The DASDBS Project
Objectives, experiences, and future prospects

RC. 89.-12

ETH

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Informationssysteme

Hans J. Schek
H.-Bernhard Paul
Marc H.Scholl
Gerhard Weikum

The DASDBS Project: Objectives, Experiences, and Future Prospects

November 1989

118

90.01.18
[illegible]

To appear in "IEEE on Transactions on Knowledge and Data Engineering" Vol. 2, No. 1, March 1990, Special Issue on Prototype Systems

Author's address:

Institut für Informationssysteme
ETH-Zentrum
CH-8092 Zürich, Switzerland

e-mail: *<last name>*@inf.ethz.ch

H.-B. Paul is on leave from Mannesmann Kienzle GmbH, Villingen, West Germany

The work reported here has mostly been carried out at the Technical University of Darmstadt, West Germany

Abstract

This paper is a retrospective on the Darmstadt database system project, also known as DASDBS. The project aimed at providing data management support for advanced applications such as geoscientific information systems and office automation. Similar to the dichotomy of RSS and RDS in System R, we pursued a layered architectural approach: A storage management *kernel* serves as the lowest common denominator of the requirements of the various application classes, and a family of *application-oriented frontends* provides semantically richer functions on top of the kernel.
Here we discuss the lessons that we learned from building the DASDBS system, particularly the following issues: the role of nested relations, the experiences with using object buffers for coupling the system with the programming-language environment, and the learning process in implementing multi-level transactions.

1 Objectives, History, and Background

The story of the Darmstadt database system, also known as DASDBS, began in 1983, but quite some influence came from the early AIM project [14, 56] which started in 1978. During this period of time, advanced database applications such as computer-aided engineering, geoscientific information systems, and office automation evolved and posed a tremendous challenge to database system researchers and developers. Like many others, we were (and are still) convinced that building additional functions on top of a conventional database system was unlikely to provide the required performance and therefore not a viable solution. We were thus highly motivated to build a new advanced database system from scratch.

We realized that building a monolithic next-generation DBMS was actually the wrong way to go. Rather, because of the divergent requirements of the various classes of advanced applications, DASDBS was designed as a family of database systems that are tailored to individual application classes. The idea was to obtain a customized DBMS by running an application-specific frontend on top of a common kernel system. The kernel itself was meant to be the lowest common denominator of the data management functions needed by a large variety of application classes.

In early 1987, a complete (Pascal) prototype of the DASDBS kernel was running under the IBM operating system VM/SP. Since that time, the kernel was ported to C/Unix, a number of enhancements have been made to the kernel, and the focus of the project has gradually shifted to the application-oriented frontends. Moreover, we have realized that building the system was one important objective, learning how we should have built it was another equally important objective. In this paper, we critically review our original design decisions and discuss the lessons that we have learned in the course of the project. Our retrospection covers both what we consider as achievements and the unexpected pitfalls that we encountered. The purpose of the discussion is to provide insights into what services a database kernel system should provide, how they should be implemented, and how they should fit into the overall architecture of a next generation data management environment.

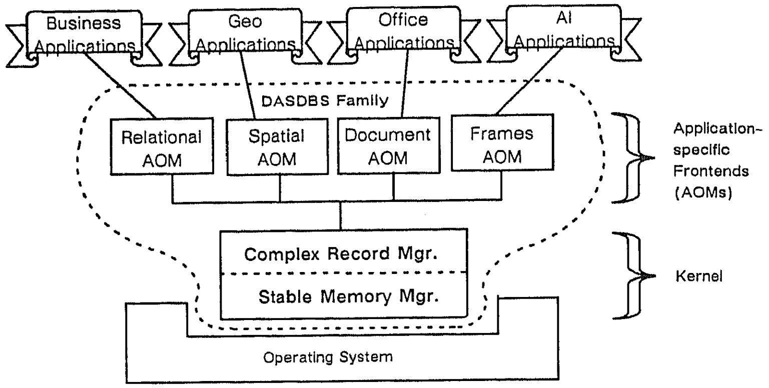


Fig.1: DASDBS Architecture

The remainder of the paper is organized as follows. Section 2 briefly discusses the architecture of the DASDBS family and the two roles of the nested relational (NF^2–) model[1]. In the main part of the paper (Section 3), we review the design and key concepts of the DASDBS kernel, namely complex-object support, set-orientation, and multi-level transaction management. Section 4 discusses experiences from developing a frontend to the kernel, namely the need for extensibility. Preliminary performance results are presented in Section 5. Finally, we conclude with an outlook on how the experience gained in the DASDBS project will influence ongoing and future projects.

2 Overall Architecture and the Role of Nested (NF^2) Relations

2.1 Overall Architecture of DASDBS

DASDBS is designed as a family of database systems as shown in Figure 1. A storage management *kernel* serves as the lowest common denominator of the requirements of the various application classes, and a family of *application-oriented frontends* provides semantically richer functions on top of the kernel.

- The *DASDBS kernel* is a storage manager that provides only basic data management functions. In particular, data independence, access optimization, and integrity checking are not included. A simple but efficient query processing strategy and storage structures for NF^2 relations are implemented in the kernel. Thus, the kernel is already a low-level platform for building high-performance applications that would otherwise bypass a full-fledged DBMS. In this sense, the kernel can also be viewed as an extended file system.
- An *Application(-oriented) Object Manager* (AOM) together with the kernel implements a full-fledged DBMS tailored to an individual application class, such as business transactions, geographical information systems, office automation, or expert system support. The services provided in an AOM include an end-user and application programmer interface,

1. NF^2 = Non-First-Normal-Form ... Relations with relation-valued attributes (nested relations)

query optimization, and access path management. For business applications based on flat relations, the AOM corresponds to the RDS of System R. An application using a DASDBS system will typically interface to its AOM, not to the kernel. The data models offered by distinct AOMs may be different from each other and from the NF^2 model of the kernel.

The rest of Section 2 describes the two roles of nested relations in the DASDBS project. The first role is that of an extensible "starter" data model used as a common backbone of the DASDBS frontends. The second role is that of a formal description of the internal structures managed by the DASDBS kernel, that is, the NF^2 model as the interface of the storage manager. The main part of the paper (Section 3) will then concentrate on the implementation of this kernel.

2.2 Nested Relations as a Starter Data Model for DASDBS Frontends

Based on our analysis of geographical applications [59] and office information systems [51] we were faced the problem of identifying data models for both areas. Soon it became clear that this would hardly be a single one for both. Instead of developing two specific models, however, we followed the idea of an *extensible "starter" data model*. The starter model would contain the essential ingredients of semantic data models. Application-specific new concepts could be added on demand (extensibility). As sketched in the following, we discovered that an extension of the NF^2 model allowing recursive schema definitions could play this role.

The vanilla NF^2 model as defined in [57] deals with hierarchies. In the same way as the flat relational model brings flat files up to a higher level of abstraction, the NF^2 model does the same for hierarchical files. It allows applying relational algebra to hierarchical data structures. This way, it combines the relational high-level operations with a richer data structure. The advantage is that by the declarative style of operations we preserve the potential for query optimization, even in the generalized setting of hierarchically nested data structures. This is clearly an achievement. Considering the mapping of application objects to a data model, though, nested relations need the following enhancements:

- natural support for many-to-many relationships (shared subobjects) and recursive relationships,
- generalization (ISA-relationship) with inheritance,
- a means for extending the typically very limited, fixed set of primitive data types.

The absence of these have been inherited from the classical relational model. While the first two aspects are essential constructs of semantic data modeling, the third one relates to integrating two different type systems. Particularly, the integration between (geometric) application objects represented in a programming language and their representation using the data structures of the data model is usually not smooth enough. Moreover, no notion of complex objects that is solely based on ERM extensions or nested relations or whatever semantic data model would support geographic or geometric data adequately, simply because of the lack of the "right" operations on the "right" data structures. Highly specialized data organizations and algorithms have been developed in such applications. Basically, there are two ways out:

One is the simple formula "data model = application language", that is, persistent programming, the other one – which we follow – can be characterized as an *extensible data model.*

The extensible data model approach is based on a "starter" data model which can be extended by new types and operations. Extensibility issues are further discussed in Section 4. Obviously the crucial question now was to identify the concepts to be included in the starter model. Conceptual studies [35, 41, 52] showed some strong connections between nested relations and semantic data models and AI frames. Set–valued relationships between objects is a common feature that can be modeled as subrelations. Unlike nested relations, however, these models deal with many–to–many relationship in a symmetrical way. General network structures can be represented using recursive schema definitions. We extended the concept of nested relations accordingly, such that the starter data model includes shared objects and recursive relationships via *recursively nested relations*.

These allow application of the relational algebra even on network structures and not only on hierarchies. For instance, the well-known supplier–parts, parts explosion scenario may be modeled by the recursive object types PARTTYPE and SUPPLIERTYPE as follows:

```
type PARTTYPE = object                              type SUPPLIERTYPE = object
                PNO: integer,                                         SNO: integer,
                DESCR: string,                                        NAME: string,
                WEIGHT: real,                                         LOC: string,
                SUBPARTS: set of PARTTYPE,                            PRODUCES: set of PARTTYPE,
                SUPPLIED_BY: set of SUPPLIERTYPE                      end.
                end.
class PARTS: set of PARTTYPE.                       class SUPPLIERS: set of SUPPLIERTYPE.
```

Notice that this definition covers both, a network–shaped relationship between parts and suppliers (the sharing aspect) and a (directly) recursive type definition (parts–subparts). The class PARTS can be viewed as a *recursively nested relation*, where for every PARTS "tuple" p we find a "subrelation" SUBPARTS(p) comprising as subobjects all (direct) component parts of p, and a second "subrelation" SUPPLIED_BY(p) containing the set of all suppliers producing p. Using such a nested relational view of the object class we can easily apply nested relational operations. Indeed, it turned out that we can use the retrieval operations of our nested algebra (or any other nested relational language) with only minor changes [55, 58]. For each particular query, we can dynamically choose the most appropriate hierarchical view: query Q1 "sees" parts as a subrelation of suppliers, whereas query Q2 takes the opposite view:

```
Q1 = project [DESCR, project [NAME] (SUPPLIED_BY) ]     Q2 = project [NAME, project [DESCR] (PRODUCES) ]
    ( select [WEIGHT > 1000] (PARTS) )                       ( select [LOC = 'Zurich'] (SUPPLIERS) )
```

For updates, we have to keep track of object sharing and thus need some additional functionality [63]. But still, the nested relational concept works for updates, too. We can, for instance, modify a part by inserting a (new or preexisting) part as a component into the SUBPARTS "subrelation" or delete parts based on conditions on their subparts.

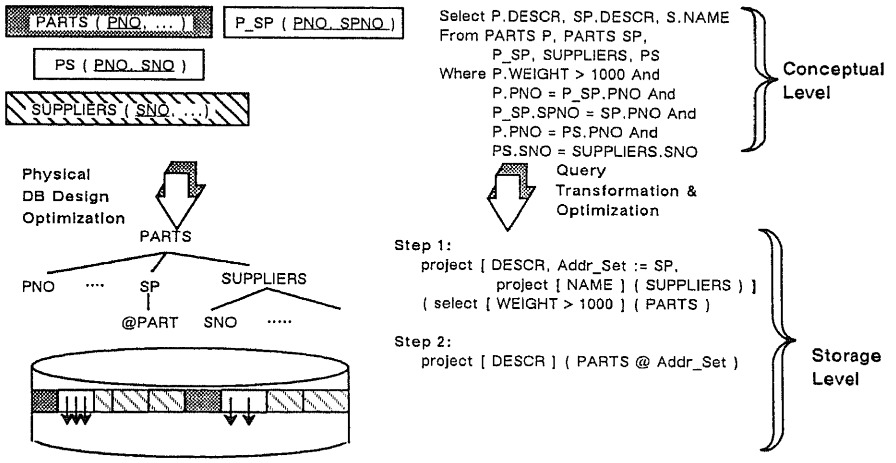
Interestingly, the EXTRA model proposed in the Exodus project [9] seems to play a similar role as a starter model; however, it contains explicit pointers as in the LauRel [34] proposal, in order to avoid recursive schema definitions. For the treatment of generalizations in our starter model, the reader is referred to [63].

2.3 Nested Relations as a Data Model for Storage Structures

NF^2 relations as a formal description of *storage structures* were an important concept in the DASDBS project. In this subsection we describe this role of nested relations.

Physical Database Design and Impacts on Optimization

We were strongly convinced that a new system should not only have a more powerful data model at its interface to the applications, but also a clear and well understood interface to its storage manager. The main motivation for using NF^2 relations as the storage-level "data model" was on optimization: If both the conceptual *and* the internal model can be rigorously described in a formal framework, it is possible to describe the mapping between them algebraically. This allows us to apply and generalize techniques of algebraic query optimization towards the physical level. As indexes can also be described by NF^2 structures, we hoped to include access path selection in the framework of algebraic optimization. The NF^2 model has been extended to cope with addresses (stored object identifiers) and address materialization was included in order to support direct access to stored objects given their addresses. It was shown in [17, 18, 62] that a large variety of known storage structures for physical database design is encompassed by NF^2 data structures. Thus, using the NF^2 model at the kernel level was supposed to simplify the physical database design and the related problem of query transformation and optimization.



```
Select P.DESCR, SP.DESCR, S.NAME
From PARTS P, PARTS SP,
     P_SP, SUPPLIERS, PS
Where P.WEIGHT > 1000 And
     P.PNO = P_SP.PNO And
     P_SP.SPNO = SP.PNO And
     P.PNO = PS.PNO And
     PS.SNO = SUPPLIERS.SNO

Step 1:
  project [ DESCR, Addr_Set := SP,
          project [ NAME ] ( SUPPLIERS ) ]
  ( select [ WEIGHT > 1000 ] ( PARTS )

Step 2:
  project [ DESCR ] ( PARTS @ Addr_Set )
```

Fig. 2: NF^2-Based Optimization

Figure 2 shows an example, where we reconsider the supplier–parts scenario introduced in Section 2.2, which is now conceptually represented in flat relations. Assume that the transaction load is such that a good physical design materializes the join between suppliers and parts by storing SUPPLIERS as a subrelation of PARTS. Notice that this storage scheme introduces redundancy so as to increase retrieval efficiency at the cost of extra overhead for updates. Furthermore, we also incorporated a "link set" for the part–subpart relationship, that is, subrelation SP of PARTS contains the storage addresses of all direct subparts (we used the notation "@PART" to denote addresses of PARTS–tuples). Such link sets (or pointer arrays) can be viewed as a special version of join indices. The advantages in terms of execution costs for certain queries, namely those involving the materialized joins, are obvious: in our example, asking for supplier names and direct component parts' descriptions of heavy parts, we have to compute just one "join" internally instead of four on the conceptual level. The internal "join" is particularly cheap since it just follows the addresses found in the "link set" by a direct access operation ("...(PARTS@Addr_Set)...").

The expectation that physical database design optimization is much simplified by the use of the well–defined NF^2 data structures and utilizing NF^2 algebra for the description of the relationship between user–level data and storage–level data structures was not yet satisfied. Although nested relations do in fact provide a uniform framework for the description of physical design alternatives, the elegance of the NF^2 model has not (yet) contributed to an elegant solution for the actual optimization problem, that is, choosing the best among these alternatives. While a first prototype of a physical design optimizer is available [49], only a few heuristics for the optimization problem could be exploited so far.

The idea of broadening the scope of algebraic query optimization was successful to a certain extent: In concentrating on the special case of a relational frontend supported by the kernel, that is, given the choice to hierarchically materialize frequent joins on the internal level, we developed a theory for algebraic query optimization, that is, a set of transformation rules. These rules enable the query optimizer to eliminate unnecessary joins [60]. A corresponding prototype has been built as a testbed, but is not integrated with the DASDBS system.

The expectation that access path selection could be included into the algebraic optimization step, too, has not been satisfied. Generally, in the case of redundant storage schemes – access paths are a particular kind of redundancy – pure algebraic optimization, that is, techniques involving no information on the state of the database, cannot succeed. An algebraic framework for the description of possible execution strategies can be used if physical operators are added. However, algebraic optimization heuristics like ordering of operations have to be complemented with cost estimation for the alternative access plans, as in Starburst [39] or Exodus [24].

Functionality of the DASDBS Kernel

The concept of using a formal model as the interface of the storage subsystem also guided the choice of the operational part of the DASDBS kernel interface. The idea was to provide only basic functions that do not incur much overhead, yet are powerful enough so that semantically richer application–oriented functions could be implemented with only a few kernel calls. What we had in mind was a storage system similar to the Research Storage System (RSS) of

System R. Unlike RSS, however, we wanted to support efficient access to complex objects, and we strove for more flexibility with regard to index and transaction management. The solution that we finally came up with was to implement a subset of the nested relational algebra within the kernel, and to exclude index management from the kernel.

The problem was to decide on the set of physical operators used to implement the operations of the algebra. Following the dichotomy of RDS and RSS in System R or MVQP and OVQP in Ingres, we planned to build only the simpler operations of the nested algebra into the kernel. Complex operations were to be provided in the frontends, if needed in the specific application class. While single–table vs. multi–table operations served as the coarse distinction between RSS and RDS operations, nested relations required a more detailed analysis.

The notion of *"single pass" queries* was introduced in [54] to encompass all expressions of the nested algebra that can be computed in a single hierarchical scan over the data (linear–time, constant–space processible). The idea was to build these operations into the kernel to be computed "on the fly" while scanning data. A detailed analysis of single pass queries is contained in [61]. Roughly, we can use nested selections and nested projections, but not in all of their possible combinations. Set comparisons have to be excluded from selections unless one of the operands is a constant. While the Verso project even aimed at realizing such simple operations in hardware [2], we considered them an appropriate set of low–level operations for the kernel.

In the context of our investigations on building a (flat) relational frontend to the kernel, we found another justification for exactly this class of kernel operations. If the flat relations at the user interface of this frontend are mapped to storage structures using materialization of frequently needed joins, any select–project–join query involving only materialized joins can be mapped to *one* single pass expression [60].

An efficient evaluation strategy for single pass queries is implemented in the DASDBS kernel [50, 49]. The main objective here was to detect non–qualifying tuples as soon as possible. Due to the nesting of conditions deep in the hierarchy, this was not a trivial task. The formal definition of the operator semantics and their algebraic properties, such as commutativity, helped in designing the optimal evaluation strategy.

For *update operations*, we took a more pragmatic approach. A nested relational language with update capabilities includes a wide variety of combined update, insert, delete, and retrieval functions [61]. For the kernel, retrieval operations inside update functions ("Update ... where ...") have been omitted, for they have to be executed in two steps anyway. Since access paths are not managed within the kernel, we have to deliver (parts of) the updated tuples to the frontend during updates to enable access path maintenance. Therefore, we can implement such 'descriptive update operations' by two subsequent kernel operations without (much) loss of performance [50].

As for *index management*, we assumed that no single access method would be suitable for all application domains. Rather, a variety of customized techniques would be best suited in each particular case. Thus, the "lowest common denominator" paradigm together with the need for explicit control over address lists in the query processors of the frontends (in order to do intersections or unions for combined predicates) motivated our decision for not building any access paths into the kernel. Consequently, all indexes in DASDBS are secondary, that is, the leaves of a B^+ tree contain address lists rather than the actual data items. Further anticipated advantages of this architecture were the uniform treatment of all kinds of data throughout the system: primary data and indexes are all handled by a single data manager, the kernel, in a uniform format, as nested relational tuples. Transaction management on indexes would thus be no different from the one on 'primary' data (see Section 3.4). Concerning the performance of this solution, our argument was the following: I/O costs were considered to be the main bottleneck. In terms of I/O costs, however, there is no difference between implementing a B–tree node on a single page or as a complex record which fits into one page. There is one I/O per node in both cases.

As a consequence of this decision, the kernel delivers addresses of stored objects on demand and has the additional functionality of materializing references (following pointers). In fact, any retrieval request can be accompanied by a set of addresses, thus restricting the set of inspected tuples to those addressed in that set. Update operations are always specified together with such an address list. An example for the use of addresses in a sequence of kernel calls is shown in Figure 2 above, where a join supported by a "link set", that is, a subrelation containing addresses of related tuples, is computed by two kernel calls. The second call (step 2) is a direct access operation with the set of addresses (obtained in step 1) as its input.

The decision not to include access paths into the kernel has never come without second thoughts. In fact, objections against a data manager without any indexing support, that is, with a single storage method, have always been around and traded off against the expected benefits. In retrospect, we are leaning towards integrating a number of access techniques as alternative storage structures into the kernel. To retain the flexibility with respect to evaluating combined predicates, however, we would deliver addresses from indexes to the frontend upon demand.

3 The DASDBS Kernel

3.1 Key Concepts and Overall Design

Because of the kernel's central role in the architecture of the DASDBS family, it was clear from the beginning that performance was very crucial. Our key concepts for achieving high performance have been the following:

- complex–object support
- set–orientation
- multi–level transaction management

By *complex–object support* we mean that the kernel should have knowledge about the structural aspects of complex objects, to allow physical clustering of related objects on disk. Because structurally related objects are often accessed together, structure–driven clustering can indeed improve disk I/O efficiency significantly. Complex objects directly serve as the physical storage unit, rather than being decomposed into smaller units. In an application–oriented frontend, we can therefore achieve any desired schema–driven clustering scheme by choosing the appropriate nested relational schema as the kernel representation of complex objects [62]. Such a storage scheme may, of course, be quite different from the structures seen at the interface of the frontend.

Set–orientation was conceived as a complement to the concept of complex objects. As complex objects are constituted by sets of subobjects, we anticipated a potential bottleneck in the repeated invocation of certain services while iterating over such a set. To avoid this bottleneck, most functions in the DASDBS kernel deal with sets of objects at a time. The performance advantages that we expected were optimized data transfers, especially between disk and memory [76], and a minimum amount of crossing interfaces during the processing of a set of objects.

Advanced applications posed new challenges also in the area of transaction management. We wanted to exploit the semantics of applications, so as to cope well with long–lived transactions that access complex objects. Moreover, the architecture of the DASDBS family suggested that transaction management should be provided already in the kernel, but should be extensible in the application–oriented layers. By carefully studying details and tricks of conventional high–performance transaction managers, we finally came up with a general framework that is known as *multi–level transaction management* or layered transactions [74, 6].

3.2 Complex–Object Support

3.2.1 Rationale

In DASDBS, physical clustering is expressed by defining the appropriate nested relations at the kernel interface. Consequently, nested relational tuples have to be implemented as complex records, as opposed to being decomposed into flat records and spread out over the disk. The principal goal is to store a complex record as a linearized consecutive string preserving the hierarchical structure (e.g., by adopting a depth–first order). This string (as soon as it is larger than one page) is to be mapped to a minimum number of (ideally) contiguous disk blocks, the *storage cluster* associated with the record. Our design of the storage layout of complex records pursued the following key objectives:

- Access to and manipulation of complex records *as a whole* as well as *any components thereof* has to be supported efficiently.
- The performance penalty on storing just ordinary flat tuples has to be negligible.

The first objective reflects the need for the kernel's knowledge about *structured* objects rather than long containers, the second objective reflects our intention of providing a platform suited for *all* kinds of applications, including traditional relational ones.

From these general guidelines we derived several consequences. Particularly, since complex records may be of virtually unlimited size, we had to design flexible internal structures dealing with potentially long elements on all levels. In order to achieve the second objective, however, only little interpretation overhead could be tolerated for the outermost level of the hierarchy. Particularly, short tuples have to share pages. Special cases of substructures, like subrelations with fixed-length subtuples, should be implemented efficiently in addition to the general, fully flexible case. For example, address lists or polygons of points could largely benefit from such specialized high-performance implementations.

To attack the I/O bottleneck when retrieving complex objects from disk, the storage structure had to support that the set of disk blocks constituting a storage cluster can be determined efficiently, so as to allow exploiting the set-oriented I/O interface (see Section 3.3.2). If a kernel request does not need all of a complex record, the relevant subset of the cluster pages should be identified, so as to avoid unnecessary I/Os and buffer contention.

The addressing scheme for nested tuples had to provide references to the objects as a whole and to any components or subsets of these. As usual, addresses should be stable with regard to reorganization. In our case of large storage objects, this should be true for inter-object as well as intra-object relocations. As physical contiguity of the cluster blocks may eventually be destroyed by the dynamics of the database, the storage structure had to support cheap reorganization techniques for reestablishing this clustering.

3.2.2 Achievements

Our solution for the storage structures of complex records does indeed meet most of the described requirements and provides disk efficiency [17, 18]. The key characteristics of the kernel storage structures are illustrated in Figure 3. Each *Storage Cluster* is considered as its own address space, that is, all intra-object references are relative to it. The first page of the storage cluster, called the *Root Page*, includes the *Page List*, that is, a list of physical page numbers constituting this address space. Thus, relocating some of the blocks of a cluster incurs minimal reorganization overhead, for none of the internal references except the page list have to be updated. Within each cluster, structural descriptions are separated from actual data. The structural information is concentrated in an *Object Directory* at the beginning of the cluster, that is, on the first page(s). An Object Directory mirrors the hierarchical structure of the stored NF^2 tuple, containing references to subtuples. In the example of Figure 3, these references point to the atomic-attribute fragment of the NF^2 tuple X, to the subtuples X.1, X.2, X.3, and to the subsubtuples X.2.1 and X.2.2 of subtuple X.2.

To retrieve the whole cluster from disk, we have to read in its root page, extract the page list, and read in the rest of the cluster in a single set-oriented I/O. For operations needing only parts of the complex record, the object directory and the query are used to compute the necessary subset of cluster pages before issuing the second I/O request. In either case, for huge clusters an intermediate (set-oriented) I/O request may result from the page list (or the object directory) not fitting into a single page.

The addressing scheme adopted for DASDBS is basically a hierarchical one. That is, addresses of subobjects are obtained by concatenating a specific suffix to the parent object's address.

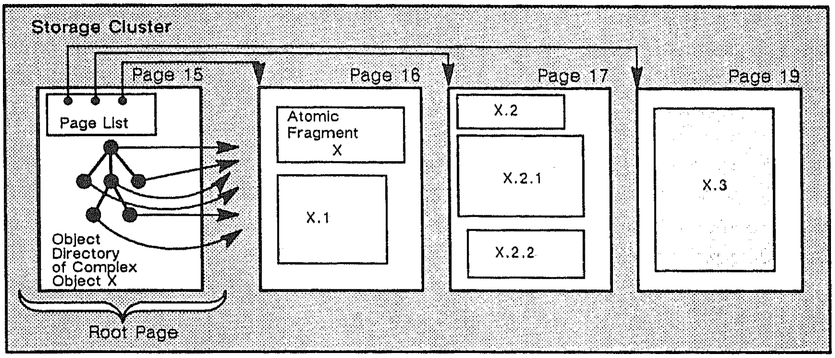


Fig. 3: Complex Object Clustering

Root objects are referenced using the well-known TID technique, thus achieving stability with respect to intra– and inter–page relocation and fast access (1–2 I/Os). Components are addressed using an ordinal numbering scheme. For instance, a third–level subtuple address might be (*tid,i,j,k,l*), referencing in the object addressed by *tid* the *j*th subtuple in the *i*th subrelation, and within this the *l*th subsubtuple of the *k*th subsubrelation. The name HITID (hierarchical TID) has been coined for this method, as the ordinal numbers are transformed into intra–cluster pointers using the object directory. All intra–cluster pointers are in turn implemented as TIDs, but with shorter page numbers relative to the cluster's address space.

The advantage of a *hierarchical* addressing scheme is that subobject references include the full path from the root object. Thus we can do smart address calculations in the case of indexes on subobjects. For instance, consider a query such as "Are there any mathematicians in the CS department?". Assuming an index on the profession attribute of the employee subrelation of departments, we can determine all candidate department addresses using the "mathematician" entry of the index. All we have to do is to strip off the employee suffix of the addresses in the list. Such capabilities would not be available in a *direct* addressing scheme where each subobject has its own, independent address. Moreover, if several subobjects referenced in an address list share a common parent (or even path), we can "factor out" the prefixes and use a nested relational representation of address sets. This looked particularly promising because access paths in DASDBS are implemented on top of the kernel, such that we could in fact implement address lists as nested subrelations.

On the other hand, the drawback of hierarchical addressing compared to direct addressing is that 1) the complete path from the root has to be traversed in order to access the addressed subobject, and 2) the length of an address depends on the level of the referenced object. We tried to avoid the first disadvantage by the layout of the object structure, namely by concentrating the object directory on the root page(s), thus avoiding additional I/Os for the traversal. The efficient manipulation of address lists becomes slightly more complicated if the address length is not constant. However, addresses pointing to the same level in the hierarchy would still all have the same length. Generally, we expected the benefits of the additional address manipulation features to outweigh this drawback.

3.2.3 Pitfalls and Lessons Learned

A general observation is that we underestimated the whole effort. Initially, we planned to have a first "throw away" prototype of the kernel within rather short terms. However, it took us much longer, such that many adhoc quick–and–dirty solutions turned into "features", simply as time went by. The turnaround interval for the design–implementation cycle had become too long as to quickly integrate new ideas or solutions into the running system. As a result, nothing was really thrown away, rather only minor reimplementations have been carried out.

As for performance, it turned out that the interpretation of the storage structures is more expensive than expected. Our first implementation language was Pascal, which prevented us from coding the internal structures into programming language constructs. For example, arrays of pointers would have led to *compiled* code for the access to the referenced elements, while our kernel code *interprets* all of these substructures. This is particularly costly, since it is invoked in the innermost processing loop of all kernel operations. The lesson we had to learn is that while we concentrated on optimizing the I/O efficiency of the kernel, we almost ignored the CPU complexity of the kernel operations. While it may be true that I/O performance is most crucial for simple operations such as primary key access in a relational DBMS, in a complex object database system computation costs are an important factor in the overall efficiency.

The main implementation lesson is that whenever we had several alternatives, we implemented the most general case first. This was not really a goal, it just turned out that way. For instance, we first implemented the case of variable–length attributes, and so on. Of course, such flexible structures incur more interpretation overhead than special cases. The specialized high–performance variants, however, are badly needed in order to provide efficient manipulation of address lists in access paths. Some of these important special cases have not been implemented yet, including subrelations having only atomic attributes that are all of fixed length. Especially this feature would have been useful for implementing access paths with address lists as nested relations. In a way, frustration about the initial performance was preprogrammed.

3.3 Set–Orientation and Buffering

3.3.1 Rationale

Set–orientation in the DASDBS architecture aimed at alleviating two sorts of performance costs:

- Whenever *object copying* is inevitable between two layers of the system, it should be optimized by batching together multiple related copy requests. For example, a polygon that is composed of thousands of points should be copied in a single step rather than handling each point separately.

- Whenever *crossing an interface* is relatively expensive compared to the amount of work that is to be performed, the amount of interface–crossing should be minimized by making requests set–oriented.

Performance problems with excessive interface–crossing encountered in building advanced applications on top of conventional database systems were reported, for example, in [26] and [27]. It was found that both the I/O costs of fetching a large complex object page by page and the CPU costs of copying a large set of flat records that constitute a complex object from the DBMS buffer into the application address space accounted for the disastrous performance. Even conventional applications often have the need for set–oriented disk access or "bulk I/O", e.g., for sequential scans. In the context of complex objects, these problems are aggravated.

Based on the above observations, we decided to provide set–orientation at three main interfaces of the DASDBS architecture, namely at the disk I/O interface, at the page buffer interface, and at the interface between the kernel and the application–oriented layer. Set–oriented disk I/O aims at increasing the effective speed of the disk, by exploiting the physical clustering of large objects, e.g., reading any number of pages that are located on the same track in a single revolution of the disk. The set–orientation at the page buffer interface primarily serves to pass set–oriented page requests to the I/O manager, but also to reduce the overhead incurred by the buffer manager's fix and unfix routines.

At the kernel interface, the copying of complex objects was deemed unavoidable, because we did not want to have a possibly large set of pages fixed in the global buffer pool for the unpredictable duration of the processing in the application–oriented frontend. Note that similar considerations, in the context of conventional applications, have led to two different fixing policies in DB2 [10], namely to keep a page fixed during the processing of all records stored in the page, or to fix and unfix the same page repeatedly when the buffer steal rate is high. We introduced two sorts of buffers, as shown in Figure 4.

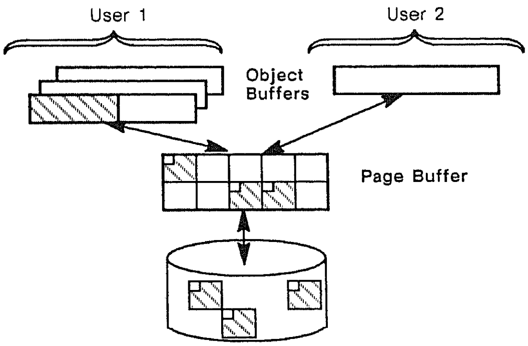


Fig.4: Global Page Buffer and User–Specific Object Buffers

- The *page buffer* contains pages which are transferred between memory and disk. The page buffer is usually memory–resident and shared among all "users".
- *Object buffers* contain sets of complex records of a single relation. Object buffers are not shared among "users" and live only temporarily. Another difference to the page buffer is that object buffers are usually subject to virtual memory paging. Query results (or portions thereof) are transferred into an object buffer. Then, the application–oriented algorithms can process the retrieved objects without further interaction with the kernel. Insertions and updates are performed first on the object buffer and propagated (back) to the kernel, again in a set–oriented mode.

Object buffers have also been suggested for improving the object hit ratio in memory (as opposed to the page hit ratio) [31], and as data transport containers in a server–workstation environment [32]. Furthermore, copying is often deemed necessary for protection, but this did not really play a role in our considerations.

In the DASDBS project as well as in various other projects (e.g. [20, 28, 22]), copying complex objects into an object buffer was desirable, to some extent, also for the purpose of *object translation*. This means that objects should be converted from their representation in disk pages into a format that is more convenient for the application–oriented processing. For example, page headers can be stripped off, and large regularly structured objects, such as matrices, that reside in non–contiguous buffer pages can be moved to contiguous space so as to enable processing the object like an array. In the case of complex objects that contain intra–object references, object translation raises additional problems that are discussed in Section 3.3.3.

3.3.2 Page–Set–Orientation

Set–orientation has proven to be an extremely advantageous concept at the page buffer interface and especially at the disk I/O interface. Our solution combines the following benefits:

- Large data requests can be performed in a single disk I/O, thus saving disk arm seeks and minimizing rotational delays. In addition, the CPU costs of setting up I/Os and handling their completion are largely reduced.
- In contrast to approaches that simply use bigger page sizes, there are no extra costs for conventional single–page accesses.
- In contrast to approaches that use several page sizes, we do not have any fragmentation problems in the still page–structured buffer pool, for the pages of a set of contiguous disk pages need not necessarily be contiguous in memory. To achieve this advantage, the disk controller or the I/O bus have to support "scatter/gather I/O" . This property is offered by many of the available controllers, but rarely exploited in most operating systems.

The only problem that we faced with regard to set–oriented I/O is that such a facility is not provided already in the operating system, where it would naturally belong. Unfortunately, the commercially prevalent operating systems do not make all the capabilities of their low–level I/O systems available at a reasonably high system–call interface, such as the (block mode) file system. A fairly detailed description of our implementation of set-oriented disk access and the results of a performance study are presented in [76]. A particularly interesting result is that set-oriented I/O not only improves disk throughput drastically, but wins also with regard to disk access response time in multi-user mode.

Open issues that need further investigation are 1) how to handle set-oriented page buffer requests that are satisfiable only partially in situations when the available memory is scarce, and 2) finding the right balance of set-oriented single-disk I/O vs. parallel multi-disk I/O, that is, clustering vs. declustering [38].

3.3.3 Object-Set-Orientation

The set-orientation at the complex record interface has turned out to be a hard issue, and we have not yet found a fully satisfying solution. The different approaches that we tried out are discussed in this subsection.

Overall, our object buffer approach was conceived as a generalization of the concept of "portals" that has been proposed in [66] as an enhanced way of coupling conventional, that is, relational, database systems with the programming-language environment. Portals allow fetching a set of tuples into a host-language variable of the type "array of record". Thus, portals preserve the set-orientation of the relational model much better than the tuple-at-a-time logic of the SQL cursor concept. Unfortunately, none of the widely used programming languages provides a type that can be used as the counterpart of a nested relational schema in a similarly straightforward way. So what we basically did was to implement a new host-language type "(nested) set of record". Operations on this type include hierarchical navigation functions for traversing complex objects, but we foresaw also a use of performing NF^2 algebra operations against such in-memory objects. In addition, we wanted to implement the type "set of record" in such a way that, in certain cases, compiled application(-oriented) code could directly operate on the type representation rather than invoking object buffer functions. For example, a polygon of points would ideally be processible as efficient as though it were an array of fixed-length items.

Object Buffer Implementation

In the design of the object buffer implementation, we considered two fundamental sorts of performance costs:

- On the one hand, the *copying* and *translation* of complex objects from the page buffer into the object buffer (and vice versa) should be very efficient.
- On the other hand, we strove also for efficiency and ease of use with regard to the application(-oriented) *processing* of the objects in an object buffer.

Investigating implementation alternatives of an object buffer basically amounted to trading off these two performance goals. In the course of the DASDBS project, we implemented three different object buffer formats. The properties of these formats are coarsely summarized in Figure 5 and discussed in more detail in the following.

	Cost of Object Copying & Translation	Cost of Object Processing	Cost of Implementing NF2 Algebra Operations on Object Buffers	Compiled Access to Arrays
Heap-oriented format	high	low	high	not supported
Bytestring format	high	high	high	possible
Disk-oriented format	low	high	low	not supported

Fig.5: Comparison of Alternative Object Buffer Formats

Heap–oriented format:
In this format, variable–length pointer–arrays are used for representing sets of objects, that is, tuple–sets or subrelations. The referenced objects are tuples or subtuples, each of which in turn contains the values of its atomic attributes and pointers (to pointer–arrays at the next deeper nesting level) for the relation–valued attributes. All these structures are dynamically allocated while an object buffer is being filled, and virtual memory addresses are used as pointers.
The main advantage of this approach is that complex objects can be traversed very efficiently, once they are in an object buffer.
The flip side of the coin is that moving objects between the page buffer and an object buffer involves fairly high object translation costs. In addition, dynamically allocating many small chunks of memory is quite expensive.

- *Bytestring format:*
In this format, sets of complex objects are broken up into two components: 1) a *data bytestring* containing all atomic attribute values in a linearized order derived from the hierarchical addresses of the involved tuples and subtuples, and 2) a *descriptor bytestring* that contains all the structural information like lengths of atomic attribute values, cardinalities of subrelations, and the offsets of certain substructures. The offsets are relative to the beginning of the data bytestring, if they point to atomic attributes, or relative to the descriptor bytestring if they point to length fields or subrelation descriptors. Because the amount of memory that is needed for these bytestrings is not known until the object buffer is completely filled, both bytestrings are actually mapped to a small number of dynamically allocated chunks of memory.
The main advantage of this approach is that the elements of a subrelation are stored contiguously (without being interspersed with descriptor information), at least within each underlying chunk of memory. So, in the important case of subrelations with fixed–length elements, such as vectors or matrices, the memory addresses of the elements can be computed efficiently. If the policy for allocating memory chunks is smart enough to store an entire subrelation contiguously (without wasting too much space), arrays can even be directly processed by the compiled application(-oriented) code as though they were programming–language variables.
The drawbacks of this format are again the object translation costs for moving objects be-

tween the page buffer and an object buffer, and the fairly high costs of updating variable-length objects in the object buffer. For example, inserting a new subtuple into a contiguously stored subrelation may incur a lot of data movement.

- *Disk–oriented format:*
  This third alternative for representing nested relations in memory is an attempt to avoid the disadvantages of the two others. An object buffer is organized in pages, and complex objects are mapped to these pages exactly like they are mapped to disk pages. Furthermore, the page size of an object buffer and the disk page size are the same.
  This approach allows moving *complete* complex objects, that is, objects that have been retrieved without projections or have been created in the object buffer with defined values for all attributes, between the page buffer and an object buffer with almost no interpretation. That is, the object translation costs are just the raw costs of byte copying in this important case. Another advantage is that, as the formats of object buffers and the page buffer are basically identical, the kernel's query processor can operate on object buffers as well, e.g., to further select objects from the result set of a previous query. In general, many Complex Record Manager submodules are directly reusable for managing object buffers.
  The drawback of this approach with only a single disk–oriented object format is that the processing of objects in an object buffer is penalized quite a bit. Traversing a complex object involves a lot of structure interpretation such as following the indirections of page–index slots.

In the course of the DASDBS project, our preference for one of these formats changed several times. The current implementation using the disk–oriented format has already been identified as a major bottleneck. We now favor the heap–oriented format over the other two formats, for it provides by far the best performance with regard to object processing [80] (see also Section 5.2). A reimplementation is near to completion. Nevertheless, the bottom line is that bridging the gap between the programming–language–oriented application view of data and the disk–oriented database–system view of data is still a big unsolved problem. This issue is further discussed in Section 4.

3.4 Multi–Level Transaction Management

3.4.1 Rationale

Our goals with regard to transaction management were twofold: On the one hand, we wanted to provide the DASDBS kernel with an efficient full–fledged transaction manager, so that higher application–oriented layers would not have to build their own transaction management. On the other hand, we wanted to exploit the semantics of application–oriented operations so as to enhance concurrency, that is, allow higher layers to influence the transaction management. Given these seemingly incompatible goals, we were faced with the following questions: How can we build a layered transaction management architecture that deals with multiple levels of abstraction? What is the correctness criterion for such an approach? What kind of concurrency control should be provided in the kernel? In particular, what concurrency control granularities should be supported? Would index locking be a performance killer, since index management was not intended to be a kernel component? And finally, what kind of crash recovery would the kernel have to provide so as to allow the application–oriented layers to extend the kernel's transaction facilities in a simple and modular way?

We initially addressed these questions by studying the details and tricks of various transaction managers implemented in commercial systems [15, 78], especially those of SQL/DS, which is the commercial version of System R. These systems hold "long locks" on tuples and index keys to guarantee serializability; "short" page locks are acquired during the execution of an RSS operation to make RSS operations appear as though they were indivisible actions [25]. Thus, RSS operations can be viewed as subtransactions that release their locks at their completion rather than passing them to their parent transaction as in a "conventional" nested transaction approach [45]. Because low–level locks are released prematurely, that is, before EOT, the name "open nested transaction" has been coined [70].

Being fascinated by the (largely unexploited) potential of the System R approach, we then tried to cast its principal ideas in a general framework. We found ourselves extending the classical serializability theory, and finally came up with a theoretical foundation of multi–level concurrency control [72] and a framework for designing and implementing layered transaction management strategies [75, 74, 6] (see also the parallel work of [5] and [47]). Based on the novel criterion of multi–level serializability, we developed a nested two–phase locking protocol that is characterized by the following rules:

1. Assume a layered system with n levels, that is, layer interfaces L0, ..., L(n–1), in bottom–up order. An Li action *a* performing an operation *f* on object *x* has to acquire an operation–specific *f* mode lock on *x* before being executed.
2. This Li–specific lock is released at the end of the L(i + 1) (trans) action *t* to which *a* belongs, together with all other Li locks acquired during the execution of *t*.

As a consequence of such a multi–level locking protocol, transaction undo requires compensating completed high–level (subtrans) actions by means of inverse actions. State–based undo at the page level, e.g., recreating a transaction's page before–images, is no longer feasible, because low–level locks are released prematurely, that is, before EOT. In general, state–based undo is not feasible even at higher levels if we want to enhance concurrency by allowing commutative update operations on the same object, such as incrementing a counter. Both transaction aborts and crash recovery have to take this into account.

3.4.2 Implementation

DASDBS transaction management is based on the developed multi–level transaction framework. The kernel includes a 2–level transaction manager that handles the page level and the level of complex record operations. In addition, a third level of concurrency control has been implemented for the relational frontend layer. This third level provides a simple form of predicate locking, namely for conjunctive predicates with atomic terms of the form < Attribute > = < Value > or < Attribute > = < Don't care >. So far, this predicate lock manager was used only within a simulation testbed, whereas the two lower levels of transaction management are fully integrated into the kernel. Figure 6 shows the overall architecture of our 3–level transaction management.

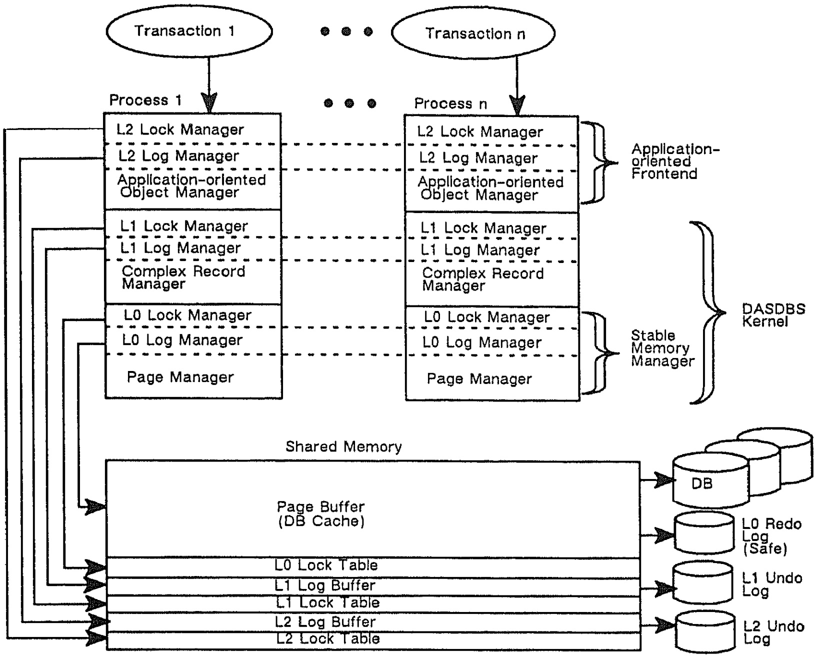


Fig.6: DASDBS Multi-Level Transaction Management

We have chosen a process–per–transaction approach for embedding DASDBS into the underlying operating system. Lock tables, log buffers, and the page buffer pool are located in shared memory. For the L0 level, page locks and segment locks are managed in a vanilla hash table. As the complex object layer deals with nested relations, a hierarchical locking scheme is used for the L1 level. This allows us to lock variable granularities ranging from an entire relation to a single attribute, subrelations being a special case. The hierarchical addresses of complex records are used as lock identifiers. For deadlock detection, the kernel employs an algorithm for computing partial transitive closures [29].

For correctness reasons, our multi–level concurrency control approach requires a multi–level algorithm also for recovery. To guarantee transaction persistence in the face of system crashes, we have chosen to provide persistence of committed L1 actions, that is, complex record operations that belong to completed transactions, at the page level L0. This property has led to the name "Stable Memory Manager" (SMM) for the page layer of DASDBS, and is achieved by logging page after–images. In fact, we have implemented the DB Cache Method [21] as L0 base recovery within the SMM. By writing after–image–sets atomically and employing an in–memory workspace concept, this method also guarantees atomicity of L1 (subtrans) actions. In the latest version of the kernel, we have further enhanced this basic approach so as to preserve the L1 action atomicity without having to force after–images at the end of each L1 (subtrans) action.

An important point of the L0 base recovery is that higher levels do not have to deal with redo. Higher levels have to record only sufficient undo information so as to be able to roll back incomplete (trans) actions of the next higher level. For example, the L1 log manager records information on inverse L1 actions for rolling back incomplete L2 actions (which are the transactions in the currently running 2–level scheme). Note that L1 action atomicity is a crucial prerequisite for this sort of higher–level undo.

3.4.3 Achievements

According to Einstein, "nothing is more practical than a good theory". We believe that the multi–level transaction framework and its underlying theory provide guidelines for designing, implementing, and analyzing practical methods.

The "Stable Memory" layer proved to be useful for higher layers. The fact that higher–level recovery does not have to care about redo simplifies implementing transaction management extensions on top of the kernel or even on the page–oriented base layer. The recently published ARIES recovery method [44] employs a similar paradigm, called "repeating history". Notice that this implies that updates of loser transactions may be redone before they are eventually undone by means of "logical", that is, higher–level actions. In fact, the DASDBS multi–level recovery approach was criticized because of this property. The similarity of ARIES and our multi–level approach is particularly remarkable, as ARIES has been designed for use in "industrial–grade" systems and therefore incorporates a lot of fine–tuning, whereas our approach aimed at reconciling acceptable performance with modularity and simplicity.

There is fairly large consensus that future operating systems will provide transaction management facilities. We believe that the concept of a "Stable Memory Manager" would be a reasonable candidate for integration into the OS. Even though OS transactions have been criticized for their inherent inability to exploit semantic knowledge [65,33], we are convinced that it is reasonable to use an OS transaction manager at least as the base layer of a multi–level transaction architecture (see also [73]). Such a facility would support atomic and persistent transactions as well as atomic actions, both appearing isolated from concurrent activities. It seems that this would factor out the base services needed by all higher levels. The multi–level transaction approach then is a way for exploiting the application semantics in the higher layers where it is defined.

3.4.4 Pending Issues

An issue that still needs and deserves further investigation is how intelligently the multi–level transaction approach can deal with index locks. In the following, we briefly discuss the 3–level approach that we are pursuing for the relational frontend of the DASDBS family. Consider an SQL–like operation such as INSERT INTO R ($K=k$, $A=a$, $B=b$) where R is a relation with a unique key attribute K and two further attributes A and B. Assume that there are indexes defined on both A and B. Then, the RSS of System R, for example, would lock the index entries, that is, key/pointer–list pairs, for the attribute values *a* and *b*, and keep these locks until EOT. Thus, a concurrent query such as SELECT * FROM R WHERE A=*a* AND B=*b'* would be blocked even if *b* and *b'* are different values. Further note that locks on the accessed index pages are acquired during the RSS operation into which the INSERT is translated. These page locks are released at the end of the INSERT, and are another source of potential data contention.

Now let us see how the DASDBS approach performs in this example. Since the kernel does not distinguish primary data objects vs. indexes, it would lock the kernel representation of the involved index entries for *a* and *b*. So in this regard, the kernel behaves pretty much like System R. However, from the kernel's point of view, the insertion of the new tuple and the updates of the two secondary indexes are three different operations which together constitute the INSERT at the next higher level L2. In this situation, if we are able to extend the 2–level transaction management of the kernel to a 3–level strategy and employ, at the additional L2 level, the simple form of predicate locking that we already implemented (see Section 3.4.2), then our approach will indeed allow more concurrency than the System R method. The locks on the index entries will be released at the end of the L2 INSERT operation, so that the above mentioned concurrent query would be blocked only for the duration of the L2 INSERT. Serializability is still guaranteed by the L2 lock on the predicate A=*a* and B =*b*. Finally, note that the sketched 3–level locking protocol releases also L0 locks on index pages earlier than the System R method, namely at the end of each single index operation.

3.4.5 Encountered Pitfalls and Solutions

The multi–level transaction approach gains enhanced concurrency at the cost of increased overhead. In particular, we have dealt with or plan to deal with excessive log I/O, and the CPU overhead of managing explicit subtransactions. In the following, we discuss our lessons on these issues.

Log I/O

Our original idea for implementing higher–level logging was to map log records through layers just like ordinary data objects. The advantage that we saw was that a high–level undo log record could be written to disk together with the corresponding lower–level redo information in a single atomic action, simply by including the writing of the high–level log record as an additional operation in the subtransaction for which the log record was created. This approach would have simplified achieving the idempotence of higher–level undo actions, because an undo log record would be read during a warmstart only if the updates of the corresponding subtransaction had actually reached stable storage. On the other hand, higher–level logging that in turn invokes lower–level logging was almost certain to cause unnecessary disk I/O. Because of this performance penalty, we dropped the idea rather soon, despite its apparent elegance.

We then decided to write higher–level log records directly, using a separate log file for each level (see Figure 6). The idempotence problem was solved by including the sequence numbers of the L1 (subtrans) actions in the corresponding higher-level log records. Since these sequence numbers are also stored in the L0 redo log, the recovery manager is able to determine exactly which higher-level actions need to be compensated. In addition, by writing undo log records for the inverse actions that are performed during a warmstart, the recovery manager keeps track of the progress that is made. This basically means that undo actions are treated like ordinary actions, so that undo actions may have to be undone in the case of repeated failures. While this may not exactly be desirable from a performance point of view [44], it is a correct recovery method and contributes to the simplicity of the warmstart procedure.

After having dropped the "layered logging" idea, the next trap we fell into was the approach of making completed L1 (subtrans) actions immediately persistent. Our implementation of the DB Cache method initially treated L1 (subtrans) actions, that is, kernel operations, as though they were transactions. That is, the after-images of such a subtransaction were forced to disk as soon as the subtransaction was completed. In addition, the higher-level undo log buffer had to be flushed to disk, according to the WAL rule. So there were at least 2 (set-oriented) I/Os for each kernel update operation, unless an additional feature like group commit were incorporated.

The immediate persistence of L1 (subtrans) actions was intriguing to us as a simple basis for adding intra-transaction savepoints, and it seemed to be unavoidable in our puristic view of a strictly layered transaction architecture. Moreover, it seemed to fit well with the evolving technology of safe RAM [12]. The main reason why we kept pursuing this approach, though, was that the DB Cache method as our L0 base recovery ensured also the atomicity of L1 (subtrans) actions (which in turn is a prerequisite for the L1 undo) in a simple yet reasonably efficient way. After all, throughput figures that we obtained from simulation experiments clearly indicated that the enhanced concurrency of a multi-level transaction approach is worth the extra logging costs.

We finally re-addressed the problem of excessive log I/O by developing an improved multi-level recovery algorithm in which the L1 (subtrans) actions are no longer made persistent immediately. Rather the writing of the after-images of a subtransaction is deferred until the EOT of the (top-level) transaction to which the subtransaction belongs, or until the atomicity of an L1 (subtrans) action would be violated. A detailed description of this algorithm and a discussion of its design rationale can be found in [77]. The improved algorithm is implemented in the latest version of the DASDBS kernel; a performance evaluation is underway.

CPU Overhead

While we are quite comfortable with our solution to the log I/O bottleneck, we have not yet really addressed the CPU overhead of multi-level transactions. In addition to the necessary bookkeeping for subtransactions, these CPU costs are mostly caused by the higher number of lock requests and releases, compared to a single-level locking protocol. Of course, conventional record locking suffers from this overhead, too, because of the necessary short-term page locks for isolating record operations. As a performance enhancement for conventional record locking, it is suggested in [44] that "latches" be used instead of "short locks" while a record operation accesses pages. The main differences are that latches are directly addressable, e.g., as an additional entry in the page header, rather than being stored in a hash table with dynamically allocated entries, and that latching does not check for deadlocks and disregards the (unlikely) possibility of starvation.

Following the direction of "light-weight" subtransactions, we can easily adopt the idea of storing lock control blocks in page headers. However, turning off deadlock detection for subtransactions cannot be so easily generalized to an advanced database system. Unlike conventional record operations that usually access one data page and a number of index pages in a well-defined order, complex operations on complex objects may access many pages, in a less predictable order. Hence, the L1 (subtrans) actions are susceptible to page-level deadlocks, so that simple latching is not feasible. We plan to investigate conditions for partially compiling a multi-level locking protocol to a latch-based protocol.

4 Application-Oriented Frontends and the Need for Extensibility

4.1 Rationale

Extensibility considerations became important in the DASDBS project when we were confronted with real-life spatial data from geosciences. These applications use a variety of different kinds of lines, regions, or surfaces, and, orthogonal to it, many different discrete representations of these continuous geometric objects exist. Supporting only basic primitives such as points, point sets, or polygon lines was not considered to be a satisfactory solution, for we wanted to avoid that every user of our system had to convert his favorite data structure and his geometric representation into ours in order to get database service for his application. Rather the system should take any geometric representation and store it into the database without (much) conversion and interpretation. Vice versa, upon retrieval from the database, data should be represented in the application format. We called this requirement the support for *externally defined types* (EDTs) [79], to emphasize the combination of the type systems of the application's implementation language and the database kernel. For the DBMS, an EDT is regarded as an abstract data type, only known by its functions. It is implemented outside the DBMS with the type system of the programming language used in the application layer. Thus, the DBMS has just the necessary knowledge, and the application retains its favorite data structure.

Basic predicates like an intersection test between a geometric object and a rectangle can be defined as EDT functions and make sense for any geometric kind and representation in two or three dimensions, that is, they are generic functions. If the type-specific implementations of these functions are available to the DBMS, any spatial index based on a rectangular subdivision of space could be used for all specific geometric representations. This very much corresponds to the idea of making use of a standard B-tree index whenever a type is to be supported for which an ordering function is defined [68, 64].

So, two objectives for extensibility were dominant in DASDBS:

- tight cooperation between DBMS and the application programming language (PL) via EDTs, seamless integration of PL types into the DBMS via the object buffer, and
- usability of standard grid files or other spatial access methods for many sorts of geometric objects.

While the second objective is a driving force also in other projects striving for extensibility like Genesis [3, 4], Postgres [67], Starburst [36], and Exodus [8, 7], the first objective seemed to have had less weight in these. We felt that the tight cooperation is equally important; Probe [23, 43] and AIM [37] seem to attack this problem, too. Both objectives will be discussed in more detail in the following.

4.2 EDT Support and Tight DBMS–PL Cooperation

The support of geographical objects demanded for a tight coupling of the application's programming language and the DBMS (attacking the "impedance mismatch" [13]). This fact is reflected in our design considerations of the kernel's object buffer (see Section 3.3.3). Imagine a sparse matrix of which the nonzero elements together with their column numbers are stored row by row as a variable–length bytestring for each row. This structure obviously supports the extraction of a single row by the DBMS. If the whole sparse matrix is required, the object buffer should ideally provide the nonzero elements in a format that is directly usable by standard equation solver routines without moving data. However, in this case additional offsets into the array of nonzero elements have to be generated in order to mark the beginnings of each row. This example shows that some data types may require the derivation or conversion of some stored data into an internal format but the bulk of data is kept unchanged. Conversion, if necessary, is done by an "OUT" function, in the sense of [79].

4.3 Extensibility of Spatial Indexes

We followed two dimensions of index extensibility. One is the "storage structure extensibility" that is investigated more generally in [36]. We defined our Geo Access Manager in such a way that any spatial access method can be linked to the system if it supports a rectangular subdivision of space in two or three dimensions. However, unlike Starburst, we require such an access method to be implemented on top of the kernel.

The second dimension of extensibility could be called "genericity of access methods" and follows the direction in Postgres [64, 42]. The key to the genericity is the observation that we do not depend on details of the geometric object. In order to store the object or a reference to it into a page belonging to a cell, the only thing we need to know is the fact whether a cell intersects a geometric object or not, that is, an EDT function TEST is needed here. If the TEST result is positive, we store the reference to the object into that page in the case of an address grid file, or the bytestring representation of the object in the case of a (clipping) object grid file. This bytestring is produced by an EDT function CLIP. In either case, the page capacity may be too small which makes it necessary to split the cell into two smaller subcells. Consequently, the geometric objects of the original cell have to be TESTed to which subcell(s) they belong. In the case of an address grid file, the reference to an object has to be inserted into one of the pages, or even into both, which may well happen in the case of extended geometric objects. In the case of the object grid file, the objects are CLIPped at the subcells and the resulting byte portions are stored into the corresponding subcell pages.

4.4 Experiences and Lessons Learned

We have implemented an application-specific access manager for geometric types on top of the kernel system, following the objectives described above [71, 80]. A generic address grid file with a two-level directory is implemented on top of the kernel, and other spatial access methods such as the R-tree and a clipping grid file are being integrated. The interface of this extended kernel, shortly called geo-kernel, is used by the language processor which is being developed at the University of Braunschweig [40].

So far, the concepts have successfully been implemented, and the first performance figures, on real-life cadastrial data look promising (see Section 5.2). However, there are quite some lessons we learned. The most important ones are the following:

- *Need for kernel extensibility*: Our original design of the kernel did not provide data type support, because we wanted to keep it application-independent, whereas types are application-dependent. For the basic types, the kernel performs comparisons on the basis of bytes, following RSS in this aspect. Therefore, numeric data may have to be converted so as to allow numeric comparison at the byte level. The decision for such rudimentary type support was a mistake. Since we aimed at introducing new user defined types (EDTs), we had to provide a means for low-level filtering inside the kernel (on the byte level), to avoid unnecessary data movement from the buffer pool to the application. Thus, basic EDT functions for filtering in a selection have to be called. Furthermore, if the application needs only a small piece of a geometric object, the kernel could easily apply the EDT function CLIP on the bytestring stored in the pages (while performing a projection). Consequently, we redesigned our architecture and followed the overall architecture shown in Figure 7. This approach allows invocation of some EDT functions right in the kernel.

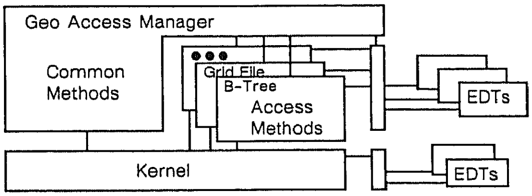


Fig.7: Extensible Geo-Kernel Architecture

- *Limits of kernel extensibility*: If a real conversion of data must be performed, that is, if a non-trivial OUT function [79] must be applied before the EDT operation can be called, this function will be called outside the kernel as the data have to be moved anyway. The consequence is that data should be stored into the pages in such a way that the operations can be applied without conversions.

- *Cost of EDT operations*: If a clipping grid file is updated, the only rule for splitting a cell is the byte count of the objects there. In view of a possibly expensive CLIP function, this must not be the only rule. It may be cheaper to store an unclipped object into two pages and to access two pages if the object is needed rather than to clip it simply because it does not fit into a page. As we have strong support for accessing sets of pages within the kernel, this may be justified further.

5 Preliminary Performance Evaluation

In this section we present preliminary performance figures of our system. In the first subsection we report on results on the kernel itself. The second subsection contains results on the geo frontend. Finally, we present first experiences with the multi–user version of our system.

5.1 Complex Object Queries

First measurements with the DASDBS kernel indicated that I/O time is almost negligible compared to CPU time even for medium–sized databases. Therefore, our experiments aimed at identifying CPU intensive parts. The test database contains two relations, a nested and a flat one, to analyze the overhead of complex structures. The nested relation stores papers having the set of authors as a subrelation. The papers are nested according to categories. So the schema of the nested relation is: Category(..., Paper(..., Author(...))). The flat relation contains only the papers without categories or authors. That is, its schema is: Paper(...). The database contains about 7,500 papers (in each of the relations) in about 2,000 categories (in the nested relation). Each paper has two authors on the average.

We present the results of two queries, both performing a projection on the paper attributes, that is, the complete tuples of the flat relation Paper and a nested projection of the nested relation Category. Both queries are processed as a relation scan, no indices are used. The difference between the queries is the selection condition: in the first query Q1, all papers published since 1970 are in the result set, that is, all papers of the database are selected. The second query Q2 has no results, as all papers are selected which are published before 1970. Thus, comparing the two queries, allows analyzing the object buffer costs.

Figure 8 presents the execution costs (CPU time) of these two queries against the two relations, broken down into the main modules of the kernel, namely SMM: Stable Memory Manager, OBM: Object Buffer Manager, and CRM: Complex Record Manager. The experiments were run on a Sun 3/280 server with 16 MB main memory. The figures have been obtained by the gprof utility of Unix. We present the figures of two object buffer implementations, namely the disk–oriented (suffix "_d") and the heap–oriented (suffix "_h") formats. Queries Q1' and Q2' refer to the flat relation Paper.

It is obvious that most of the time is spent on interpreting the structures of complex records.The advantage of the disk–oriented format, namely copying complex objects from the page buffer to the object buffer without object translation, was not utilized in this experiment. As Figure 8 shows, the speedup by using the heap–oriented OBM was quite significant. The

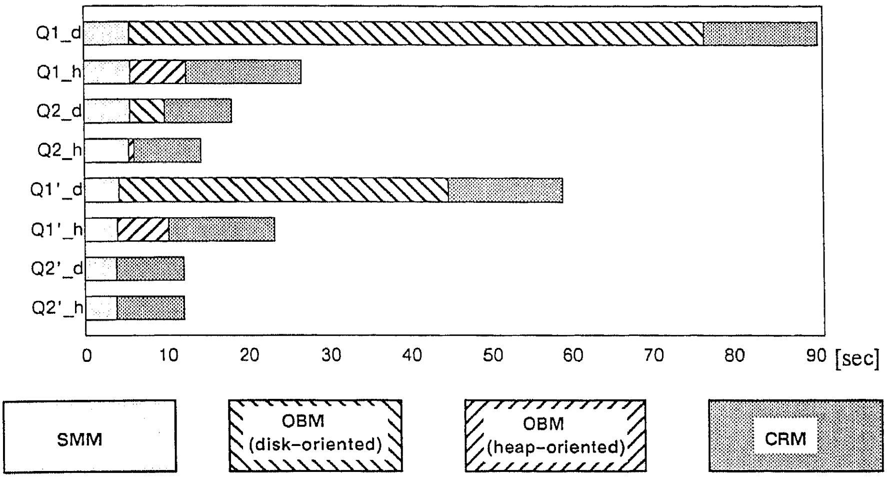


Fig.8: DASDBS Kernel Performance

results of Qi and Qi' show that the overhead of nested structures is minimal. While the difference in the CRM part is marginal, the main difference is in the OBM part. The reason is that filling an object buffer with Paper subtuples requires inserting a Category tuple first. If it turns out that a category does not contain any papers satisfying the search predicate and hence the Category itself is not included in the query result, then the Category tuple has to be removed from the object buffer after all its subtuples are scanned. This property of our single-scan query execution algorithm accounts for the OBM portion of query Q2.

5.2 Geo-Application Experiments

First evaluations of the extensible grid file used as a spatial index in the geo-kernel have been carried out using in-window selections. The test data collection is a set of parcels, simple non-overlapping polygons with 3 to 7 edges. Different DB sizes have been created with 7000, 22700, and 101000 objects covering areas of 25000 by 25000, 45000 by 45000, and 95000 by 95000 units in 1, 3.1, and 14.3 MBytes of external storage, respectively. The grid file index is based on the bounding boxes of the objects. If such a box overlaps cell boundaries, references are stored in each cell. The bucket size was 2 KBytes, the index sizes for the different databases were 860, 2460 and 12360 KBytes. The retrieval results are the average of 550 queries for each of four different window sizes. The query windows were randomly located in the data space. Figure 9 shows the index search execution times (CPU in seconds). The average result size is 4 points of a polygon, thus access to the actual data can be neglected, for very small items are retrieved and direct access via addresses is performed.

	#tuples	#ref's	#buckets	Query 1	Query 2	Query 3	Query 4
matches				1	7	40	70
window size				2 x 2	10 x 1000	100 x 10000	2000 x 2000
DB 1	7,000	10,600	430	0.25	0.33	0.82	1.20
DB 2	22,700	35,400	1,430	0.24	0.37	0.99	1.42
DB 3	101,000	161,000	6,280	0.27	0.47	1.63	2.59

Fig.9: Spatial Query Performance (in seconds)

Executing Query 1 as a relation scan against DB 1 required 13.8 seconds. Compared to this figure, the performance of the Geo Access Manager is quite encouraging. In a real–life geo database, the actual data size would be much larger than the index. This would make our results even better. An important information on the quality of the grid file as a clustering method is the ratio of objects and references. If we stored real geometries in the cells instead of bounding boxes, this ratio would indicate the number of calls of the EDT function CLIP. Since this clipping factor is only about 1.5, that is, 50 percent of the geometries would have been clipped, this poses no problem although the factor is slightly increased with the database size.

5.3 Multi–User Throughput

To gain first insights into the kernel's multi–user performance, we ran a toy version of debit/credit transactions (only 250 accounts, no branch and teller totals) on a SUN–3/280 with disk access over the local network. Disk I/O was performed remotely on purpose, in order to compensate for the moderate MIPS rate, that is, to simulate a better ratio of CPU vs. disk performance and to avoid getting into a CPU bottleneck too soon. We compared a page–oriented single–level transaction management approach (that is, directing BOT and EOT directly to the SMM) with the improved version of our 2–level transaction management. The results that we obtained at a multiprogramming degree of 4 are summarized in Figure 10.

	Through-put	# Lock Req. per minute		#Lock Waits per minute		Lock Wait Probability		# Deadlocks per minute		# Log I/Os per minute
	[TAs/sec]	L0	L1	L0	L1	L0	L1	L0	L1	
1-level TA Mgt.	2.29	4866	—	468	—	0.096	—	96	—	154
2-level TA Mgt.	1.97	3288	1946	33	24	0.010	0.012	0	2	147

Fig.10: Multi-user performance of the DASDBS kernel

The CPU was the throughput–limiting factor for both the single–level and the 2–level strategy, so data contention was not really the bottleneck. This accounts for the disappointing throughput of the 2–level strategy, for we have not yet paid sufficient attention to its CPU overhead (see Section 3.4.5). It is interesting to note, though, that the 2–level strategy achieved much better results with regard to the page lock wait probability (# L0 Lock Waits / # L0 Lock Requests) and the frequency of deadlocks.

Note that the presented results are only initial figures, obtained from a rather unstable version of our multi–level transaction manager. We are planning to port the DASDBS kernel to a

Sequent Symmetry shared–memory multiprocessor and make a more serious attempt to break the 2 TAs/sec mark. In addition, we think that debit/credit is too simple as a benchmark (especially our toy version), and are looking into more complex applications that demand high concurrency, such as electronic stock trading.

6. Future Prospects

Building a next–generation database system is, of course, a neverending project by definition. While we are still in the process of further evaluating the DASDBS kernel and fully implementing the planned application–oriented frontends, it is quite intriguing to design already the next system architecture. The main directions that we would like to pursue towards better application support and improved performance are the following:

- Nested relations in their pure form have successfully served as a model for storage structures. Encouraged by the strong similarities with nested relational concepts discovered in our analysis of existing semantic data models and knowledge representation techniques from AI, we pursue an evolutionary approach towards a richer data (or object) model that is useful as a conceptual data model. In fact, many researchers working on descriptive languages for their OODB models realized that if they try to carry over relational operations, they turn out to be more like nested relational than flat relational ones [30, 19, 9]. In a recent project called COCOON (recursive acronym for COcoon...Complex–Object–Orientation based on Nested relations), we are developing a new approach that incorporates recursive type definitions and thus the ability to model network structures [63]. The emphasis in this project lies on extensibility (the idea of a "starter" data model as a common backbone of DASDBS frontends), query optimization, and physical database design.

- To further evaluate the kernel architecture, we are building a Document AOM, that is, the application–specific frontend of an office information system [81], in a joint project with Mannesmann Kienzle. The underlying office data model includes objects such as documents, folders, keywords, user–defined attributes, text, and file cabinets, as well as different hierarchical and non–hierarchical relationships. We study different mappings of the office objects to complex record structures in order to evaluate different physical designs. Our office filing & retrieval system supports powerful retrieval operations on attributes and document contents using signatures as a special access method [16]. Furthermore, we are investigating the management of archives by optical disks. The next step will be extending the model and the system to capture hypermedia documents and providing suitable operations on them.

- Our object buffer concept and the EDT support in the kernel are steps towards a smoother coupling of application(–oriented) programs and database systems. As we see the necessity for and the prevalence of highly tailored data structures and specific programming environments in many application domains, we are not convinced that the persistent programming language direction [1] with its "seamlessness" paradigm will become a viable solution. After all, we will have to live with several programming languages (including FORTRAN) for another 30 years. Variety persists forever.

- Object copying and translation, especially pointer conversions, are a main source of unsatisfactory performance. One way of addressing this issue could be utilizing the emerging concept of mapped files [69, 53].This means that disk-resident data can be mapped directly into the virtual address space of the application. Such an approach has been pursued successfully in the Bubba project [11], based on a single-level store architecture. On the other hand, there is quite some controversy about whether large databases fit entirely into a 32-bit address space, whether longer addresses are the right direction to go, and if a single-level store is feasible in a distributed system (e.g., [46]). This suggests that mapped objects should be provided as an optional feature.
- Since the run-time interpretation of storage structures is one of the performance problems in the current DASDBS kernel, we are considering if it is worthwhile to compile operations on complex objects directly to the machine-instruction level (actually using C as an intermediate target language). Such an approach would probably involve redesigning the underlying storage structures so as to allow as much compile-time optimization as possible. As an ultimate goal, one could envisage that schema information and knowledge of data constraints could be exploited for generating object-specific code, e.g., code for retrieving the set of employee numbers in a particular department. This basically aims at replacing the generic kernel code by compiled (access) methods. In contrast, state-of-the-art relational database systems compile only to a level of RSS-like generic access operations.
- Finally, we are thinking of better ways to benefit from recent and expected technological advances. Like others, we plan to investigate the impact of large memory and disk-arrays [48] on the architecture of file systems and database systems. Moreover, with the advent of shared-memory multi-processors, hardly anybody can ignore the big issue of parallelism. It is interesting to note that the multi-level transaction approach already includes all the necessary synchronization mechanism for intra-transaction parallelism. This is because low-level subtransactions are treated uniformly, regardless of whether two subtransaction belong to the same transaction or to different ones. We plan to take multi-level transactions as one of several building blocks for future work on parallelism.

Acknowledgements

The roots of the NF^2 model go back to the early AIM project (1978–1982) at IBM Heidelberg Scientific Center (HDSC). Also, architectural considerations started there at the time when one of the authors (H.-J. Schek) was with IBM HDSC. The DASDBS project at the Technical University of Darmstadt (THD) started in 1983, concentrating on architectural aspects. Support by THD, the German Research Association (DFG), and Insotec Consult, Munich, is gratefully acknowledged. We are indebted to all (past and present) members of the DASDBS project, namely Martin Breunig, Peter Broessler, Gisbert Droege, Helmut Frankenbach, Dagmar Horn, Bin Jiang, Christian Laasch, Fatema Maher, Axel Moenkeberg, Volker Obermeit, and Christian Rich; particularly, Christof Hasse, Walter Waterfeld, Andreas Wolf, and Peter Zabback who provided us with the performance figures, Uwe Deppisch who designed the storage structures, and Arnie Rosenthal who helped in improving the presentation of the paper.

References

[1] Atkinson, M.P., Buneman, O.P., Types and Persistence in Database Programming Languages, ACM Computing Surveys Vol.19 No.2, 1987.

[2] Bancilhon, F., Richard, P., Scholl, M., On Line Processing of Compacted Relations, VLDB 1982

[3] Batory, D.S., Barnett, J.R., Garza, J.F., Smith, K.P., Tsukuda, K., Twichell, B.C., Wise, T.E., GENESIS: An Extensible Database Management System, IEEE Transactions on Software Engineering Vol.14 No.11, 1988

[4] Batory, D.S., Leung, T.Y., Wise, T.E., Implementation Concepts for an Extensible Data Model and Data Language, ACM TODS Vol.13 No.3, 1988

[5] Beeri, C., Bernstein, P.A., Goodman, N., A Model for Concurrency in Nested Transactions Systems, Journal of the ACM Vol.36 No.1, 1989

[6] Beeri, C., Schek, H.-J., Weikum, G., Multi-Level Transaction Management, Theoretical Art or Practical Need?, Int. Conf. on Extending Database Technology, Springer LNCS 303, 1988

[7] Carey, M.J., DeWitt, D.J., Frank, D., Graefe, G., Muralikrishna, M., Richardson, J.E., Shekita, E.J., The Architecture of the Exodus Extensible Database System, 1st Int. Workshop on Object-Oriented Database Systems, 1986

[8] Carey, M.J., DeWitt, D.J., Richardson,J.E., Shekita, E.J., Object and File Management in the Exodus Extensible Database System, VLDB 1986.

[9] Carey, M.J., DeWitt, D.J., Vandenberg, S.L., A Data Model and Query Language for Exodus, ACM SIGMOD 1988

[10] Cheng, J., Loosely, C., Shibamiya, A., Worthington, P., IBM Database 2 Performance: Design, Implementation, and Tuning, IBM Systems Journal Vol.23 No.2, 1984

[11] Copeland, G., Franklin, M., Weikum, G., Uniform Object Management, 2nd Int. Conf. Extending Database Technology, 1990

[12] Copeland, G., Keller, T., Krishnamurthy, R., Smith, M., The Case for Safe RAM, VLDB 1989

[13] Copeland, G., Maier, D., Making Smalltalk a Database System, ACM SIGMOD 1984

[14] Dadam, P., Kuespert, K., Andersen, F., Blanken, H., Erbe, R., Guenauer, J., Lum, V., Pistor, P., Walch, G., A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies, ACM SIGMOD 1986

[15] Dadam, P., Pistor, P., Schek, H.-J., A Predicate Oriented Locking Approach for Integrated Information Systems, IFIP World Congress, 1983

[16] Deppisch, U., Signatures in Database Systems, Ph.D. thesis, Technical University of Darmstadt, 1989 (in German).

[17] Deppisch, U., Paul, H.-B., Schek, H.-J., A Storage System for Complex Objects, 1st Int. Workshop on Object-Oriented Database Systems, 1986

[18] Deppisch, U., Paul, H.-B., Schek, H.-J., Weikum, G., Managing Complex Objects in the Darmstadt Database Kernel System, in: A. Buchman, U. Dayal, K. Dittrich (eds.), Object-Oriented Database Systems, Topics in Information Systems, Springer Verlag, to appear

[19] Derrett, N., Kent, W., Lyngbaek, P., Some Aspects of Operations in an Object-Oriented Database, IEEE Database Engineering Vol.11, No.4, 1988

[20] Dittrich, K.R., Gotthard, W., Lockemann, P.C., DAMOKLES - The Database System for the UNIBASE Software Engineering Environment, IEEE Database Engineering Vol.10 No.1, 1987

[21] Elhardt, K., Bayer, R., A Database Cache for High Performance and Fast Restart in Database Systems, ACM TODS Vol.9 No.4, 1984

[22] Ford, S. et al., ZEITGEIST: Database Support for Object-Oriented Programming, 2nd Workshop on Object-Oriented Database Systems, 1988

[23] Goldhirsch, D., Orenstein, J.A., Extensibility in the PROBE Database System, IEEE Database Engineering Vol.10 No.3, 1987

[24] Graefe, G., DeWitt, D., The Exodus Optimizer Generator, ACM SIGMOD 1987

[25] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., Traiger, I., The Recovery Manager of the System R Database Manager, ACM Computing Surveys Vol.13 No.2, 1981

[26] Guttman, A., Stonebraker, M. Using a Relational Database Management System for Computer Aided Design Data, IEEE Database Engineering Vol.5 No.2, 1982

[27] Haerder, T., Huebel, C., Langenfeld, S., Mitschang, B., KUNICAD - A Database-Supported Geometrical Modeling System for Solids (in German), Informatik Forschung und Entwicklung Vol.2 No.1, 1987

[28] Haerder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A., PRIMA - A DBMS Prototype Supporting Engineering Applications, VLDB 1987

[29] Jiang, B., Deadlock Detection is Really Cheap, ACM SIGMOD Record Vol.17 No.2, 1988

[30] Kim, W., A model of queries for object-oriented databases, VLDB 1989

[31] Kim, W., Ballou, N., Banerjee, J., Chou, H.-T., Garza, J.F., Woelk, D., Integrating an Object-Oriented Programming System with a Database System, OOPSLA 1988

[32] Kuespert, K., Dadam, P., Guenauer, J., Cooperative Object Buffer Management in the Advanced Information Management Prototype, VLDB 1987

[33] Kumar, A., Stonebraker, M., Performance Evaluation of an Operating System Transaction Manager, VLDB 1987

[34] Larson, P.-A., The data model and query language of LauRel, IEEE Database Engineering Vol.11 No.3 (Special Issue on Nested Relations), 1988

[35] Lausen, G., Schek, H.-J., Semantic Specification of Complex Objects, IEEE Symp. Office Automation, 1987

[36] Lindsay, B., McPherson, J., Pirahesh, H., A Database Management Extension Architecture, ACM SIGMOD 1987

[37] Linnemann, V., Kuespert, K., Dadam, P. et al., Design and Implementation of an Extensible Database Management System Supporting User-Defined Data Types and Functions, VLDB 1988

[38] Livny, M., Khoshafian, S., Boral, H., Multi-Disk Management Algorithms, ACM SIGMETRICS 1987

[39] Lohman, G., Grammar-like Functional Rules for Representing Query Optimization Alternatives, ACM SIGMOD 1988

[40] Lohmann, F., Neumann, K., Ehrich, H.-D., Design of a Database Prototype for Geoscientific Applications, 3rd GI Conf. on Database Systems in Office, Engineering, and Scientific Applications, 1989

[41] Lorie, R., Schek, H.-J., On dynamically defined complex objects and SQL, 2nd Int. Workshop on Object-Oriented Database Systems, Springer LNCS 334, 1988

[42] Lynch, C., Stonebraker, M., Extended User-Defined Indexing with Application to Textual Databases, VLDB 1988

[43] Manola, F., Orenstein, J., Dayal, U., Geographic Information Processing in the Probe Database System, 8th Int. Symp. on Automation in Cartography, 1987

[44] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P., ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, IBM Research Report RJ6649, San Jose, 1989

[45] Moss, J.E.B., Nested Transactions: An Approach to Reliable Distributed Computing, MIT Press, 1985

[46] Moss, J.E.B., Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach, 2nd Int'l Workshop on Database Programming Languages, 1989

[47] Moss, J.E.B., Griffeth, N.D., Graham, M.H., Abstraction in Recovery Management, ACM SIGMOD 1986

[48] Patterson, D., Gibson, G., Katz, R., A Case for Redundant Arrays of Inexpensive Disks (RAID), ACM SIGMOD 1988

[49] Paul, H.-B., The DASDBS Database Kernel System for Standard and Non-standard Applications - Architecture, Implementation, Applications (in German), Ph.D. thesis, Technical University of Darmstadt, 1988

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

[51] Paul, H.-B., Schek, H.-J., Soeder, A., Weikum, G., Supporting the Office Document Filing Service by a Database Kernel System (in German), 2nd GI Conf. on Database Systems for Office, Engineering, and Scientific Applications, 1987

[52] Reimer, U., Schek, H.-J., A frame-based knowledge representation model and its mapping to nested relations, to appear in Data and Knowledge Engineering, 1989.

[53] Rubsam, K.G., MVS Data Services, IBM Systems Journal Vol.28 No.1 (Special Issue on Enterprise Systems Architecture), 1989

[54] Schek, H.-J., Towards a Basic Relational NF^2 Algebra Processor, Int. Conf. on Foundations of Data Organization, 1985

[55] Schek, H.-J., Nested relations – a step forward or backward?, IEEE Database Engineering Vol.11 No.3 (Special Issue on Nested Relations),1988

[56] Schek, H.-J., Pistor, P., Data Structures for an Integrated Database Management and Information Retrieval System, VLDB 1982

[57] Schek, H.-J., Scholl, M.H., The relational model with relation-valued attributes, Information Systems Vol.11 No.2, 1986

[58] Schek, H.-J., Scholl, M.H., The two roles of nested relations in the DASDBS project, in: S. Abiteboul, P. C. Fischer, and H.-J. Schek (eds.), Nested Relations and Complex Objects in Databases, Springer, 1989.

[59] Schek, H.-J., Waterfeld, W., A Database Kernel System for Geoscientific Applications, Symp. on Spatial Data Handling, 1986

[60] Scholl, M.H., Theoretical Foundation of Algebraic Optimization Utilizing Unnormalized Relations, 1st Int. Conf. on Database Theory, Springer LNCS 243, 1986

[61] Scholl, M.H., The Nested Relational Model – Efficient Support for a Relational Database Interface (in German), Ph.D. Thesis,Technical University of Darmstadt, 1988

[62] Scholl, M.H., Paul, H.-B., Schek, H.-J., Supporting Flat Relations by a Nested Relational Kernel, VLDB 1987

[63] Scholl, M.H., Schek, H.-J., A Primer on Complex-Object-Orientation, Manuscript, ETH Zurich, 1989

[64] Stonebraker, M., Inclusion of New Types in Relational Database Systems, IEEE Data Engineering Conf., 1986

[65] Stonebraker, M., DuBordieux, D. Edwards, W., Problems in Supporting Data Base Transactions in an Operating System Transaction Manager, ACM Operating Systems Review, Vol. 19 No. 1, 1985

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

[67] Stonebraker, M., Rowe, L.A., The Design of Postgres, ACM SIGMOD 1986

[68] Stonebraker, M., Rubenstein, B., Guttman, A., Application of Abstract Data Types and Abstract indexes to CAD Data Bases, IEEE Conf. on Engineering Design Applications, Database Week, 1983

[69] Tevanian A. et al., A Unix Interface for Shared Memory and Memory Mapped Files Under Mach, Summer Usenix Conf., 1987

[70] Traiger, I.L., Trends in Systems Aspects of Database Management, 2nd Int. Conf. on Databases, 1983

[71] Waterfeld, W., Wolf, A., Horn, D., How to Make Spatial Access Methods Extensible, 3rd Int. Symp. on Spatial Data Handling, 1988

[72] Weikum, G., A Theoretical Foundation of Multi-Level Concurrency Control, ACM PODS 1986

[73] Weikum, G., Pros and Cons of Operating System Transactions for Data Base Systems, ACM/IEEE Fall Joint Computer Conf., 1986

[74] Weikum, G., Enhancing Concurrency in Layered Systems, 2nd Int. Workshop on High Performance Transaction Systems, 1987, Springer LNCS 359

[75] Weikum, G., Principles and Realization Strategies of Multi-Level Transaction Management, Technical Report, Technical University of Darmstadt, 1987

[76] Weikum, G., Set-Oriented Disk Access to Large Complex Objects, IEEE Data Engineering Conf., 1989

[77] Weikum, G., Hasse, C., Broessler, P., Muth, P., Multi-Level Recovery, Manuscript, ETH Zurich, 1989

[78] Weikum, G., Schek, H.-J., Architectural Issues of Transaction Management in Layered Systems, VLDB 1984

[79] Wilms, P.F., Schwarz, P.M., Schek, H.-J., Haas, L.M., Incorporating Data Types in an Extensible Database Architecture, 3rd Int. Conf. on Data and Knowledge Bases, 1988

[80] Wolf, A., The DASDBS Geo-Kernel: Concepts, Experiences, and the Second Step, Int. Symp. on Design and Implementation of Large Spatial Databases, 1989

[81] Zabback, P., Paul, H.-B., Deppisch, U., Office Documents on a Database Kernel - Filing, Retrieval, and Archiving, Manuscript, ETH Zurich, 1989