno* respectively. *Att* was loaded in a random sequence with a unique index on *eno,cno*. Relations were loaded one after the other. An additional index to *Emp* on *dno* was created to support joins.

Sort-*dno*-DB: Similar to 4NF-DB, except the fact that *Emp* tuples are loaded in ascending *dno* sequence, i.e. now the *dno*-index is the clustered index.

"NF^2"-DB: Utilizing the fact that tuples from several relations are stored in the same page, if they are inserted in the corresponding sequence, an internal layout was established, that simulates our NF^2 database structure as closely as possible. For each *Dept* tuple the corresponding *Emp* tuples and their *Att* tuples were loaded in sequence. This simulates the NF^2 schema shown in figure 3.

A series of queries was run against the three databases, performance of SQL/DS was evaluated in terms of I/Os and DBSS-Calls. Sample queries reported in table 2 are (in algebra notation):

Q1: $\sigma[eno = 4242](Emp) \bowtie Dept$

Q2: $\sigma[dno = 42](Dept) \bowtie Emp$

Q3: $\sigma[ename = \text{'Jones'}](Emp) \bowtie_{dno} \sigma[ename = \text{'Smith'}](Emp) \bowtie Dept$

Q4: $\sigma[dno = 42](Dept) \bowtie Emp \bowtie Att$

Moreover, query 1 was evaluated with the indices mentioned above and ***without any index*** in order to get a quantitive measure of the influence of indices on join processing costs. The results are contained in table 2. The following observations can be drawn:

- The join without any index support (segment scan) is by a factor of 300–400 slower than the one with index (index scan). Obviously, this factor depends on the size of the segment. Query 1 without index perform equal on the 4NF and Sort-*dno* databases.
- Clustering employees according to department numbers makes query 2 run faster, because all accessed employees share a (small set of) page(s). The "NF^2" structure is even better, because the *Dept* tuple is also on this (set of) page(s). A factor of 2 can be observed compared to the 4NF schema (unclustered). For the DASDBS kernel, we expect even more performance enhancements, because the pages containing employees of a department are fetched into the buffer by ***one chained I/O***.
- In query 3 the SORT-*dno* database performs better than "NF^2", because of the superior clustering for this particular query. In the NF^2 structure, also the *Att* tuples are clustered with the departments, but this is not needed in query 2. In the "NF^2" database the employee relation spans 2,140 pages, while in the other two databases 1,250 pages are enough to keep all *Emp* tuples. Nevertheless, "NF^2" performs better than 4NF. As indicated in query 2, without the *Att* tuples a two-level "NF^2" structure would be superior to Sort-*dno*.
- Query 4 takes full advantage of the "NF^2" three-level clustering strategy, thus this structure outperforms all of the others. In particular, a factor of 2 is achieved compared to the "usual" clustering approach (Sort-*dno*).

	4NF		Sort-*dno*		"NF^2"	
	I/Os	Calls	I/Os	Calls	I/Os	Calls
Q1 w/o ind.	5528	39	—	—	4305	39
Q1 w. ind.	14	42	16	42	13	42
Q2	35	61	17	61	14	61
Q3	3537	559	1306	514	2189	514
Q4	131	193	96	193	46	193

Table 2: Performance of selected queries

6 Summary and Outlook

We discussed the issue of physical database design expressed in terms of the nested relational model. As an example for a logical schema we considered flat relations. Emphasis was put on techniques that efficiently support join processing. Common structures like indices, join indices or link fields have been presented as well as more unusual ones, namely denormalization. All of these techniques can be described formally by Complex Records of the DASDBS kernel, i.e. as nested relations. Moreover, the substantial problems introduced by the most effective join support mechanism, denormalization, namely redundancy and loss of information can easily be avoided by using ***nested*** join relations.

Besides the formal description, which is an advantage on its own, we could benefit from the fact, that the front-end data model (flat relations) is a subset of the internal one: the transformation of operations issued to the database from the logical schema view to the internal representation can be performed easily. The mappings between internal and logical relations are defined as (nested relational) algebra expressions. Thus, simple substitution of these expressions into the user queries yields internal equivalents. However, similar to the usual view optimization problem, there is the need for algebraic optimization in order to avoid computing redundant operations. Our aproach to this problem was to find transformations of the nested algebra operations that allow application of known join elimination techniques. To establish this goal, null values were introduced to achieve information preserving mappings, the algebraic operators were extended accordingly and a special reduction operation was introduced to eliminate null values when necessary.

The kernel interface allows to generate access paths on top of it, as addresses of internal records can be given outside. Such addresses can later be used to formulate direct access queries. While our aim at the beginning of research in this direction was, to include access path selection at query execution time in the algebraic optimizer, our current solution pursues the classical two step approach of a separate "access plan generation" after the algebraic optimization.

Physical database design has been discussed systematically by relaxing the paradigm of avoiding redundancy in the database. A distinction between primary data (contained in the logical relations) and derived, auxiliary data (e.g. references, access paths) was useful in the discussion of ***system controlled redundancy*** introduced to enhance performance. This way, we could also show that non-hierarchically related data can be represented by our kernel. We presented several alternatives for many-to-many relationships mapped to the kernel data structures. These included all structures found in, e.g. network database systems. Therefore, other front-ends for the DASDBS kernel, including those that support some notion of "shared subobjects" [Ro86], "Complex Objects" [LKMPM85] or other types of structures [BB84, Mi87] will also find the kernel an appropriate storage system as far as structuring is concerned. In contrast to the relational front-end, however, operations of such models cannot mapped to kernel operations so easily by simply substituting algebraic expressions.

The advanced modelling facilities of those data models include features like recursive structures or attribute inheritance that may introduce additional complexity to the physical design problem. However, the alternatives will also be characterized by either introducing redundancy or references. Thus, we think that our approach will also be appropriate for these applications.

7 References

[Ab86] Abiteboul, S., et al.: *VERSO, a DBMS Based on Non-1NF Relations*, TR 253, INRIA, 1986

[AB84] Abiteboul, S., Bidoit, N.: *Non First Normal Form Relations to Represent Hierarchically Organized Data*, ACM PODS, Waterloo, 1984

[ABU79] A.V. Aho, C. Beeri, J.D. Ullman: *The Theory of Joins in Relational Databases*, ACM TODS (4:3), 1979

[ADABAS] Software AG: *ADABAS: DBA Reference Manual*

[As76] Astrahan, M.M., et al.: *System R: Relational Approach to Database Management*, ACM TODS (1), 1976

[ASK80] Astrahan, M.M., Schkolnick, M., Kim, W.: *Performance of the System/R Access Path Selection Mechanism*, IFIP Congress, 1980

[ASU79] Aho, A.V., Sagiv, Y., Ullman, J.D.: *Equivalences Among Relational Expressions*, SIAM J. Comp. (8:2), 1979

[Ba86] Batory, D.S.: *GENESIS: A Project to Develop an Extensible Database Management System*, Proc. Int. Workshop on Object-Oriented Database Systems, Pacific Grove, 1986

[BB84] Batory, D.S., Buchmann, A.P.: *Molecular Objects, Abstract Data Types, and Data Models: A Framework*, VLDB, Singapore, 1984

[Ch81] Chamberlin, D.D., et al.: *History and Evaluation of System/R*, CACM, 1981

[CNW83] Ceri, S., Navathe, S., Wiederhold, G.: *Distribution Design of Logical Database Schemas*, IEEE TOSE (9:4), 1983

[Da86] Dadam, P., et al.: *A DBMS Prototype to Support Extended NF² Relations: An Integrated View on Flat Tables and Hierarchies*, ACM SIGMOD, Washington, 1986

[DGW85] Deppisch, U., Günauer, J., Walch, G.: *Storage Structures and Addressing Techniques for the Complex Objects of the NF² Relational Model*, Proc. GI Conf. Database Systems for Office Automation, Engineering, and Scientific Applications, Karlsruhe, 1985

[DPS86] Deppisch, U., Paul, H.-B., Schek, H.-J.: *A Storage System for Complex Objects*, Proc. Int. Workshop on Object-Oriented Database Systems, Pacific Grove, 1986

[FT83] Fischer, P.C., Thomas, S.J.: *Operators for Non-First-Normal-Form Relations*, Proc. IEEE COMPSAC, 1983

[Hä78] Härder, T.: *Implementing a Generalized Access Path Structure for a Relational Database System*, ACM TODS (3:3), 1978

[IMS] IBM Corp.: *IMS/VS: Data Base Administration Guide*

[LC86] Lehmann, T.J., Carey, M.J.: *A Study of Index Structures for Main Memory Database Management Systems*, VLDB, Kyoto, 1986

[LKMPM85] Lorie, R., Kim, W., McNabb, D., Plouffe, W., Meier, A.: *Supporting Complex Objects in a Relational System for Engineering Databases*, in: Kim, W., Reiner, D.S., Batory, D.S. (eds.): Query Processing in Database Systems, Springer, 1985

[LMP87] Lindsay, B., McPherson, J., Pirahesh, H.: *A Data Management Extension Architecture*, ACM SIGMOD, San Francisco, 1987

[Ma83] Maier, D.: *The Theory of Relational Databases*, Pitman Publishing Ltd., London, 1983

[Mi87] Mitschang, B.: *The Molecule-Atom Data Model* (in German), Proc. GI Conf. Data Base Systems for Office, Engineering and Scientific Applications, Darmstadt, 1987

[MS77] March, S.T., Severance, D.S.: *The Determination of Efficient Record Segmentations and Blocking Factors for Shared Data Files*, ACM TODS (2:3), 1977

[NCWD84] Navathe, S., Ceri, S., Wiederhold, G., Dou, J.: *Vertical Partitioning Algorithms for Database Design*, ACM TODS (9:4), 1984

[OH85] Ott, N., Horländer, K.: *Removing Redundant Join Operations in Queries Involving Views*, Information Systems (10:3), 1985

[Ol78] Olle, T.W.: *The CODASYL Approach to Data Base Management*, J. Wiley & Sons, Chichester, 1978

[ORACLE] Oracle Corp.: *ORACLE User Manual*

[PA86] Pistor, P., Andersen, F.: *Designing a Generlized NF² Model with an SQL-type Language Interface*, VLDB, Kyoto, 1986

[PSSWD87] Paul, H.-B., Schek, H.-J., Scholl, M.H., Weikum, G., Deppisch, U.: *Architecture and Implementation of the Darmstadt Database Kernel System*, ACM SIGMOD, San Francisco, 1987

[RKB85] Roth, M.A., Korth, H.F., Batory, D.S.: *SQL/NF: A Query Language for ¬1NF Relational Databases*, Techn. Rep. TR-85-19, University of Texas at Austin, 1985

[RKS85] Roth, M.A., Korth, H.F., Silberschatz, A.: *Extended Algebra and Calculus for ¬1NF Relational Databases*, Techn. Rep. TR-84-36, University of Texas at Austin, 1985

[Ro86] Rowe, L.A.: *A Shared Object Hierarchy*, Proc. Int. Workshop on Object-Oriented Database Systems, Pacific Grove, 1986

[Sche85] Schek, H.-J.: *Towards a Basic Relational NF² Algebra Processor*, Conf. Found. Data Org. (FODO), Kyoto, 1985

[Schk75] Schkolnick, M.: *The Optimal Selection of Secondary Indices for Files*, Information Systems (1:4), 1975

[Scho86] Scholl, M.H.: *Theoretical Foundation of Algebraic Optimization Utilizing Unnormalized Relations*, Int. Conf. on Database Theory, Rome, 1986, LNCS 243, Springer

[SQL] IBM Corp.: *SQL/Data System: Planning and Administration for VM/System Product, Release 3*

[SR84] Stonebraker, M., Rowe, L.A.: *Database Portals: A New Application Program Interface*, VLDB, Singapore, 1984

[SS80] Schkolnik, M., Sorenson, P.: *Denormalization: A Performance Oriented Database Design Technique*, Proc. AICA Conf., Bologna, Italy, 1980

[SS81] Schkolnik, M., Sorenson, P.: *The Effects of Denormalization on Database Performance*, Res. Rep. RJ3082 (38128), IBM Res. Lab. San Jose, Ca., 1981

[SS83] Schek, H.-J., Scholl, M.H.: *The NF² Relational Algebra for a Uniform Manipulation of the External, Conceptual, and Internal Data Structures* (in German), in: J.W. Schmidt (ed.), Sprachen für Datenbanken, IFB 72, Springer, 1983

[SS86] Schek, H.-J., Scholl, M.H.: *The Relational Model with Relation-Valued Attributes*, Information Systems (11:2), 1986

[SW86] Schek, H.-J., Weikum, G.: *DASDBS: Concepts and Architecture of a Database System for Advanced Applications*, TR DVSI-1986-T1, Techn. Univ. Darmstadt, German Version to appear in: Informatik Forschung und Entwicklung

[Ul82] Ullman, J.D.: *Principles of Database Systems (2nd ed.)*, Computer Science Press, Rockville, MD, 1982

[WNP87] Weikum, G., Neumann, B., Paul, H.-B.: *Concept and Realization of a Set-Oriented Page-Layer for Efficient Access to Complex Objects*, Proc. GI Conf. Database Systems for Office Automation, Engineering, and Scientific Applications, Darmstadt, 1987

[WY76] Wong, E., Youssefi, K.: *Decomposition—A Strategy for Query Processing*, ACM TODS (1), 1976