Modularization of the DADAISM Ada Database System Architecture

Arthur M. Keller Gio Wiederhold
Stanford University

Draft of May 21, 1991

Abstract. We describe a new database management system architecture designed for long term evolutionary growth of modular DBMSs. This architecture includes the development of Ada package specifications for the individual modules. Alternative code bodies can be written corresponding to these package specifications for variations in desired functionality. These alternative code bodies can also exist side by side, such as a hash structure package along with a B-tree package, or a relational storage module along with an object storage module. Functional specifications are written in Anna, a formal annotation language for Ada. This effort does not represent a new data model but rather a new approach to database implementation. This approach can be used to build database systems of any model, including a hybrid relational and object store approach suitable for applications such as Computer Aided Software Engineering (CASE). Our architecture specifically takes into consideration multi-level security, distributed and parallel computing environments, and evolutionary growth.

1. Introduction

This paper presents the modularization of a new database management system architecture. This DBMS is not intended to be viewed as another implementation of one of the traditional models: relational, hierarchical, network, universal, relational, or functional; nor does it present a new model. This design presents a collection of module specifications for a DBMS, which can be used to support any of those data models with the appropriate implementations. The selection of these modules as well as the implementation details are determined by functional and performance requirements of the specific system.

For the descriptions and explanations, we will emphasize modules that support the relational model, because of the clarity and formality of this model. The considerations for relational implementations [Codd 86] are also important when other models are being supported within this scheme.

We believe that a new database system implementation paradigm is necessary to handle the evolution of database systems over the next few decades. Hence, we are describing an architecture for database system implementation. This architecture is to be distinguished from the GENESIS [Bato 88] and EXODUS [CDRS 86] projects. The GENESIS effort for retargetable database implementations and a database system compiler is concerned with allowing different database structures and new database algorithms, and they could be used to generate modules that can be integrated into our evolutionary architecture. EXODUS is a toolkit for specific database applications and supports only a much coarser granularity than DADAISM.

1.1. Context

The DADAISM project (**D**atabases in and for **Ada**-supported **I**nformation **S**ystem **M**anagement) is part of the Ada Foundation Technology efforts of the Software Technology for Adaptable and Reliable Systems (STARS) program. STARS is a major software initiative to make Ada technology available to the Department of Defense community. An important application of the STARS technology is to develop Computer Aided Software Engineering (CASE) tools that facilitate the creation, maintenance, and reuse of software.

This research was originally motivated by a plan to re-engineer a large multi-site control and command system, the Worldwide Military Command and Control System (WWMCCS), using modern software principles. WWMCCS is intended to provide a capability to receive information, apply the resources, assign missions, and

Authors' current addresses: Arthur M. Keller, Stanford University, Computer Science Dept., Stanford, CA 94305-2140, ark@cs.stanford.edu; Gio Wiederhold, wiederhold@cs.stanford.edu.

The work reported in this document was conducted under contract N14-87-C-2503 from the Naval Research Laboratory. This work is based in part on prior work conducted as part of Institute for Defense Analyses Project T-4-206 under Contract MDA-903-84-C-0031 for the Department of Defense. The publication of this paper does not indicate endorsement by the Department of Defense, the Naval Research Laboratory, Advanced Decision Systems, Stanford University, SRI International, or Institute for Defense Analyses, nor should the contents be construed as reflecting the official positions of these organizations.

This paper is based in part on "Modularization of an Ada Database System," by Gio Wiederhold, Arthur M. Keller, Sham Navathe, David Spooner, Murray Berkowitz, Bill Brykczynski, and John Salasin, which was presented at the Pre-VLDB Conference on August 21–22, 1986 at Beijing, China.

provide direction to the unified commands. It evolved to a means for continuous and responsible command and control, and is today the Communications, Command, Control, and Intelligence (C^3I) backbone. It consists of a network of warning sensors, command center computer systems, and telecommunications. It is used for worldwide control of US military forces in peacetime, crisis, or war. High priority is given to C^3I initiatives, such as the WWMCCS Information System (WIS) upgrade. In accordance with recent Department of Defense (DoD) policies, the "mission-critical software" for the WIS will be designed and implemented in Ada.

The function of the WIS DBMS is to manage all user and system-generated data in the form of database objects, and to manage higher-level information concerning each of these objects as database directories and dictionaries. The WIS DBMS also manages information concerning users of the system and uses this information to control access to database objects and WIS applications and tools.

Currently, multiple approaches are being considered. One approach is to recode existing data-processing programs. Another approach is to embed of existing *Commercial-Off-The-Shelf* (COTS) database technology within Ada [Frie 86]. Database systems embedding new concepts have been developed [CDFG 83], and those concepts are being used experimentally in complex settings. Other current efforts include making SQL, as now defined, accessible to Ada as a host language [SAME 88]. Unfortunately, since the SQL language was not originally designed to serve a host function, this interface remains awkward [Stonebraker 88]. DADAISM must make SQL available as a user function, but does not commit its entire processing architecture only to SQL services.

The work described in this paper is part of an advanced development task. The concept underlying the specific work described here is to *start out fresh*. Our motivation is to provide new software and data engineering approaches to database systems, using the power of Ada in the most appropriate way. We consider the database implementation problem as consisting of layers, starting with a simple operating system interface and proceeding to the level of knowledge-based user services. We believe strongly that the cost of adapting re-engineered applications to 20-year and 10-year old technology, as exemplified by existing DBMSs, is greater than the cost of rethinking a new DBMS implementation. If we do not apply *now* the principles we have learned in the past twenty years, we will be overtaken by others who can start afresh.

Proposing a fresh start for a major system carries a considerable risk. Doing this in the database area is made more feasible due to the initial orientation of Ada. Since Ada was designed as a language for embedded systems, it did not inherit any prior data-processing notions—a problem other general purpose languages as PL/I had to cope with. Only the most primitive input-output functions are specified in Ada. Hence, we start with a relatively clean slate.

1.2. Architecture

The DADAISM architecture takes advantage of the structure of Ada. Since Ada supports alternative code bodies with the same declarative specification, it becomes natural to maintain and use alternate modules. However, in order to keep the specification unchanged, the initial design must consider the information requirements by any candidate module. For instance, modules developed in the prototyping stage will not use information needed for an optimizing version of the same module type. Similarly, modules for a workstation database will not use much of the security information available for protection in a shared environment.

For the modular design to be valid over a range of environments, information interfaces must be specified that provide access to optional information. It is difficult to assure that in a simple prototype all requirements have been considered. Our approach is to permit growth in this area by specifying a module interface for access to schema information, and permitting the schema information to grow as well. Mismatches are to be handled by default arrangements and overloading.

The context of this research project reinforced the consideration of security and access controls within the initial design, rather than treating security as add-on features. If a high level of access protection is needed, overall reliability and performance will be improved by incorporating clean and appropriate interfaces in the original system design. Whenever the requirements are less strict, the cost of having null or minimal modules in their place will be small.

1.3. DBMS Benefits

We wish to list succinctly the benefits to be expected from the use of a database management system, so that we can later verify that these advantages are indeed obtained.

1 *Improved information sharing,* to be achieved through data independence (i.e., permitting most procedures to remain unchanged, while data structures adapt to deal with new processing requirements). The essential feature is keeping the knowl-

edge about the data in a schema rather than individual programs; such a schema can easily be viewed as an extension of Ada specifications.

2 *Control of security and integrity,* achieved by providing a careful interface between the users and the persistent storage. Since the data are shared the access specifications must permit references at a small granularity (i.e., individual records and their fields).

3 *Distribution of data* must be supported, since any modern computing system will employ multiple processors, and local and remote networks. The data may be replicated at multiple nodes, which increases availability and performance but introduces problems of consistency maintenance.

Many of these objectives are inadequately served by existing DBMSs. Most DBMSs used in large-scale data processing were designed in the late 60s. Although the systems may be fairly efficient, once installed, they suffer from obsolete technology, inflexible models, and awkward distribution. On the other hand, modern DBMSs, often based on the relational model, focus on transaction processing, simplifying programming and distribution, and also provide a direct user interface. However, they often provide a poor program interface; they display a major impendance mismatch between their set-oriented operations and the value and record-oriented operating modes of traditional programming languages.

With these considerations, we are developing a modern DBMS, written in Ada, that is carefully modularized to permit enhancements and growth. We believe that the one-time development cost of a new modular DBMS is much less than the cost of maintaining a patchwork, non-modular system.

1.4. Layers

In an earlier report we identified layers appropriate for a database management system to serve a broad and diverse community [FKWB 86]. These layers have been refined to the following structure:

$\mathcal{L}6$. Application-oriented packages, including knowledge-base aided access for high-level users.
$\mathcal{L}5$. Database Access, using query languages such as SQL or QBE based on the relational calculus.
— *Integrity Boundary* —
$\mathcal{L}4$. Database integrity layer enforcing multi-relation contraints.
— *Discretionary Security Boundary* —
$\mathcal{L}3$. Relation layer implementing single relation selection operations.
$\mathcal{L}2$. File access layer based on a schema and multi-indexed files.
$\mathcal{L}1$. System services for management of external storage.
— *Mandatory Security Boundary* —
$\mathcal{L}0$. Operating system and hardware.

These layers provide guidelines for defining abstract machines in the sense of Edsger Dijkstra [Dijk 66,71], but do not provide the granularity needed for the decomposition of a large software system into modules that lend themselves to independent development, reuse, and replacement.

1.5. Concepts

On reflection, it appears that the architectural approach proposed here, as a conceptualization of architectures found in current DBMS, bears a significant resemblance to the *blackboard concepts* used in some Artificial Intelligence Systems. An early example of a blackboard is found in in HEARSAY [REFN 76] and recent work is BB1 [Haye 85].

We make a clear distinction in this design document between knowledge and data:

- **Knowledge:** Controlling and general information, typically complex and relatively small in volume.
- **Data:** Manipulated and factual information, typically regular but voluminous.

This distinction helps to reveal design issues and rules for information management within the design. These concepts can also help generate input for subsequent design phases:

- For transferring *knowledge-associated information*, calling mechanisms and list-structures are appropriate.
- For transferring *Data-associated information*, block moves, tabular structures, and shared access are appropriate.

There will be many high-demand, knowledge-oriented structures for which the efficiency of access will be critical to performance because the access is of high frequency. The schema will contain such information. An evaluation of information flow along the described inter-module paths will quantify the demand, and we have already considered techniques to satisfy high frequency access.

2. Interfaces

The critical design issue for the users is the interface between their tasks and the DBMS. The interface must be adapted to the kind of use being made of the DBMS, and support an adequate level of performance. We consider data-processing requirements in particular.

Most data processing today is performed by programs, and not by end-users sitting at terminals. Unless this mode of operation is going to disappear, a DBMS must provide appropriate interfaces for the Ada programmer that are

1 Consistent with the architecture, language types and structure, and
2 Comparable in performance to programmer-defined files.

We do not intend to exclude on-line terminal access to the DBMS for queries and planning functions. However, a prerequisite for such access is storing the data in the database, which is only possible if the DBMS is accepted by programmers.

2.1. User Access Alternatives

We can now summarize the alternative access interfaces which a comprehensive DBMS must be able to serve.

1 Program access to relational algebra operations ($\mathcal{M}$451)
2 Compiled transaction access by modules supporting general relational access and update as represented by SQL ($\mathcal{M}$401-409)
3 Fast access for ad-hoc transactions, interpreted by modules ($\mathcal{M}$411-419)
4 Specialized user access, mediated by prestored analytical programs ($\mathcal{M}$421-439)
5 High level query language access, as increasingly used by planners, often referred to as 4th generation languages ($\mathcal{M}$501)
6 Inferential access by knowledge-based approaches ($\mathcal{M}$521)
7 Direct file level access for high performance trusted programs ($\mathcal{M}$101)

Having multiple interfaces into a single system avoids the awkwardness of adapting to a single interface, such as SQL, for a variety of environments. It also removes many valid objections that system managers can present to the use of databases. For instance, real-time sensor input is, in practice, accommodated at a low level via trusted programs because of volume and throughput requirements. Conventional query protection is meaningless and costly for such tasks.

2.2. Modularity

Certain types of functions are assigned to each layer. We associate each class of function with a DBMS program module. Describing these modules in turn will help to clarify the layering.

The granularity of the initial modules was in part determined by existing software. Consequently, the functions for these modules are relatively easy to specify. But this causes these modules to cover multiple layers. This paper specifies a much more general and finer grained decomposition.

Implicit in these suggestions is the notion that a modern DBMS can be constructed in a modular fashion. Existing DBMSs cannot be used to prove that hypothesis; most currently existing systems are too massive and integrated. (While some modern DBMSs are designed in a modular fashion, the result is still an integrated DBMSs that cannot accommodate substitution of some modules while retaining others.) A user must choose between taking all features of a system, or leaving the system alone. The supporting database can no longer be modernized without incurring risks of losing reliability. Growth and new equipment cannot be accommodated. Now, improvement of the applications requires a tremendous porting effort to new software support systems, with many problems of compatibility between new and old programs.

Without modularity, the new DBMSs cannot promise all features required for demanding applications initially, nor have the flexibility to adapt to later changes. Without a suitable DBMS, the users build their own functionally equivalent system using conventional data-processing approaches. With such a particular system all needs currently recognized may be served, but the liabilities of high maintenance cost and growth limitations are great.

We strongly believe that true modular design and implementation is essential to promote economical and controlled growth of a DBMS. The underlying language structure of Ada is meant to support such a premise. In the remainder of this paper we will investigate the issues of modularity for a DBMS and propose a specific modular system organization.

2.3. Decomposition

There are several criteria for modular decomposition. We consider three in parallel:

1 The first criterion is *layers of abstraction*. This aspect was already shown in terms of the conceptual layers listed in the introduction. There should be an orderly hierarchy of services. Each layer should support all services needed within a consistent computing paradigm.

2 The second criterion is *function*. We encourage the replacement of modules to support alteration in function as perceived by the user. For instance, within layer $\mathcal{L}5$ an SQL language processor should be interchangeable with a QBE processor. This provides a radically different interface to the user, with at most slightly different functionality, but using substantially the same services from layer $\mathcal{L}4$.

3 A third criterion is *performance*. Satisfying performance requirements stresses internal interfaces. Since modules need simple and reliable interfaces, the semantics of communication must be simple, regular, and devoid of side effects. Layers and modules introduce interfaces which cause performance losses. To avoid excessive losses, there should be as few interfaces as feasible in high-bandwidth data flows. Furthermore, control flows should serve large granules (i.e., a command should address the largest possible data unit rather than, say, single bytes).

Some of these criteria are in conflict with each other. We have decided that correctness and clarity should dominate performance issues, since we believe that in the long term, *messy* systems will degrade faster than *clean* ones.

3. Piercing the Layers

Orthogonal to the layers are several internal functions which cannot be isolated to one layer only. These functions require great care during design, since it is here that side effects of computations easily become problematic. Specifically, concerns that transcend layers are:

- Data definition and schema access.
- Concurrency control.
- Logging for recovery and audit.
- Recovery actions.
- Distribution and replication of data.
- Security.

A special issue in security is the specification of discretionary protection, since the required partitioning is orthogonal to the layers as defined here [Spoo 86].

Our solution for multi-layer functions is to assign the controlling tasks to modules that can be well identified within a single layer, and then to identify formal structures for information sharing that can "pierce" the layers.

4. Modules

The table below will list the modules being considered and the layer which owns them. The modules are grouped by primary function, where the groups correspond to the following subsections of this document.

We will now discuss the individual modules in a somewhat schematic way. A short introduction will identify the functions expected of each group of modules.

A. The functions will be specified further for each actual module (subsection A). The functions can often be carried out in a variety of ways, so that it is important not to overspecify the modules procedurally.

B. Our design process is *information driven* from a high-level design point-of-view. That is, for every module, we consider the information it needs for its function, as well as what and where the source of knowledge and data are (subsection B). For instance, from this point of view, an SQL interpreter may be indistinguishable from a QBE interpreter. However, a compiler for SQL will differ from an interpreter, at least in the timing of its information needs. The distinctions will increase as the module specifications become more refined.

We use the term *knowledge* here to identify the information used to control the decision-making components of the module. We use the term *data* for information which is largely passively processed, reaching either the database storage or presentation to the end user. At layer $\mathcal{L}6$, some data may be transformed and reduced. It is there that a merger of knowledge and data occurs.

C. Most modules do not require internal storage extending beyond the length of their call, but some modules will have to maintain state information internally. Information typically has to be retained across several calls for the length of a transaction (subsection C). Ada permits packages to declare "own" variables, and permits packages to be used by multiple tasks, so that adequate language mechanisms exist. Languages which use strictly hierarchical scoping are unsuitable, since internal state information cannot be owned and retained by the module and has to be placed in the outermost scope of the system. We do not specify the organization of the repository for such information, since that can be very dependent on the capability of the underlying operating system. All operating systems provide various means to identify, maintain, and switch information associated with user processes.

4.1. File Access Management

It is critical for database systems to have an effective

Number	Module Name	Layer Assignment
$\mathcal{M}$101	Object and record manager	$\mathcal{L}$2
$\mathcal{M}$102	Block manager	$\mathcal{L}$1
$\mathcal{M}$103	Asynchronous input-output	$\mathcal{L}$1
$\mathcal{M}$104	Access path manager	$\mathcal{L}$2
$\mathcal{M}$105	Nested segment manager	$\mathcal{L}$2
$\mathcal{M}$201	Schema table manager	$\mathcal{L}$3
$\mathcal{M}$211	Schema language executor	$\mathcal{L}$4
$\mathcal{M}$212	Reorganization manager	$\mathcal{L}$3
$\mathcal{M}$221–$\mathcal{M}$229	Schema language interfaces	$\mathcal{L}$5
$\mathcal{M}$301	View manager	$\mathcal{L}$4
$\mathcal{M}$302	Query execution planner/optimizer	$\mathcal{L}$4
$\mathcal{M}$311	Single-stream operation executor	$\mathcal{L}$3
$\mathcal{M}$312	Multi-stream operation executor	$\mathcal{L}$4
$\mathcal{M}$401-$\mathcal{M}$419	System-provided DBMS interfaces	$\mathcal{L}$5
$\mathcal{M}$421-$\mathcal{M}$439	User-provided DBMS interfaces	$\mathcal{L}$5
$\mathcal{M}$451	Algebraic interface	$\mathcal{L}$5
$\mathcal{M}$501-$\mathcal{M}$519	High-level query processors	$\mathcal{L}$6
$\mathcal{M}$521-$\mathcal{M}$539	Knowledge-based processors	$\mathcal{L}$6
$\mathcal{M}$601	Transaction manager	$\mathcal{L}$2
$\mathcal{M}$602	Lock manager	$\mathcal{L}$1
$\mathcal{M}$611	Scheduler for multiple users	$\mathcal{L}$1
$\mathcal{M}$621	Log Manager	$\mathcal{L}$1
$\mathcal{M}$622	Recovery Manager	$\mathcal{L}$2
$\mathcal{M}$701	Distributed access mediator	$\mathcal{L}$4
$\mathcal{M}$702	Federated access mediator	$\mathcal{L}$4
$\mathcal{M}$711	Distributed access driver	$\mathcal{L}$1
$\mathcal{M}$721	Distributed access server	$\mathcal{L}$5
$\mathcal{M}$801	Administrative interface	$\mathcal{L}$4
$\mathcal{M}$802	High-level auditor	$\mathcal{L}$4
$\mathcal{M}$801	Low-level auditor	$\mathcal{L}$2
$\mathcal{M}$901	Security filter	$\mathcal{L}$3
$\mathcal{M}$902	Security table manager	$\mathcal{L}$3
$\mathcal{M}$911	Security officer interface	$\mathcal{L}$3
$\mathcal{M}$921	Security performance transformation	$\mathcal{L}$4

interface with the operating system functions. Often the requirements of database management are at odds with their operating systems. We find that DBMSs replicate much of the operating system code, so that functions such as buffering and scheduling are carried out in duplicate. We identify those functions that we would like to be performed by the operating system by placing them in layer $\mathcal{L}$1. This does place some requirements of functionality and access to information on the operating system. Specifically, while there may be functional transparency about locality of data, locality information is needed for a database optimizer to be effective, and control over locality of data can improve the performance of clustered access structures. We distinguish the issue of information hiding in the case that the user is, in actuality, another software system that can automatically, or via the DBA, adapt to changes in underlying layers, from the case when the user is a programmer, who will not be available when changes in the support layer occur.

4.$\mathcal{M}$101. Indexed file access ($\mathcal{L}$2)

A. The File Access Module transforms the named and partitioned (paged) address space provided by operating

system's file services into a set of named, variable length records, accessible by any of several name sets. The name sets are typically implemented by indexes, although hashing and tries are alternative access techniques. The File Access Module is the primary module that interfaces with the functions at the operating system's file services level.

We expect records to be of variable length. A simple file access system may map the variable length user records into fixed length units within the operating systems' address spaces, but in general we expect an $\mathcal{M}101$ module to do better. These functions are somewhat more demanding than many currently so-called *file access methods*.

Performance issues dictate the configuration, since this module is in the main flow. For instance, when file access methods fail to provide for variable length records, there is invariably a variable-to-fixed record mapping routine at a higher layer. Adequate prototypes of such file support systems do exist [Allc 80], [DCKK 85], and we have used them as models in the design of this module. The function of this module for Ada is described in more detail in [Kell 83, Kell 87]. The specifications for the module have been completed [Ama 88]. The File Access Module occupies most of layer $\mathcal{L}2$.

We envision the potential for multiple modules occupying this slot in the architecture. The particular module we describe here supports relational storage using B-tree indexes. An additional module can support hash table access to data. Non-relational but record oriented user interfaces can still use the storage structures of these modules. However, for object-oriented database systems, an object store module will occupy this slot, possibly along with the relational storage modules that contain control information. This hybrid approach appears promising for applications such as CASE tools.

B. In order to make the File Access Module self-contained, an abbreviated schema definition is stored internally in each file. The file access routines only use information specifying record and field formats (in the file schema), since they are not sensitive to the contents of a file. The Record Management System for DEC's VAX-VMS [RMS] provides a model of the schema requirements at this layer.

The File Access Module must be able to locate fields within data records for index creation and may need to know the data representation for collating sequence for indexes and for more effective utilization of storage. This information is stored in the file schema, which is extracted from the database schema (stored in several files) by the Reorganization Manager ($\mathcal{M}212$) when the file is created or reorganized. The file schema is inspected at open and close of a file. Hence, it is the responsibility of the Transaction Manager ($\mathcal{M}601$) to assure that files are closed if a reorganization is requested.

The control flow for file access management is mainly associated with query and update requests delivered by the processor which handles access to single relations ($\mathcal{M}311$). Certain operations, such as creating and destroying files, are delivered from the database administrator by the Schema Language Executor ($\mathcal{M}211$) via the Reorganization Manager ($\mathcal{M}212$).

The data obtained in response to a query is always obtained from layer $\mathcal{L}1$ using the Block Manager ($\mathcal{M}102$), and data for file update flow to $\mathcal{M}102$.

C. This module maintains internal state as long as a file is in use. Local data structures include:

- Identification of the file currently accessed.
- Contents and identification of current data block.
- Contents and identification of the current access path.

Note that this information is maintained on a per file-open basis. Hence all local information can be referenced using the file ID private type as a pointer to the data structure associated with a file open. Appropriate use of the Block Manager ($\mathcal{M}102$) and the Lock Manager ($\mathcal{M}602$) ensures that the File Access Module need not keep additional state locally.

The identification of the data objects must be protected from corruption, so that access checking by the Security Filter ($\mathcal{M}903$) of privileges assigned to users for operations to data objects cannot be bypassed.

4.$\mathcal{M}$102. Block Manager ($\mathcal{L}1$)

A. Block management services are best provided by the operating system. The function is only listed here to help in the definition of its interface. Specifications for the module have been completed [Kenn 88a]. Some current DBMSs do provide these services internally because of perceived inadequacies of their operating systems. This is the case, for instance, in DEC VMS where the relational (RDB) and the network (DBMS-32) DBMSs share a buffer manager outside of the operating system [Yang 86].

The Block Manager ensures the integrity of file blocks in two ways. First, blocks currently being manipulated as kept in buffers, and each buffer has a lock status. Second, we assume that a block is either completely written to disk or that the disk is unchanged.

The Block Manager also assists the File Access Module ($\mathcal{M}101$) in maintaining file integrity. The Block Manager coordinates asynchronous input-output performed by module $\mathcal{M}103$. Integrity of the disk data structures over a crash requires that the blocks be written in a particular order, specifically that blocks are written before blocks that point to them are written. This information is encoded in the level number as described below.

B. The Block Manager receives data and information about the data blocks to be retrieved or stored from the File Access Module ($\mathcal{M}101$) and the Log Manager ($\mathcal{M}621$). Under control of the operating system, it ships the blocks to the Ada run-time environment of the operating system. In a distributed file system, blocks for selected files may be shipped directly to the communication interface of the operating system. In a distributed database system, we prefer to ship records or streams of records rather than blocks, using the Distributed Access Driver ($\mathcal{M}711$).

To optimize disk access times, the Block Manager may rearrange the order in which blocks are stored and fetched. However, blocks must often be deposited in a constrained partial order to avoid inconsistencies when systems crash [Wied 83a, p. 625]. The information provided by the File Access Module ($\mathcal{M}101$) has to be adequate to identify such constraints. This information is encoded in the level number (as in level of a B-tree, for example) given by the File Access Module ($\mathcal{M}101$) to the Block Manager, which schedules deferred block writes accordingly.

C. In addition to the data buffers themselves, the Block Manager maintains some knowledge about the state information such as:

- Buffer identification and file origin.
- Buffer users.
- Buffer clean or dirty.
- Buffer safe (i.e., already copied to persistent storage).
- Buffer status pending (i.e., fetch or store pending or in progress).
- Buffer lock state.

The buffer pool for a file must be shared among all users of the file, except in a distributed system when more complex algorithms apply. This requires that the Block Manager must maintain in a static location the head of a linked list of buffer pools. This linked list is referenced as such only at file open and close when the association between user file ID and buffer pool is made.

In a distributed system where the actual files may be allocated (and perhaps replicated), such allocation information has to be made available for optimization by the Query Execution Planner/Optimizer ($\mathcal{M}302$) at level $\mathcal{L}4$. The information is stored in the database schema. Some distinguished operating systems attempt to hide from the user where a file is allocated to prevent problems that could arise if the user shortcuts the formal access path. However, for an automated system, such concerns are not relevant. Shortcutting an access path is, in fact, a legitimate optimization, and the system will automatically adapt to any change in distribution allocation.

4.$\mathcal{M}$103. Asynchronous Input-output ($\mathcal{L}$1)

A. Ada input-output is essentially synchronous. This module is necessary to allow asynchronous input-output where permitted by the operating system and run-time environment. This module is best provided as part of the operating system's file services.

For access to the storage devices, an operating system will provide routines to read or write blocks on specific devices at specific storage locations. We do not expect those routines to be a responsibility of the DBMS. A minimal interface may be required on systems which do not provide an external interface for block access, but only access to files viewed as continuous character streams or pipes. The actual transfer to storage is by block, hence such an interface will be minimal.

B. The primary control and data flow to this module emanates from the Block Manager $\mathcal{M}102$ with every block of data being requested or deposited. On extremely simple systems, with low performance demands, the Block Manager may be elided or replaced by a minimal command translating interface, so that in those systems this flow comes directly from the file access manager $\mathcal{M}101$.

A secondary stream of control and data flow comes through the Block Manager ($\mathcal{M}102$) from the logger $\mathcal{M}621$ and may be obtained by the recovery manager $\mathcal{M}622$. These two types of flow will always be to distinct devices.

Information about file status is made available on request to the Schema Table Manager $\mathcal{M}201$, to complement actual schema information with current file status. All outgoing data and control flow disappears into the operating system.

C. This module, in conjunction with the operating system's file services, maintains a fair amount of internal status about the devices and their current status relative to each open file. The only access to this information is passive, via requests made by the Schema Table Manager $\mathcal{M}$201.

4.2. Schema Management

The schema is used to hold all descriptive information about the data, except for security information. The concept of having a schema is the main distinguishing feature between traditional data processing and database processing. We noted already that the schema transcends several layers and that this information has to be carefully managed. However, since most of the information flows are one way, the protection of integrity is simplified.

4.$\mathcal{M}$201. Schema Table Manager ($\mathcal{L}$4)

A. This module maintains as data the knowledge used to drive the system. It is responsible for maintaining the database schema stored in files (but not the individual file schemas) and keeping a schema cache for database access. Most of the information is static during normal use, so that update activity of this module is low. The procedure to maintain the schema is only active during the receipt of updates to the schema from the Schema Language Executor ($\mathcal{M}$211) module. The update interface to the Schema Table Manager ($\mathcal{M}$201) is the same as the update interface to the Multi-Relation Operation Executor ($\mathcal{M}$312), but the query interface to the Schema Table Manager is simpler. Specifications for the module are described elsewhere [Marm 88].

B. The knowledge to create schema entries derives nearly completely from the database administrator, using module $\mathcal{M}$201. The encoded knowledge, or meta-data, is information being stored in this module. The meta-data must be persistent, and is, of course, stored on database files.

Schema data are stored using the File Access Module ($\mathcal{M}$101), using directly the Single Relation Operation Executor ($\mathcal{M}$311). It is important that security be provided for this information, using module $\mathcal{M}$903.

The stored schema data will be available to any user having the appropriate access privileges, although we expect that write-access will be tightly controlled by the security officer. Columns of the schema table, as described below, will correspond for security protection with attributes in a relational model, and will typically have distinct access privileges.

C. This module is the major source of state for the system. Examples of typical schema contents are found in [Wied 83a, chapter 8]. A basic set is specified using Ada in the proposed standard for Ada/SQL access [Frie 86]. Types of information generated at the various layers are:

$\mathcal{L}$6 From knowledge-based routines: application and user semantics, fuzzy knowledge

$\mathcal{L}$4 From the database administrator information relevant to:

ENTITIES Assignment of attributes to files or relations, domain definitions for attributes, internal representation formats, etc.

RELATIONSHIPS High level data semantics as interfile constraints and interfile dependencies, view specifications.

USERS Presentation formats, data labels, units definition, etc.

PERFORMANCE Access path specification as indexing directives and replication information, recovery directives which specify to the logger what should be placed on the logging files for recovery from failures, static optimization information as initial cardinalities.

$\mathcal{L}$3 Database reorganization, upon completion of a task, will update the schema to reflect the latest physical allocations of data.

$\mathcal{L}$1 On demand from the file modules: Last user, last time used, type of last use, transaction name of last use, cardinality, other information helpful for optimization.

Neither mandatory nor discretionary security information is kept within the schema. A separate security table $\mathcal{M}$901 is employed for that purpose.

Data obtained from storage may be cached in memory while the database is in use. We expect the local, active schema information to be organized as a table; a typical row of the table describes a relation or an attribute of a relation. The columns of the table contain information types, as *representation*, *size*, *access-path*, etc. Each type of element of the schema will be associated for update with no more than one layer.

4.$\mathcal{M}$211. Schema Language Executor ($\mathcal{L}$4)

A. The Schema Language Executor processes schema change requests and atomically passes them onto the Schema Table Manager ($\mathcal{M}$201) for updating the schema files and cache and onto the Reorganization Manager ($\mathcal{M}$212) for conversion of the data files. Information about the current state of the schema is obtained from the Schema Table Managers ($\mathcal{M}$201). Specifications for the module have been written [Zele 88].

B. Reading of schema table information using module $\mathcal{M}201$ can transcend levels, but to control information flow and implement hiding concepts, specific read-authority should exist. The Security Filter ($\mathcal{M}903$) controls what information can go into the schema cache and what updates can be performed by this module ($\mathcal{M}211$). For these routines, the schema information comprises the data, and the knowledge that is embodied in the authority versus request matrix.

General categories for information to be retrieved by the submodules are:

- High-level data semantics, reference to application semantics, fuzzy knowledge
- Presentation formats, data labels, units definition
- Mapping of relation names to files, views, interfile constraints, interfile dependencies, optimization information, cardinalities, etc.
- Reorganization information
- Logger directives, access path specification, field representation formats, indexing directives, physical allocation directives for entire file or its fragments.

With the current design definition, the service layer $\mathcal{L}3$ does not normally require schema information. Rather, all schema information has been imbedded in the operation tree by the View Manager ($\mathcal{M}301$).

C. No state information is retained within this set of modules between schema change requests. However, during a schema change request transaction, the state of the operation is maintained until completion.

4.$\mathcal{M}$212. Reorganization Manager ($\mathcal{L}$3)

A. The reorganization module is invoked by the Schema Language Executor ($\mathcal{M}211$) whenever organizational transformations are made to the stored data. Typical operations invoking this module are: adding or removing attributes to relations, changes of access path configurations, and changes of allocation and replication of distributed files or file fragments.

Two principal versions of this module can exist:

1. Reorganization initiated by the database administration, and
2. Reorganization driven by internal performance monitoring possibly at the request of the database administrator.

Reorganization maintains the same logical view of the database, and hence exists at a relatively low level.

B. The knowledge of the required logical state, as well as of any mandatory physical constraints (sequentiality, indexes) is maintained in the schema table maintained by module $\mathcal{M}201$. All data manipulations are performed through the File Access Module ($\mathcal{M}101$). Module $\mathcal{M}201$ will update parameters of the schema at the request of Module $\mathcal{M}211$ when reorganization has been completed and right before the reorganization transaction commits.

Reorganization needs to be inhibited when there are active transactions on the relevant portions of the database. A request to the Transaction Manager ($\mathcal{M}601$) is needed to verify such state and inhibit interfering subsequent transactions. Reorganization will make demands on the Log Manager ($\mathcal{M}621$) and the Lock Manager ($\mathcal{M}602$), those requests are mediated through the Transaction Manager ($\mathcal{M}601$).

C. No internal state is maintained by the reorganization module beyond the span of a reorganization task.

4.$\mathcal{M}$221–$\mathcal{M}$229. Schema Language Interfaces ($\mathcal{L}$5)

A. These modules translate the schema specification to the format required by the Schema Language Executor ($\mathcal{M}211$). The manner in which these specifications are made differs greatly among DBMSs, so that we allow for multiple, and possibly parallel modules. A DBMS-independent proposal for a schema language is found in [Qian 85].

Data definition includes both logical concerns, as represented in data models, and physical concern, for instance data representation, access path maintenance, and data distribution.

In systems that permit view information to be updated, a disambiguation step is required, since the view that a user has is only an incomplete view of the database. Rules to interpret view updates can be provided by the database administrator [Kell 86] to a specialized Schema Language Interface ($\mathcal{M}221$). View capability is important not only for ease of user access but also to guarantee flexibility of data reorganization.

B. Data definition translators obtain information about the current schema table contents from module $\mathcal{M}201$. They transmit information to update the schema to the Schema Language Executor $\mathcal{M}201$ when a new or revised schema is to be installed. A reorganization, controlled by the Reorganization Manager $\mathcal{M}212$, may be invoked at that time.

C. Internal information is kept only while a new or revised schema is being developed; afterwards this information becomes available through the Schema Table Manager ($\mathcal{M}201$).

4.3. Database Processing

We provide a set of functions which carry out common data processing services for the user interface sensitive modules at layer $\mathcal{L}5$. These routines manipulate data in the high bandwidth flow. These routines are also entered for service requests from remote sites, arriving at the Distributed Access Server module ($\mathcal{M}721$). Direct user access is also permitted into layer $\mathcal{L}4$, but no *user-friendly* interface should be expected.

Some DBMS interface programs (some of $\mathcal{M}401$–$\mathcal{M}439$, $\mathcal{M}501$–$\mathcal{M}539$) included in the base systems can bypass the Relational Algebraic Access module ($\mathcal{M}451$) and communicate directly with the View Manager ($\mathcal{M}301$) at layer $\mathcal{L}4$.

We believe that many functions performed on data by different database management systems are similar. These functions need not be intrinsic to the database modules identified as $\mathcal{M}451$, $\mathcal{M}401$–$\mathcal{M}439$, or $\mathcal{M}501$–$\mathcal{M}539$ and they can be usefully separated. We identify the functions with the well-known operations of the relational algebra.

One motivation for the separation of these functions into two modules ($\mathcal{M}311$ and $\mathcal{M}312$) is the need to interpose a Security Filter ($\mathcal{M}903$), which operates on selected data going to or coming from single relations.

The View Manager ($\mathcal{M}301$) provides the interface that enforces database integrity. The Query Execution Planner/Optimizer ($\mathcal{M}302$) is a rather large module that has a rich internal structure only partially described in this paper.

4.$\mathcal{M}$301. View Manager ($\mathcal{L}$4)

A.The View Manager ($\mathcal{M}301$) implements database integrity and translates operations on views into operations on the database. Thus, Module $\mathcal{M}301$ implements the *Integrity Boundary*.

The View Manager ($\mathcal{M}301$) inserts into the operation tree multi-relational integrity enforcement. It also translates queries through views into queries on the underlying database. Updates through views are translated into specialized database operations in the operation tree that understand how to perform view updates operations directly on the database [Kell 85]. These specialized database operations are performed by the Single- and Multi-Relation Operation Executors ($\mathcal{M}311$ and $\mathcal{M}312$).

Note that we do not use the view capability simultaneously as a security mechanism. We do, however, allow views to be used to assist in maintaining database integrity, particularly for join views. The query transformation part of security and access protection services, while programmatically similar, are performed under a different control mechanism on level $\mathcal{L}4$ by module $\mathcal{M}921$. The placement of the view service in modules $\mathcal{M}301$ and $\mathcal{M}312$ in layer $\mathcal{L}4$ implies that it is a service at a higher and optional level in comparison to the security enforcement modules $\mathcal{M}901$–$\mathcal{M}903$. Under a philosophy where views and security are integrated (IBM System R or Ingres), the view modules would be forced to level $\mathcal{L}3$, required to be within the trusted system boundary, and less accessible for development and improvement. View processing is still in a stage where research work is necessary to adapt databases to user needs.

B. The View Manager derives its knowledge from the information provided via the Schema Table Manager $\mathcal{M}201$. View specifications are provided by the database administration using one of the Schema Language Interfaces ($\mathcal{M}221$).

C. It may be necessary to maintain view state internally for a user to present complete and consistent information. Not much experience exists dealing with suitable user-oriented view models.

4.$\mathcal{M}$302. Query Execution Planner/Optimizer ($\mathcal{L}$4)

A. This module performs the optimizing functions for the system, unless it is bypassed by users wishing to control access sequences directly. Logical descriptions (in terms of relations, tuples, and attributes) are mapped here to physical configurations (files, records, and fields). View transformation and integrity constraints have already been inserted into the operation tree by the View Manager ($\mathcal{M}301$); module $\mathcal{M}302$ reduces the cost incurred.

The output of a query optimizer is a schedule of file access operations, to be executed by the execution routines ($\mathcal{M}311$ and $\mathcal{M}312$). In simple environments, where few accesses use multiple files or remote information, this module may be minimal.

B. The required knowledge about the data resides within the schema. The optimizer contains procedural knowledge of optimization strategies. Optimization decisions require knowledge of the physical data configuration: location of files, access paths, etc. Knowledge about actual data location and assignment is provided via the Schema Table Manager ($\mathcal{M}201$). Most data specifications are provided by the data administrator using Module $\mathcal{M}221$ to update the Schema Table ($\mathcal{M}201$).

An optimizer for a distributed system also requires knowledge about data allocation and replication, as placed into the schema at the request of the Schema Language Executor ($\mathcal{M}$211). Information about the size of files is associated with the schema and typically updated periodically.

Advanced optimizers also want information about data value distribution. These flows are not included now.

C. An optimizer can maintain transient information about buffers and data caches likely to be available. Reuse of recently retrieved information can increase the efficiency of query processing greatly [Fink 82].

4.$\mathcal{M}$311. Single Relation Operation Executor ($\mathcal{L}$3)

A.The Single Relation Operation Executor performs the selection operation on an individual relation, resulting in a record stream. The *selection* operation uses available indexes implemented by the File Access Module ($\mathcal{M}$101) so that only subsets of the data have to be passed through it. Any restrictions due to *join* processing is also processed here. There is also the *filter* operation, which implements selection on a record stream. Projection is assigned to Module $\mathcal{M}$312 for reasons discussed there.

One motivation for the separation of these functions into two modules is the need to interpose a Security Filter ($\mathcal{M}$903), which operates on data going to or coming from single relations after the selection has been performed. Specifications for Module $\mathcal{M}$311 have been completed [YuLi 88].

Modules $\mathcal{M}$311 and $\mathcal{M}$312 are also responsible for performing specialized database update operations that implement view update requests [Kell 85].

B. The operations are executed in the sequence specified in the operation tree passed by the caller. Operations may select single records or sets of records. The data is provided by the File Access Module ($\mathcal{M}$101) of layer $\mathcal{L}$2.

All information from the schema that is needed by this module is placed in the operation tree by the View Manager ($\mathcal{M}$301) and the Query Execution Planner/Optimizer ($\mathcal{M}$302).

C. Internal state is maintained only to assure correct completion of the single relation accessing requests.

4.$\mathcal{M}$312. Multi-Relation Operation Executor ($\mathcal{L}$4)

A. Module $\mathcal{M}$312 performs *projection*, the catenation processing after restriction due to any *join*s, and the operations which combine data: *union*, *intersection*, and *difference*. The *projection* operation is assigned to Module $\mathcal{M}$312 for two reasons: projection may require a sort to elide duplicate tuples, and the security filter may need to access the values to be projected out for security predicates. This assignment affects performance only where projection drastically reduces result sizes. Specifications for Module $\mathcal{M}$312 have been completed [YuLi 88].

In some configurations of database usage, for instance by some KBMSs ($\mathcal{M}$521), joining operations may be performed within the calling modules. In those cases, it may not be necessary to include all functions of $\mathcal{M}$312. However, view processing by Module $\mathcal{M}$301 may insert join operations that were not known to modules such as $\mathcal{M}$521.

Modules $\mathcal{M}$311 and $\mathcal{M}$312 are also responsible for performing specialized database update operations that implement view update requests [Kell 85].

B. These operations are also executed in the sequence determined by the calling routines. Operations expect sets of records. Local data is provided via Module $\mathcal{M}$311, remote through the Distributed Access Mediator ($\mathcal{M}$701).

Update data are provided by modules as $\mathcal{M}$501, $\mathcal{M}$401, and $\mathcal{M}$451, and are mediated by the View Manager ($\mathcal{M}$301).

C. Internal state is maintained only to assure correct completion of the single relation accessing requests.

4.4. Database Management

This group encompasses all the database model-sensitive routines, resulting in a great variety of modules. Through these modules, the primary high level programmed access takes place, especially for routine update and report production. These modules provide the user interface commonly associated with commercial DBMS products. These processors can come in many flavors; hence we do not provide a detailed list of alternatives. Some modules will be part of a initial installation expected to be supplied with an Ada DBMS; others will be supplied by users as needed.

4.$\mathcal{M}$401–$\mathcal{M}$439. System-provided and User-provided DBMS Interfaces ($\mathcal{L}$5)

A. No full description will be given here, but some general classes are recognized. We use $\mathcal{M}$401–$\mathcal{M}$419 to designate modules which are part of a typical DBMS; for instance, a module to handle the developing SQL standard. Other modules will be provided by user organizations to suit their own needs; we assign $\mathcal{M}$421–$\mathcal{M}$439 for these module versions.

B. The modules deliver data to end users or end-user programs. The data is obtained from the databases and presented according to the knowledge encoded in the schema.

ACCESSING THE DATABASE Data requirements for these modules are satisfied either by an Algebraic Access module ($\mathcal{M}451$) at layer $\mathcal{L}5$ or by direct access to the View Manager ($\mathcal{M}301$) at layer $\mathcal{L}4$. Such access must be mediated for security protection by the Security Filter ($\mathcal{M}903$). All commands are also passed through the Security Query Transformation module ($\mathcal{M}921$).

A Query-by-Example processor can generate relational algebra statements [Ling 85], and thereby use the Algebraic Access module ($\mathcal{M}451$).

We also partition the modules into two sub-types depending on their information access and storage requirements. We assign here module numbers $\mathcal{M}401$–$\mathcal{M}409$ and $\mathcal{M}421$–429 to indicate compiled access to the database, and $\mathcal{M}411$–$\mathcal{M}419$ and $\mathcal{M}431$–439 interpreted access to the database. The data and information needs of these various module types are similar but they differ in binding time.

For production tasks, compiled versions of these modules will be preferred. Here the knowledge is embedded at compile-time into the module. We show an Ada-SQL and QBE as typical of current approaches to compiled access, and SYSTEM 2000 as typical of mainly interpretive access. Note that any of these languages can be implemented in either fashion. An example is provided by [Frie 86] and the subsequent discussion [FrKe 85].

ACCESSING THE SCHEMA Compiled access is the preferred mode for routine data-processing. Information from the schema is incorporated at compile-time and controls the actual code generation process. This is the mode used for instance by IDS, the current mainstay of the WWMCCS systems.

In interpreted mode, the schema contents are used during query processing. The result is a simpler implementation and more flexibility, with loss in performance when operations are routine. Interpreted modules can yield a higher level of performance when the database is very volatile in size or locality.

C. Once a module is compiled, it maintains internal state. In compiled mode, any changes in the schema require recompilation of the programs. In SYSTEM-R such recompilations are initiated automatically as required [CAKL 81].

An interpretive module does not keep internal state; all of its control information is obtained, as needed, from the schema table via the Schema Table Manager ($\mathcal{M}201$).

4.$\mathcal{M}$451. Algebraic Access ($\mathcal{L}$5)

A. This module is envisaged as a very thin layer, and provides a protective interface for commands being passed from layers $\mathcal{L}5$ or $\mathcal{L}6$. This protection is important for calling modules which are developed outside of the control and detailed specification of this overall DBMS design. Specifications for a prototype Algebraic Access module have been completed [Lin 88].

Data is not to be moved here, but handed directly to the View Manager ($\mathcal{M}301$) at layer $\mathcal{L}4$. Note that this interface was entirely omitted in our prototype specification.

B. The Algebraic Access module is driven by commands from user interface modules as $\mathcal{M}421$ or $\mathcal{M}521$. These are verified using knowledge obtained from the Schema Table Manager ($\mathcal{M}201$) and converted into an efficient internal format for the routines which execute the relational operations, modules $\mathcal{M}311$ and $\mathcal{M}312$.

For most of the commands, any references for the data are passed through with the commands. The data references are set-oriented, allowing large granules to be handled per command transfer. The commands are passed to the View Manager ($\mathcal{M}301$), which performs view translation and implements database integrity.

C. No internal state should be kept by this module.

4.5. High-level Processors

The programs at this level have application and system knowledge which permits them to serve users in an *intelligent* fashion.

Tasks emphasized at this layer are:

- Simple, non-procedural query formulation.
- Presentation of information at a terminal.
- Data reduction.
- Multifile access optimization using application knowledge.

Issues of compilation versus interpretation follow the distinctions made in modules $\mathcal{M}401$ and $\mathcal{M}411$.

4.$\mathcal{M}$501-$\mathcal{M}$519. Query Processors ($\mathcal{L}$6)

A. A simple user module of this type is the proposed WIS Database Interaction Tool (DIT) [Berk 85a]. It consists of a query interpreter and report generator. Models for the facilities needed for such systems exist on personal computers, as Q&A by Symantec, SEQUITUR by Pacific Software, and PFS-FILE. On larger machines these services are approximated by ANSWER-DB, STAIRS of IBM, and MARK-IV of Informatics.

B. Data accesses made by query processors typically bypass the intermediate query-language processors provided at layer $\mathcal{L}5$. Their need is for direct procedural internal interfaces as provided by the View Manager ($\mathcal{M}491$).

Knowledge for the performance of these tasks is again contained in the schema $\mathcal{M}201$. It may come from a variety of sources.

C. Information from previous transactions may be kept during a user session for merging into final presentation. Time stamps should identify retrieved data to permit recognition of access asynchrony and the possibility of inconsistency.

4.$\mathcal{M}521$-$\mathcal{M}539$. Knowledge-based Modules ($\mathcal{L}6$)

A. Knowledge-based systems process and reduce aggregations of data using artificial intelligence techniques to deal with complex relationships and uncertainty. They are oriented towards aiding in decision-making and planning. Examples of knowledge-based modules are found in [Wied 84].

B. A KBMS obtains knowledge about the database from the schema via Module $\mathcal{M}201$ and may store *private knowledge* within the database itself. The use of private storage relieves the database administrator for responsibility of information which is expert knowledge, and often tentative and uncertain. These modules will use services of the query routines $\mathcal{M}401$,... at layer $\mathcal{L}5$ or the Algebraic Access module ($\mathcal{M}451$). They are able to deposit knowledge gained in processing into the private storage. Knowledge gained that is of general value can be stored in private storage and can then be accessed by a DBA and used to upgrade the schema accordingly.

Knowledge of a specific application can help these modules to access the database in an efficient fashion. We expect these modules to often skip accessing modules at layer $\mathcal{L}5$. A major reorganization could affect such efficiency, however. We believe that once operational stability is achieved, the associated efficiency of access at this layer can be of sufficient benefit.

C. Knowledge-based routines require a great deal of semantic information. Some of the information can be effectively kept with the schema, other information is maintained internally and, as needed for persistence, stored in designated database files.

4.6. Transaction Management

Transaction management implements control of user interactions. There are two functions: checking for and avoiding interference of concurrent transactions, and making scheduling decisions when interference occurs.

In the proposed architecture, we have scheduling decisions performed independent of concurrency control, which must interrupt transactions and then invoke the scheduler module at the operating system layer. Our DBMS is now decoupled in an important sense from the scheduler. Making a *wrong*, that is, a non-optimal decision in the scheduler, will only affect system performance, not correctness of the result.

In many treatises on concurrency control, the scheduling task is embedded within the concurrency control code [BrHa 72]. Embedding means that the transaction scheduler $\mathcal{M}601$ has to be modified when alternate scheduling strategies are needed, and this can affect reliability. The Transaction Manager ($\mathcal{M}601$) is a much larger module and critical to system correctness; hence we consider such a separation essential.

Logging is the function which writes persistent redundant data for recovery from system, transaction, and user failures. Its requirements, and hence its activities, differ greatly depending on the environment.

In a bank, logging occupies a major fraction of the resources. In a volatile and real-time environment, little logging may take place since, for signal or data acquisition, the cost and volume of data logging may be greater than the benefit. Logging may demand much storage bandwidth and capacity. There are database systems without logging, in a workstation that is in fact typical.

Two types of recovery operations must be supported:

- *Rollback:* backing out aborted transactions, or undoing all the effects of erroneous transactions which were already committed.
- *Restoration:* restarting the system using a backup file, and then applying all subsequent changes to the file. Backups should hence be created regularly. Restoration is not described in this paper.

4.$\mathcal{M}601$. Transaction Manager ($\mathcal{L}3$)

A. This module controls multi-user access and prevents interference of transactions. The protection extends from layer $\mathcal{L}2$ down; we expect that the top layer will support long sessions for which traditional transaction management is not appropriate. This module is another component of the service layer ($\mathcal{L}3$). It will be minimal, perhaps even omitted, in situations where there is:

a) single user (workstation) operation
b) no updating
c) only one active process with update privileges

The primary task of this can be implemented in two distinct styles: optimistic versus pessimistic concurrency control.

- O Using the *optimistic* style a permission to access is granted until an access conflict is detected. When an access conflict is found, the Transaction Manager invokes the Scheduler ($\mathcal{M}$611) for a priority decision. Then the Recovery Manager ($\mathcal{M}$622) is invoked to restore the database to an earlier version.
- P Using *pessimistic* style synchronization, the internal tables are scanned for every request and the executor is delayed until safe execution can be assured.

The information flow available to this module must satisfy either approach. A separate report expands on the tasks expected from an Ada/DBMS transaction module [Cher 85].

In a locking paradigm, the Transaction Manager performs deadlock detection using information maintained by the Lock Manager ($\mathcal{M}$602).

B. The transaction management module is invoked by three sources. Most invocations will be for commands recognized by the executor module $\mathcal{M}$311 as being relevant for transaction management. When a data reorganization occurs ($\mathcal{M}$212), an internally initiated transaction must be scheduled. Finally, a DBA interface to this module has two functions: initiating an abort of a transaction where control has been lost and initiating a dump transaction.

Backups require a quiescent state of the system to assure consistency of the backup data. In very large database systems, incremental dumping and subsequent backup file synchronization may have to be supported by this module.

The Transaction Manager ($\mathcal{M}$601) explicitly calls the Log Manager ($\mathcal{M}$621) to record system status for rollback, recovery, and audit. It provides identifying data and timestamps for this task.

C. The transaction control module must maintain the following information internally:

1. A list of all active users or transactions
2. A record of all objects locked for access by users or transactions
3. The type of access (read, write, or perhaps increment)

4.$\mathcal{M}$602. Lock Manager ($\mathcal{L}$1)

A. The Lock Manager performs concurrency control for shared database access. In a pessimistic concurrency control approach, locks are used. The same interface may be used by an optimistic concurrency control approach that does conflict checking at the commit point.

Locking is done by symbolic name on three different namespaces based on the class of item locked:

- File identifier for file-level lock
- Block and file identifiers for physical block-level lock
- Record and file identifiers for logical record-level lock
- Field, record and file identifiers for field-level lock

Block-level locks are maintained by the Block Manager ($\mathcal{M}$102) because the buffers are also a shared resource that must handle concurrent access. We do not yet envision implementation of field locks, but include them because they can be added into this module easily if desired.

Calls exist for marking the start, commit, abort, and end of transactions. Nested transactions are also supported. The Lock Manager may immediately return in case of failure to acquire lock or it may wait until the lock is available. The Transaction Manager ($\mathcal{M}$601) performs deadlock detection using information maintained by the Lock Manager.

B. The Lock Manager is called by the File Access Module ($\mathcal{M}$101) to lock files (primarily at open/close) and records (between the operations of a transaction). The Lock Manager is also called by the Transaction Manager ($\mathcal{M}$601) for the start, commit, abort, and end of transactions, and also for deadlock detection. The Lock Manager is also called by the Reorganization Manager ($\mathcal{M}$212) to lock entire files during reorganization.

C. The Lock Manager needs to maintain a shared lock table that is accessed for all users.

4.$\mathcal{M}$611. Scheduler for Multiple Users ($\mathcal{L}$1)

A. The scheduler module implements the planned operating mode for the DBMS. In an Ada environment the function is delegated to the task management package associated with an Ada sensitive operating system. The scheduler embodies priority rules for scheduling, and resolves conflicts found by the concurrency control embodied in the transaction management module $\mathcal{M}$601. The operating system layer is appropriate for the scheduler, since scheduling involves knowledge and assessments of the availability of global resources.

Much of the scheduling knowledge is encoded within the module, and alternate strategies are inserted by replacing this module. Since this module is invoked at times of conflict, it cannot make demands on any resources (files, input or output, additional memory) which may be scarce.

Alternative scheduling strategies for a DBMS are:

1. Fairness to all users—timesharing.
2. Minimal conflict—priority given to users holding scarce resources.
3. Time-based scheduling—deadline-driven processing.
4. Priority—user identification determines assignment.

High-performance, transaction-oriented systems tend to use strategy 2. Mixed approaches are valid in certain applications but rarely available today, leading to DBMS inflexibility.

B. The results of the scheduler are simple decisions within the transaction control mechanism of module $\mathcal{M}601$. Some of the knowledge needed is transmitted with the invoking call. The remainder is obtained from the operating system.

C. We see no internal state being maintained in this module.

4.$\mathcal{M}621$. Log Manager ($\mathcal{L}3$)

A. The logging module $\mathcal{M}621$ must provide at least the information needed to enable the capabilities required by the transaction control module $\mathcal{M}601$. For instance, an optimistic concurrency protocol requires a persistent log of all before-images of transactions which have not yet been committed or aborted.

Distributed concurrency control is best accomplished using versions of data [BeGo 82], and here the logging module has to maintain persistent versions of past transactions which follow any transaction not yet expunged from the system. Replication of data, controlled by the access planner $\mathcal{M}302$ and executed by the distribution server $\mathcal{M}711$ to distributed computers also changes the requirements, reducing data logging but increasing the complexity of data identification.

There is also device dependency. In paperless systems, logging requirements increase. When optical media with intrinsic version capability [Rath 84], are used, less logging is needed than for magnetic disks. For optical disks, logging may be eliminated in favor of periodic backup (dump commands) as signaled by the database administrator issuing dump commands via the transaction manager $\mathcal{M}601$.

Specifications for the Log Manager have been completed [Kenn 88b].

B. The logging module provides data for the Recovery Manager ($\mathcal{M}622$) and audit modules $\mathcal{M}801$. Control knowledge for logging decisions is based on the callers to the Log Manager.

The data being logged will be labeled by information provided by the Transaction Manager ($\mathcal{M}601$) as transactions are initiated. The initialization information for a transaction is provided with the command text, transmitted from the executor module $\mathcal{M}311$. Each data block being emitted or received is identified with a transaction identifier.

In our architecture, the logger obtains the data from the flow of data records between the executor $\mathcal{M}311$ and the file access routine $\mathcal{M}101$. This flow includes data identification and content. Positioning the logger $\mathcal{M}621$ at this point in the flow means that the logger accesses records, rather than blocks. This follows current conventions [Gray 81], but does not permit file restoration using after-images from media crashes. It also does not protect information loss due to failures of the $\mathcal{M}101$ and $\mathcal{M}102$ modules, which could damage other data in the blocks obtained.

A restoration in our architecture requires replay of past transactions. Backup or dump functions are invoked by the transaction management module $\mathcal{M}601$ automatically, or in response to commands issued by the DBA.

In an alternative architecture, the logger obtains data blocks from the flow between the file access routines $\mathcal{M}101$ and the Block Manager $\mathcal{M}102$. The logging volume is larger and some blocks may contain data from several active transactions. Such an architecture can recover more effectively from media damage and software disasters.

The logger output bypasses the operating systems buffer management $\mathcal{M}102$, since it writes with high priority, typically to dedicated devices. It is critical that all schema file alterations be logged as well. Data to be logged comprise five types:

1. Transaction begin, commit or abort, identification, and timestamp.
2. Before-images (records) for rollback restoration from aborted or erroneous transactions.
3. After-images (records) for insertion restoration from system crashes other than media failures.
4. Message journal containing formal query text and responses for operational replay and audit.
5. Transaction thread identifying both blocks and records read and written.

There is redundancy in this list, and rarely all five types of data are logged.

Logging may also differ by application within one system. However, since it is the data which must be protected, in particular the data which are owned by an application, the required information for specific logging methods can be kept in the schema $\mathcal{M}$201 and obtained from a schema interpreter.

C. The logger should not maintain internal state, so that its operation is unaffected by system failures. All its information is within the files that is used to log the information. This assures one aspect of *idempotency* as defined by [Gray 81].

4.$\mathcal{M}$622. Recovery Manager ($\mathcal{L}$2)

A. The recovery manager repairs damaged databases and assists in the recovery from transaction aborts for optimistic transaction protocols. Recovery management is initiated either

1. Automatically by the operating system during its recovery from a detected crash.
2. Automatically by the transaction management module $\mathcal{M}$601 to effect restoration of aborted transactions.
3. Manually by someone designated by the DBA when a software crash is presumed.
4. Manually to recover from the effect of erroneous transactions.

Recovery management runs similar to an application, possibly with high priority given by the scheduler $\mathcal{M}$611.

B. The decision about the type of recovery needed is provided by the caller. All other knowledge is obtained from the files written by the logging module $\mathcal{M}$621. These are accessed directly, using module $\mathcal{M}$103. No use is made of the schema. Recovery may need error status information from the operating system and the file access system module $\mathcal{M}$101.

For media recovery, the operating system is first called upon to provide a replacement. Recovery only uses information and data stored by the logger to effect correction. It will not cause the writing or alteration of log records itself to assure its own ability to recover from subsequent crashes. It uses the file access system ($\mathcal{M}$101) to correct the data files.

C. It maintains internal state during a recovery operation. This information must not cause the recovery module to miss recovery steps if a failure occurs during recovery.

4.7. Distributed Databases

Support for distributed databases is an important aspect of database system design. Several of the modules are affected by distribution, such as the Schema Table Manager ($\mathcal{M}$201), and there are four modules specifically for distributed databases. The Distributed Access Mediator ($\mathcal{M}$701) encapsulates requests for access to remote data. These requests are passed to the Distributed Access Driver ($\mathcal{M}$711) for transmission to the remote site. At the remote site, the request is received by the Distributed Access Server ($\mathcal{M}$721), which processes the request for execution at that site. For federated, autonomous databases, such as public databases, the Federated Access Mediator ($\mathcal{M}$702) performs the necessary transformations at the request of the Distributed Access Mediator ($\mathcal{M}$701) for communication using foreign database protocols.

4.$\mathcal{M}$701. Distributed Access Mediator ($\mathcal{L}$4)

A. Distributed access is intended to be an integral portion of an Ada-DBMS. Four modules mediate the interfaces. This module, $\mathcal{M}$701, receives requests for remote services and possibly data for updates. The Distributed Access Driver module ($\mathcal{M}$711) packages requests for transmission to remote sites. The Distributed Access Server module ($\mathcal{M}$721) receives these requests and starts local execution of them. The Federated Access Mediator ($\mathcal{M}$702) is responsible for access to public and autonomous database systems. Other modules are affected, especially the Query Execution Planner/Optimizer ($\mathcal{M}$302), which must now recognize distributed allocation if it is to generate efficient access plans in a distributed environment.

We envisage the language for inter-site requests to follow the relational algebra. A distributed DBMS (DDBMS) will maintain a separate super-schema to permit translation of global names to their local equivalent. If transformations of data are required, we propose that a view be defined for that function, and the available facilities of the View Manager ($\mathcal{M}$301), and view update operations implemented in the Operation Executors ($\mathcal{M}$311 and $\mathcal{M}$312), be called on.

The super-schema may be partitioned, so that understanding of requests may require access to schemas at other sites of the distributed DBMS (DDBMS). Multibase of CCA [DaHw 83] provides a model for a passive DDBMS, where no control over the remote sites is exercised and no remote updates can be processed.

Multiple requests may be needed for retrieval from fragmented relations and for updates of replicated data. The View Manager ($\mathcal{M}$301) handles partitions and fragmentation of relations through a view mechanism, and this may be simplified by the Query Execution Planner/Optimizer ($\mathcal{M}$302).

This module also handles distributed deadlock detection in conjunction with modules at the remote sites.

B. This routine is invoked by local requests for processing on remote sites. Requests are sent to the Distributed Access Driver ($\mathcal{M}$711) for transmission to remote sites.

C. This module maintains status information on the current remotely executed transactions and subtransactions.

4.$\mathcal{M}$702. Federated Access Mediator ($\mathcal{L}$4)

A. This module mediates access to remote public and autonomous databases. Restructuring of requests may be needed to handle the formats required by remote sites.

B. This module is invoked by the Distributed Access Mediator ($\mathcal{M}$701) upon requests for remote execution of subtransactions on public, autonomous, or heterogeneous databases.

C. Descriptive information of remote database systems and their protocols are available to this module.

4.$\mathcal{M}$711. Distributed Access Driver ($\mathcal{L}$1)

A. This module mirrors the usage conventions of the primary file access server $\mathcal{M}$101, but transmits the request to other sites. The decision as to which of the two modules to invoke is made by the Multi-Relation Operation Executor ($\mathcal{M}$312) on the basis of file information obtained from the schema. If relations are fragmented, then the executor $\mathcal{M}$312 may have to issue multiple requests to satisfy a select expression. If data are replicated, the $\mathcal{M}$312 executor module must issue any update requests to all the sites involved.

The data request to a remote site will be phrased in a language based on the relational algebra operators: `select` and `project`, and will use local naming conventions. It is the responsibility of the remote site to translate and execute the query. This translation may require a super schema (handled by a routine as $\mathcal{M}$701) and possibly view translation, perhaps generating joins and other operations at the remote site. This means that details of local bindings at remote sites do not have to be made known to requesting sites.

B. This routine only does a formatting transformation on the requests to make them suitable for transmission, and unpacks any records received into the format used within the DBMS.

It can also be invoked by the Recovery Manager ($\mathcal{M}$622) to obtain copies of data, so that the remote site can act as a log. Using replication as an adjunct to logging is essential in situations where a distributed node can be damaged, so that its normally stable storage is lost. Using remote replication can be advantageous as an alternative to local logging on small or mobile sites.

C. This module keeps track of outstanding requests until they have been completed.

4.$\mathcal{M}$721. Distributed Access Server ($\mathcal{L}$5)

A. This module accepts requests from remote sites and submits them to the Algrebraic Access module ($\mathcal{M}$451) for further processing. This module is the point of contact for requests from remote sites, which usually come from the Distributed Access Driver ($\mathcal{M}$711) from the Distributed Access Mediator ($\mathcal{M}$701) or the Federated Access Mediator ($\mathcal{M}$702).

B. This module maintains status information on the current remotely submitted transactions and subtransactions.

C. The superschema information is stored in a file and retrieved as needed. Portions retrieved may be cached internally. The superschema contains:

1. Knowledge provided by the Distribution DBA.
2. Information obtained by this module from the local and remote schemas.

Access to the local schema is mediated by the Schema Table Manager ($\mathcal{M}$201). Remote access uses communication facilities and schema access by the same routine at the remote site, or by its equivalent if the DDBMS includes non-Ada DBMSs.

More work is required to specify all the uses of this information; we envisage a separate set of access modules for distribution information.

4.8. Audit

The audit function monitors the operation of the system. The information gained provides the necessary feedback for long-term system management and control. Individual transaction type performance and transaction use patterns may be monitored. Access to audit information is mediated by security controls; typically the audit users will have a high level of read-only access rights.

4.$\mathcal{M}$801. Administrative Interface ($\mathcal{L}$4)

A. The audit module is invoked by the DBA or the security officer to monitor and verify the operation of the system.

B. As the functionality of the database grows, the Administrative Interfaces takes on importance in auditing the proper operation of the system.

C. No internal state needs to be kept by the audit module.

4.9 Security Enforcement

Security protection for the DBMS requires a number of modules, for which the security table $\mathcal{M}$901 provides the information. The concepts for these modules are described in detail in a separate document [Spoo 86].

Mandatory and discretionary security controls are needed for the Ada-DBMS. Mandatory controls require a rigid separation of access rights and must be implemented within the operating system, and hence are not part of this DBMS design. Any communication request between levels of mandatory security will be treated as if it were a request to a remote computer, and is mediated through the communication subsystem, which has to provide the same protection function for any remote access request.

The Ada-DBMS is responsible for implementing discretionary controls within each security level imposed by the mandatory controls. There are five modules related to discretionary security enforcement:

$\mathcal{M}$901	Security Table Manager
$\mathcal{M}$902	Security Interpreter
$\mathcal{M}$903	Security Filter
$\mathcal{M}$911	Security Manager
$\mathcal{M}$921	Query Modification

The first four of these modules are intended to be part of the trusted computer base (TCB). Together they form a boundary which permits the definition of a secure envelope so that other modules within the envelope need not be trusted. All communication out of this envelope is mediated by these trusted modules, and no other communication is permissible.

The last module $\mathcal{M}$921 is included primarily for performance reasons and is not required for correct security enforcement. Hence, it need not be part of the trusted computer base.

4.$\mathcal{M}$901. Security Table Manager ($\mathcal{L}$3)

A. The security table manager $\mathcal{M}$901 maintains the *authorization matrix*. The authorization table includes the discretionary controls defined in terms of predicates associated with triples of the form ⟨user or clique, data object, operation⟩.

We assume that the predicates take the form of functionals. These functionals, when evaluated, filter a tuple by blanking out attributes of the tuple or supressing the entire tuple.

B. The authorization matrix is readable by a trusted program within the DBMS $\mathcal{M}$902 and writable only by the security officer.

C. This module maintains critical internal state. Logically, it might be part of the Data Dictionary/Directory [Berk 85], however, physically, it should be stored and managed separately to guarantee its integrity. The functionals may be arbitrarily complex and are specified by the security officer using any language developed for this purpose. Once compiled, they are stored in a library and their names used as access privileges in the authorization matrix.

4.$\mathcal{M}$902. Security Table Interpreter ($\mathcal{L}$3)

A. The security interpreter is the interface between the security filter $\mathcal{M}$903 in the executor and the authorization matrix of $\mathcal{M}$901.

B. It accepts requests from the security filter $\mathcal{M}$903 and the security transformation module $\mathcal{M}$921. It returns the set of predicates which define the relevant discretionary controls. Specifically, given the name of a clique, data object, and operation type, the Interpreter consults the authorization matrix for the names of all predicates which must be satisfied. It then retrieves these predicates from $\mathcal{M}$901 and returns them to the requestor. If the data object is a relation, the interpreter returns all predicates defined for the relation as well as any defined for the attributes of the relation.

C. It does not maintain internal state.

4.$\mathcal{M}$903. Security Filter ($\mathcal{L}$3)

A. The Security Filter is sandwiched between the two halves of the Executor $\mathcal{M}$312 and $\mathcal{M}$311 and filters all tuples flowing through it. It first processes discretionary controls defined on the relation of which the tuple is a member. If the discretionary controls are not satisfied, the tuple is passed no higher in the system, and hence is never seen by the user. If the tuple does pass the discretionary controls for the relation, then discretionary controls for attributes of the tuple are checked. If these controls are not satisfied for any attribute in the tuple, that attribute is blanked out. Thus, the value of that attribute is never seen by the user. The Security Filter will similarly filter data in an update query.

B. All the information required is obtained from the security table $\mathcal{M}$901 via the security interpreter $\mathcal{M}$902.

C. No internal state is maintained

4.$\mathcal{M}$911. Security Manager ($\mathcal{L}$3)

A. The Security Manager is the interface the Security Officer uses to change the authorization information in the Security Table maintained by the Security Table Manager ($\mathcal{M}$901).

B. The Security Manager is invoked upon request of the Security Officer and it uses private calls of the Security Table Manager for query and maintenance of the Security Table.

C. It does not maintain internal state.

4.$\mathcal{M}$921. Security Query Transform ($\mathcal{L}$4)

A. For efficiency reasons, we include a query modification module for security predicates for use by the Planner. This module is called by the execution planner $\mathcal{M}$302 when generating a plan for processing a query to modify the plan to retrieve only data which will satisfy the discretionary security controls. This can greatly reduce the volume of data retrieved by a query. It will also allow the planner to project out columns of a relation that will eventually be *blanked out* by the Security Filter ($\mathcal{M}$903).

B. Security query modification is invoked by the query planner $\mathcal{M}$302 and obtains the knowledge for its advice via the security table interpreter $\mathcal{M}$902.

C. It should not maintain internal state.

5. The Future of DADAISM

There are several complementary aspects to future DADAISM development, so that success in one aspect is bound to benefit the others.

First of all, there is the importance of having an Ada-oriented database system available for the community. Only a DBMS which provides effective interfaces for data processing, sensor data acquisition, as well as high-level SQL access can hope to be accepted by managers who are concerned with aggregate system performance and availability. There is no doubt that a well-architectured and developed Ada DBMS will outperform software which embeds large bodies of foreign code and must depend on interfaces oriented towards query languages.

If the DADAISM project is in itself not successful, it will have developed technology useful in successor systems in and for Ada. But any major delay incurred will cause many tasks that could be managed in a DBMS environment to be made operational using a variety of file management techniques instead.

A question to be answered by the DADAISM effort is still the extent to which true modularity concepts can be used in large systems. Traditionally in large systems, unconstrained access to information among program modules was exploited in order to achieve adequate performance. We cannot assume that our architecture specification has captured all the essential information sharing paths. But if we can constrain changes to pairs of modules (information-generating and information-receiving), we will have placed modularization concepts on a much sounder footing.

The ability to separate modules may have an important effect on the software industry. This industry, in addition to some large and well-established companies, includes many bright young entrepreneurs. For these, developing and demonstrating to the market an entire DBMS is not feasible. It is our hope that the modularization can accomplish this goal. Examples where specified interfaces have opened the market abound in software and hardware. For instance, the Interbase Software Company could not get off the ground until it decided to adapt to the DEC VAX Standard Relational Interface.

Last but not least, the development of the DADAISM project satisfies an educational aspect. The decomposition of software into modules, especially with functional specifications in Anna as used in DADAISM, will improve the dissemination of modern software development technology principles. When students can feel that large software projects can be manageable, exciting, and effective, we will gain in an area where now much doubt is expressed. Such doubt in turn drives away many good students and encourages people to expend intellectual effort on tasks not as demanding and critical as programming in the large.

5.1. Expectations from having modules

We expect two major gains in terms of adaptability and reliability from having modules.

1 Having a library of modules permits tailoring of the DBMS to small and large computer systems. We expect for instance single user workstations to become common. Here access controls can be much simpler than on systems where many user share a processor.

2 Distribution of functions can also be eased. We may find systems without major local storage, so that file access and logging functions may reside on other processors.

3 Adaptability over time is enabled. Databases must live for a long time. The ability to replace modules will accomodate change and growth.

4 With well defined modules we also gain an ability to use multiple vendors for development, and can profit from the best new technology.

5 The lower level modules can reduce the problems with inconsistent operating system services at the low level. We always prefer that the OS does much, but many current DBMSs achieve portability by replicating many OS functions.

5.2. Current State of DADAISM Prototype

The architecture described in this paper has been partially formally specified and a partial operational prototype exists. The operational modules implement a single user database system including single tuple update requests. The implemented modules have formal interface specifications written in Ada. Functional specifications are written in Anna, a declarative specification language for Ada developed by David Luckham. These modules have operational, but not necessarily complete, implementations in Ada. Statistics on the number of lines of Ada/Anna specifications and code bodies are in the table. The modules with asterisks in the column for bodies are merely program stubs.

Module	Specs	Body
Block Mgr	950	1500
Log Mgr	550	700
Transaction Mgr	350	550*
File Access	2050	8050
SROE/MROE	3200	6700
Rel Alg	500	3050
Schema Table Mgr	1900	3650
Schema Lang Exec	1100	150*
Load Database	400	650
Demo Program	50	2800
Totals	11,050	27,800

We are using Anna for the functional specification language because a declarative language allows the specifications to be checked as assertions against the operational code. Alternatively, comparing a procedural specification language against an implementation is in general an undecidable problem. Although the toolset for Anna is still under development, there are already tools that allow the specifications to be validated using simulation to determine completeness and correctness. There is also a tool to convert the specifications into assertions that are inserted into the code bodies to validate through program testing that the code bodies satisfy the specifications under the conditions tested. Declarative specifications are also more amenable to program verification once the state of program verification theorem provers is sufficiently advanced. Furthermore, declarative specifications also allow new and improved algorithms to be used to implement modules, and allow the module functionality to be adapted to operational needs while still satisfying the interface and functional specifications. While writing the specifications in Anna is quite an involved process, the additional time spent in design is more than balanced by the reduced time in implementation and program testing.

Our next step is to implement concurrency control and transaction support. This involves continued elaboration of the design in specifications and implementation. Steps after that include implementation of database optimization, query compilation, SQL support, and data distribution and remote access. Once a critical mass of database code bodies are operational, the security modules can be specified and have prototype implementations. A full implementation of the security modules also awaits the existence of a secure Ada run time environment.

In parallel with this continued development of the architecture and implementation, we are considering enhancements to support storage of objects, such as those needed for computer aided software engineering (CASE) tools. This enhancement of functionality will be an important test of our architecture. We will not avoid making more major changes to the architecture if they result in an improved structure for supporting objects. But if the changes to the architecture for object support maintain the overall structure, then our confidence increases that the architecture can support evolving database implementations over several decades.

5.3. Lessons Learned About DADAISM During Its Development

Our modularized design evolved throughout the course of this project. Early in the project under STARS funding we developed an initial modularization diagram, based in part on the work done under WIS. As our work progressed, we found need to make several changes in the general design and in that diagram. This section focuses on the changes to the design that are not completely reflected in the earlier sections, which describe primarily the design for the relational implementation. The reader may find it helpful to refer to the attached modularization diagram when reading this section.

Our design has clarified schema management. While we continue to believe in the importance of a multi-level schema, we have found that the schema is best stored in an integrated format. This permits better management of the schema data, and it ensures that the schema data for the various levels are maintained with mutual consistency.

We also refined the allocation of schema-related functionality to the various modules. The Schema Table Manager now maintains the schema tables stored as relations in the database while it maintains a cache of parts of the schema in active use. The Reorganization Manager changes the data files to correspond to changes in the schema from user Data Definition Language (DDL) requests. The coordination of schema changes and the corresponding data changes is managed by the Schema Language Executor, which ensures the integrity of the database with respect to the schema. One or more Schema Language Interfaces allow users to enter DDL requests in the user's preferred DDL dialect (e.g., SQL, QBE). Any further expansion of DDL capability will require the addition of concepts for entity-relationship, structural, and KRL-type (knowledge representational) models.

We have simplified the security framework somewhat. In particular, we decided that the separation of the Security Filter and the Security Table Interpreter was artificial, so these modules have been combined. We realized that schema data are also subject to security constraints on disclosure, so we moved the Schema Table Manager, which maintains a cache of the schema, below the Security Filter.

Asynchronous input-output is a problem in Ada. We settled on a separation between the Block Manager (originally called the Buffer Manager) and the Asynchronous Input-Output module because asynchrony may or may not exist independently of buffering. In operating systems that provide for asynchronous input-output, we suggest using it directly rather than using Ada's input-output packages. We note that many forms of the UNIX* operating system do not support asynchronous input-output, although the Mach operating system does support this capability.

We moved several modules to lower layers as we came to better understood their functionality and the layering. The Transaction Manager is now part of Layer 2, Object Access (formerly known as File Access). We are considering porting DADAISM to the Camelot transaction system residing on top of Mach, in which case the DADAISM Transaction Manager would provide a higher level of abstraction on top of Camelot's transaction facility. We also moved the Log Manager to Layer 1, System Services, as we recognized this to be a system service rather than a file access function. We observe that the designers of Camelot agree in that they support logging.

We carefully considered how to support objects in the DADAISM design. This resulted in changing the design so that it divides the File Access Module (FAM) into three modules: the Object and Record Manager, the Access Path Manager, and the Nested Segment Manager. This division is based on the three main functions of the old FAM, and the code appears to be easily separable. Part of the reason for the separation is that the FAM has become unmanageable; we have not yet been able to include all the specified functionality in the implementation. In addition, we believe that the separation will facilitate increasing the functionality to support objects.

The lowest level of the new FAM set of modules is the Nested Segment Manager. This module handles a collection of variable length segments that are self-describing and hierarchically nested. Also included is an object identifier (OID) access structure based on the tuple identifier (TID) access structure in the FAM.

Our original design of the FAM always used the index for finding the desired data since the data were not linked together. We found that avoiding the linking of data enabled some algorithms to be designed in a much cleaner form, permitting a simpler recovery algorithm to be used. The algorithms for inserting, deleting, and replacing records were also simplified as a result. The algorithms for sequential access, however, including setting the cursor, became considerably more complex. Consequently, we have decided to link the data in the new implementation of the Nested Segment Manager. Note that this will not affect either the interface or the functionality of this module. We will be able to reuse much of the original FAM code to serve object management, and we will consider the use of other STARS Foundation code, in particular, the RSR routines, to maintain the hierarchies.

The Access Path Manager maintains B-trees, hash tables, and other access paths for the data stored by the Nested Segment Manager using OIDs for references. The lists of nested segments referred to in a B-tree index (e.g., all the objects with the same key for that index) are stored in a separate nested segment. This concept was used in the FAM, and we are adopting it in the formal interface specifications.

The Object and Record Manager (ORM) coordinates access path and record or object access. For example, to find an object using an index, the index is first searched. The results of the index search are used by the ORM to obtain the object from the Nested Segment Manager.

The use of formal functional specifications with Anna has considerably improved the Block Manager. Before these formal specifications were written, we had developed an earlier version of the Block Manager (called the Buffer Manager) that not only contained but a small portion of the intended functionality but also included bugs. After developing formal functional specifications for the Block Manager, we better understood the distinction between functionality and algorithms. Furthermore, the re-specified Block Manager turned out to be easier to implement and much easier to debug.

The Anna tools developed by David Luckham's group were part of the reason for this ease. For example, we were able to change the Block Manager implementation so that it wrote out the buffers only if they had changed. This change improved the DADAISM's operational performance considerably, but it did not affect perceived functionality. Using our testing techniques, we were able to determine quickly that the revised implementation still met the unchanged functional specifications. We must emphasize that this determination was only possible because we used declarative functional specifications that do not specify the choice of algorithms; an algorithmic specification language would have required that the specifications be changed.

Our modular database design has evolved to become a much stronger framework for database implementation under STARS Foundation sponsorship. We believe that the development of a complete set of formal specifications for the modules of a database system has many benefits. Among these is the ability to evolve the system, so that new requirements and technology can be incorporated through the use of alternative code bodies matching the original (unchanged) specification.

Our support for multiple interfaces at differing levels will facilitate the development of reverse-engineered interfaces that permit existing systems to continue to access the data after migration to DADAISM. Inasmuch as we are providing functional specifications, it will be possible for multiple software vendors to develop modules that can be assembled to operate together as a complete system.

A prototype implementation of the complete specifications will be able to serve as a research vehicle for numerous experiments in database implementation. Designers could, for example, experiment with new optimization or concurrency control algorithms. Students learning about database implementation could learn about the DADAISM design and then implement their own versions of a module. DADAISM represents a database system that can share data referenced through either an object-oriented interface or a relational interface. The concepts we have refined in the development of the DADAISM specification for a modular database system in and for Ada appear to have generality that is applicable to other complex, multi-objective software systems.

5.4. Lessons Learned About Ada Software Development

The specification and development of the DADAISM system over many months has provided the implementation team with considerable practical experience with respect to the development of large, Ada-based software systems. Looking back over that experience, we wish to share some general observations about the development process and to provide some recommendations about approaches we found helpful. We hope that our experiences will enable others to develop Ada-based systems more effectively.

Modularization is a key to implementation success. This may sound a bit like "motherhood and apple pie," for who would dispute this statement. We ourselves held a strong belief in modularization before the project started. Nevertheless, we were surprised at the inadequacy of our working definition of proper modularization and at the sensitivity of the our development progress to this factor. We found that some of our modules turned out to be too large, and a further decomposition provided much better definitional clarity and improved manageability for both implementation and maintenance. We cannot stress enough the importance of proper modularity for facilitating system understanding and development ease.

The practical issue is quantification. How can the developer measure proper modularity? Quantification is difficult, but perhaps our experience can provide some guidelines. Modules that proved to be troublesome (and that proved to be more manageable after further subdivision) had about 1500 lines of formal specification, 10,000 lines of code, and a complexity measure of 400 conditionals and 100 loops. Modules that were the least troublesome, on the other hand, had about 750 lines of formal specification, 3000 lines of code, and a complexity measure of 160 conditionals and 70 loops.

Complexity considerations similarly apply to software interfaces; the greater the complexity of an interface, the greater the difficulty in working with it. We found the process of formally specifying software to be helpful in addressing this issue. Not only did this process facilitate the writing of Ada package specifications, but it also provided clues as to the complexity of the interface. Interfaces that became "unwieldy or too difficult to specify" were candidates for redesign. The modification of these interfaces not only clarified the software design but simplified the software as well.

Ada provides a number of very powerful representation capabilities for data and procedures. Unfortunately, we found that considerations of development efficiency dictated that some of these capabilities be used sparingly or only in restricted circumstances. The use of imported data types is a case in point. We found that development difficulty went up exponentially when datatypes are imported through more than one interface, particularly when only type definitions, and not the services, are provided. (An example of a use that causes development problems would be the provision of datatype S in Package A merely by defining it as a subtype of type T provided by Package B.) Although this usage provides a powerful and compact representation, the resulting reduction in human understandability more than offsets the gain.

Again and again we found simplicity to be a key to effective development. Whether simplicity is achieved directly (e.g., by modularization) or indirectly (e.g., by avoidance of constructs that increase complexity) is not so important as the degree of system simplicity achieved. Perhaps human understanding is the key to an effective system development effort. We found that clear (and correct) documentation, code simplicity, and access to experienced personnel with a thorough understanding of the design were correlated with development effectiveness. That is, the benefits of simplicity previously noted may stem from the developers' improved understanding of the system rather than from any improvements in the system per se+.

We also found that porting a system from one hardware platform to another continues to be a problem, despite the fact that Ada provides a mechanism for producing relatively system-independent code. We were quite careful in our development practices to support portability; we knew before we began the specification effort that we would have to demonstrate the DADAISM's operation on two different hardware platforms. Yet, despite Ada's capability and our awareness, we still found that porting the software from one platform to another was a non-trivial task.

A final observation concerns our use of formal functional specifications. We used the declarative functional specification language Anna that has been developed by David Luckham and his group at Stanford University, and this approach turned out to benefit the development effort to a far greater degree than we had expected. We were able, for example, to change algorithms to improve the Block Manager's performance without changing either the interface or the functional specifications. This was only possible because our functional specifications were declarative and did not specify the choice of algorithms. Although the DADAISM project benefitted significantly from the use of the Anna annotation language and tools and from the support received from the Anna project, benefits flowed in both directions. Our experiences in using Anna for an actual Ada development effort provided valuable feedback that will be used to improve Anna.

There are also some development techniques or approaches that we recommend as a result of our experience. The first of these relate to capturing expertise. One of the reasons experienced personnel play such a critical role in a software development effort is that they have a vast store of knowledge that is otherwise undocumented. When this knowledge is lost (e.g., staff turnover, lack of accessibility at crucial times), development effectiveness suffers. We recommend the following two approaches which we found helpful in capturing and disseminating such experience.

a. Paired development—Develop critical modules with programmer pairs rather than with single individuals. The more experienced person of the pair provides a flow of experience to the junior person. Not only does this provide training, but it serves to provide "redundant storage" of critical knowledge.

b. Staff rotation—Implement each module with a team that was not associated with the specification of that module. This serves two purposes. First, the specifier tends to be more careful in the specification process under these conditions, knowing that all the information about the design will have to be communicated through the specification rather than through personal memory. Second, the implementer undertakes the task without the benefit of any hidden or subconscious understandings about the module. This surfaces specification assumptions that were undocumented. As a result, the quality of both the documentation and the software are much improved. We strongly recommend this approach.

A tool we developed that turned out to be quite helpful was the Bit String Manager. This module provided a useful extension to existing Ada data types, allowing us to handle any type of data that might be presented, not just data that were definable at compile time. We believe that system developers will find such a general capability quite useful and recommend that they incorporate some variant of it in their systems.

With respect to the datatype importation problem, we recommend that abstract datatypes and their constructors be placed in a separate package. While a good design can provide the same benefit, it is often difficult to foresee every datatype that might cross an interface. The separate package approach provides a degree of clarity that facilitates development.

At a higher level of abstraction, the type importation issue can be related to a tendency to rely on syntax for higher-level abstract datatypes without the availability of sufficiently explicit semantics. At the higher level, we expect more intelligence from our types, something that syntax alone cannot provide. Packaging new types with subprograms that incorporate operations and further constraints can provide some of the needed semantics. Note that constraints add knowledge and, by making the data objects become more specialized, increase usability. The next step in abstraction may well require that higher level datatypes embody heuristics to improve performance for their specialized tasks.

6. Conclusion

This modularization is only the beginning of the Ada DBMS design. This stage of work has already permitted us to consider the desiderata for a modern DBMS in a more comprehensive fashion than most expositions now available. We have also learned to address the system design process at a high level of abstraction.

This project presents an example of a data-oriented design methodology. In software projects where we expect that a common data structure will serve many application functions, a data semantics-oriented approach may be a better alternative to a top-down design starting from a procedural objective [Stev 82]. In this *data engineering* approach we still proceed top-down, but start from the information requirements.

Procedural specifications do not provide a solid base, especially when we envisage that modules are replaceable. No procedure can generate information unless the required data and knowledge are made available to the module. When data and knowledge are identified initially, then, in most cases, a competent programmer will have few problems in coding the required transformation procedures and generating a successful program [Wied 86, 88].

7. Acknowledgement

This work was supported in part by the STARS Ada Foundation Techology Program under contract N14-87-C-2503 from the Naval Research Laboratory to SRI International with subcontract number C-11922 to Stanford University. This work was done in collaboration with the Program Analysis and Verification Group at Stanford and the Knowledge Base Management Systems project at Stanford, both of which are supported in part by the Defense Advanced Research Projects Agency, Department of Defense, under Contract N39-84-C-0211. We wish to thank Norman Nielsen for serving as Principal Investigator at SRI International and Vladimir Vinea for assisting. Wino Geelen also provided administration for the project at SRI International. Peter Neumann, Matthew Morgenstern, and Teresa Lunt of SRI International helped on security issues and the modularization design. David Cheriton of Stanford University helped with the modularization design and with issues related to distributed systems. David Luckham of Stanford University collaborated on the Anna specifications. The specifications and implementations were done by Ronaldo Amá, Laie Caindec, Akiro Fujii, Charles Don Griffin, David Harden, Linda Ho, John J. Kenney, Elie Kanaan, Dorothee Koch, Moung Liang, Diana Lin, Francisco Locsin, Luis Marmolejo, Hisashi Nakamura, Sung Park Monica Urata, Jim Rhee, Will Rusch, Lun-Shin Yuen, and Melvin Zeledón, with assistance from Arun Swami. We appreciate the assistance in using Ada and Anna from Will Tracz and Doug Bryan. Lance Murray and Frances Krommenhock assisted with integration and porting. We appreciate the advice and encouragement of Col. Joseph Greene and Elizabeth Wald. The report was improved in response to the insightful criticism of Col. Bill Whitaker.

The initial work was supported in part and motivated throughout by the WWMCCS Information System effort, sponsored by the WIS-SPMO Technology Directorate through Contract No. MDA903-84-C-31 to IDA. The authors brought to this work experience that was supported by a variety of resources including DARPA (Contract N39-84-C-211: KBMS). We have also benefitted by work performed by other individuals on the WIS effort, reflected in numerous reports submitted to IDA. We especially want to recognize John Smith of CCA and Fred Friedman of RACOM. This paper is based in part on "Modularization of an Ada Database System," by Gio Wiederhold, Arthur M. Keller, Sham Navathe, David Spooner, Murray Berkowitz, Bill Brykczynski, and John Salasin, which was presented at the Pre-VLDB Conference on August 21–22, 1986 at

Beijing, China and at the conference of the Information Processing Society of Japan on August 29, 1986 in Tokyo, Japan. This report reflects our own concepts and does not represent an official position of any of the sponsors.

8. References

[Ama 88] Ronaldo Amá: "File Access Module—M101"; DADAISM Project Document, April 27, 1988.

[Bato 88] D.S. Batory: "Concepts for a database system synthesizer"; *ACM PODS*, 1988.

[Berk 85a] Murray Berkowitz: "WIS Database Interaction Tool"; Tech. Report, IDA, Nov.1985.

[Berk 85b] Murray Berkowitz: "WIS Data Dictionary/Directory"; Tech. Report, IDA, Nov.1985.

[BDRZ 85] Richard P.Brägger, Andreas M.Dudler, Juerg Rebsamen, and Carl August Zehnder: "GAMBIT: An Interactive Database Design Tool for Data Structures, Integrity Constraints, and Transactions"; *IEEE Trans.on Software Eng.*, Vol.SE11 No.7, July 1985, pp.574–583.

[BrHa 72] Per BrinchHansen: "A Comparison of Two Synchronizing Concepts"; *Acta Inf.*, Vol.1 No.3, Feb.1972, pp.190–199.

[BeGo 82] P.A. Bernstein and N. Goodman: "Multiversion Concurrency Control, Theory and Algorithms"; Harvard Univ. Tech.Rep.TR-20-82, 1982;*ACM-PODC 1*, Aug.1983, abstract.

[CDRS 86] M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita: "Object and File Management in the EXODUS Extensible Database System"; *VLDB 12*, Aug.1986.

[CDFG 83] A. Chan, U. Dayal, S. Fox, N. Goodman, D. Ries, and D. Skeen "DDM: An Ada Compatible Distributed Database Manager"; *IEEE COMPCON*, 1983.

[DaHw 83] U. Dayal and H.Y. Hwang: "View Definition and Generalization for Database Integration in a Multibase System"; *IEEE TSE*, Vol.SE-10 No.6, Nov.1983, pp.628–645.

[CAKL 81] D.D. Chamberlin, M.M. Astrahan, W.F. King, R.A. Lorie, J.W. Mehl, T.G. Price, M. Schkolnick, P.G. Selinger, D.R. Slutz, B.W. Wade, and A. Yost "Support for Repetitive Transactions and Ad Hoc Queries in System R"; *ACM TODS*, Vol.6 No.1, Mar.1981, pp.70–94.

[CDFG 83] A. Chan, U. Dayal, S. Fox, N. Goodman, D. Ries, and D. Skeen "DDM: An ADA Compatible Distributed Database Manager"; *IEEE COMPCON*, 1983.

[Cher 85] David Cheriton: "WIS Transaction Management"; IDA Internal Document, Sept.1984.

[Codd 86] Edgar 'Ted' Codd: "An Evaluation Scheme for Database Management Systems that are Claimed to be Relational";*Second IEEE CS Data Engineering Conference*, Los Angeles CA, February 1986, pp.720–729.

[DCKK 85] David DeWitt, H-T. Chou, R. Katz, and T. Klug: "Design and Implementation of the Wisconsin Storage System"; Software Practice and Experience, Vol.15 No.10, Oct.1985, pp.943–962.

[Dijk 66] E.W. Dijkstra: "The Structure of the THE Multi-programming System"; *CACM*, Vol.9 No.3, pp.143–346, May.1966.

[Dijk 71] E.W. Dijkstra: "Hierarchial Ordering of Sequential Processes"; *Acta Inf.*, Vol.1 No.2, Oct.1971, pp.115–138.

[DLSH 87a] Dorothy E. Denning, Teresa F. Lunt, Roger R. Schell, Mark Heckman, and William R. Shockley: "A Multilevel Relational Data Model"; 1987 Symp. on Security and Privacy, IEEE CS, April 1987.

[DLSH 87b] Dorothy E. Denning, Teresa F. Lunt, Roger R. Schell, Mark Heckman, and William R. Shockley: "Secure Distributed Data Views: The SeaView Formal Security Policy Manual"; A003: Interim Report, July 1987.

[DoD 85] Department of Defense: "Department of Defense Trusted Computer System Evaluation Criteria"; DOD 5200.28-STD, December 1985.

[DoDa 87] Jane E.D. Donaho and Genell K. Davis: "Ada-Embedded SQL: the Options"; Ada Letters, 1987.

[Fink 82] S.J. Finkelstein: "Common Expression Analysis in Database Applications"; *ACM-SIGMOD 82*, Schkolnick(ed), Orlando FL, Jun.1982, pp.235–245.

[Frie 85] Fred Friedman and Bill Brykczynski: "Description of a Rough Draft Ada/SQL Standard"; IDA Internal Document, July 1985.

[FrBr 86] Fred Friedman and Bill Brykczynski: "Ada/SQL: A Standard, Portable Ada-DBMS Interface"; IEEE CS Data Engineering Conference, February 1986.

[FKWB 86] F. Friedman, A.M. Keller, G. Wiederhold, M. Berkowitz, J. Salasin, D.L. Spooner: "Reference Model for Ada Interfaces to Database Management Systems"; *Second IEEE CS Data Engineering Conference*, Los Angeles CA, February 1986, pp.495–514.

[Frie 86] Fred Friedman: "Ada/SQL A Standard, Portable Ada-DBMS Interface"; *Second IEEE CS*

Data Engineering Conference, Los Angeles CA, February 1986, pp.515–522.

[FrKe 85] Fred Friedman and Arthur Keller: "Discussion on Ada/SQL Specs"; IDA Internal Document, Oct.1985.

[FuCT 86] A.L. Furtado, M.A. Casanova, and L. Tucherman: "A Framework for Design/ Redesign Experts"; *First Conference on Expert Database Systems*, Charleston SC, Apr.1986.

[Gray 81] J. Gray, et al., "The Recovery Manager of the System R database Manager" *ACM Computing Surveys*, Vol. 13 No. 2, Jun 1981, pp. 223–242.

[Haye 85] Barbara Hayes-Roth: "BB1: An Architecture for Blackboard Systems that Control, Explain, and Learn about their own Behaviour"; Stanford University Report STAN-CS-84-1034, Dec 1984; *AI Journal*, 1985.

[KaVi 86] Zaynep Kamel and Donald H. Vines, Jr.: "A Virtual Database Interface for Ada Applications"; IEEE CS Data Engineering Conference, February 1986.

[Kell 83] Arthur M. Keller: "Indexed File Access for Ada"; IDA Internal Document, Oct.1983, pp. 138-153.

[Kell 84] Arthur M. Keller and Gio Wiederhold: "Database Tools, Design Issues for Ada"; IDA Internal Document, Sept.1984.

[Kell 85] Arthur M. Keller: "Updating a Database Through Views"; PhD Thesis, Stanford University, 1985.

[Kell 86] Arthur M. Keller: "The Role of Semantics in Translating Views Updates"; IEEE CS Computer, vol.19 no.1, Jan.1986, pp.63–73.

[Kell 87] Arthur M. Keller and Laurel Harvey, "A Prototype View Update Translation Facility," Report TR-87-45, Dept. of Computer Sciences, Univ. of Texas at Austin, December 1987, 11pp.

[Kenn 88a] John J. Kenney: "Block Manager—M102"; DADAISM Project Document, March 26, 1988.

[Kenn 88b] John J. Kenney: "Log Manager—M701"; DADAISM Project Document, May 4, 1988.

[Lin 88] Diana Lin: "Relational Algebra Interface—M301"; DADAISM Project Document, March 26, 1988.

[Ling 85] Kurt Lingel: "A QBE Processor Using the Relational Algebra"; Stanford MS Project, Aug.1985.

[LiZi 87] Barbara Liskov and Stephen Zilles: "An Introduction to Formal Specifications of Data Abstractions"; chapter 1 of manuscript in preparation.

[Luck 87] David C. Luckham: "ANNA: A Language for Specifying Ada Software"; in preparation, 1987.

[Luvo 84] David Luckham and Friedrich W. vonHenke: "An Overview of Anna: A Specification Language for Ada"; Stanford University, Computer Systems Laboratory Technical Report 84-265, Program Analysis and Verification Group Report 26, September 1984.

[LvKO 87] D.C. Luckham, F.W. vonHenke, B. Krieg-Brückner, O. Owe: *Anna, A Language for Annotating Ada Programs*, Springer-Verlag, Lecture Notes in Computer Science, Vol. 260, 1987.

[Luck 89] David Luckham: "Programming with Specifications, an Introduction to Anna"; draft of a book in preparation, 1989.

[Marm 88] Luis G. Marmolejo-Meillon: "Schema Table Manager—M201"; DADAISM Project Document, March 26, 1988.

[McCo 87] L. Scott McCoy: "Interfacing Ada and Relational Databases"; Ada Letters, 1987.

[McRe 87] Mark McNickle and Ann Reedy: "Experiences in Using Ada with DBMS Applications"; Ada Letters, 1987.

[Mend 88] Geoffrey O. Mendal: "The Anna-I User's Guide and Installation Manual"; unpublished manuscript, 20 January 1988.

[Mend 89] Geoffrey O. Mendal, et al.: "Anna Tools User guide"; unpublished manuscript, 1989.

[NiWa 84] John Nissen and Peter Wallis: *Portability and Style in Ada*, Cambridge University Press, 1984.

[Qian 85] Xiaolei Qian and Gio Wiederhold: "Data Definition Facility of CRITIAS"; *Proceedings of the 4th International Conference on Entity-Relationship Approach*, Oct.1985, pp.46–55.

[Rath 84] Peter Rathmann: "Dynamic Data Structures on Optical Disks"; *IEEE Data Engineering Conf.*, Los Angeles, April 1984

[REFN 76] Raj Reddy, L. Erman, R. Fennel, and R. Neely: "The HEARSAY Speech Understanding System: An Example of the Recognition Process"; *IEEE Trans. on Computers*, Vol.C-25, pp.427–431.

[RMS] VAX Record Management Services Reference Manual, No.AA-Z503A-TE, Digital Equipment Corp., Sep. 1984.

[SAME 88] SAME-DC: "SQL Ada Module Extension—Design Committee (SAME-DC) documents," Marc Graham, ed., Software Engineering Institute, Carnegie Mellon Univ., Pittsburgh PA, 1988.

[Smit 83] John Smith et al, CCA: "Database Technology Review and Development Estimate," Int.Report, IDA, Oct. 1983, pp. 155-181.

[Spoo 86] David Spooner, Arthur M. Keller, Gio Wiederhold, John Salasin, and Deborah Heystek, "Framework for the Security Component of an Ada DBMS" (extended abstract), *12th Int. Conf. on Very Large Data Bases*, Kyoto, Japan, August 1986, pp. 347–354.

[STARS 87] "Ada Code Standards" STARS program, December 15, 1987, document 5150-210.

[Stev 82] Stevens,W.P.: "How Data Flow Can Improve Application Development Productivity"; *IBM Syst.J.*, Vol.21 No.2, 1982, pp.162–178.

[Unis 88a] "Reusability Library Framework AdaKNET/AdaTAU Design Report"; Unisys Corp. Defense Systems, Paoli PA, 31 May 1988.

[Unis 88b] "Reusability Library Framework—Software User's Manual"; Unisys Corp. Defense Systems, Paoli PA, 31 August 1988.

[Wied 83a] Gio Wiederhold: *Database Design* 2nd ed.; McGraw-Hill, 1983.

[Wied 83b] Gio Wiederhold: "Databases and Information Management"; WIS Implementation Study Report Volume III - Background Information; Thomas H. Probert (ed.) IDA, Oct.1983, pp.75-137, 154.

[Wiederhold 84] Gio Wiederhold: "Knowledge and Database Management"; *IEEE Software*, vol.1 no.1, January 1984, pp.63–73.

[Wied 86] Gio Wiederhold, Arthur M. Keller, Sham Navathe, David Spooner, Murray Berkowitz, Bill Brykczynski, and John Salasin, "Modularization of an Ada Database System," *International pre-VLDB Symposium*, Beijing, China, August 1986, pp. 202–232.

[Wied 88] Gio Wiederhold: "Data Engineering"; in *Handbook of Software Engineering*, Les Belady and Mike Evangelist, eds., to appear.

[Yang 86] Teri Yang: personal communication.

[YuLi 88] Lun-Shin Yuen and Diana Lin: "Single and Multiple Relation Executor Modules—M311 and M312"; DADAISM Project Document, March 26, 1988.

[Zele 88] Melvin Zeledón-Guzmán: "Schema Language Executor Module—M211"; DADAISM project document, March 28, 1988.