COLLECTED READINGS ON A DATABASE COMPUTER (DBC)

David K. Hsiao
Editor

1979

The Ohio State University

COLLECTED READINGS ON A DATABASE COMPUTER (DBC)

EDITED BY

DAVID K. HSIAO
DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
THE OHIO STATE UNIVERSITY
COLUMBUS, OHIO 43210

MARCH 1979

CONTENTS

Reprint 3 - "DBC - A Database Computer for Very Large Databases," by Banerjee, Hsiao, and Kannan, IEEE Transactions on Computers, C-28, No. 6, 1979 (to appear in June 1979), 16 pages.

Reprint 4 - "Data Network - A Computer Network of General-purpose Front-end Computers and Special-purpose Back-end Data Base Machines," by Banerjee and Hsiao, Proceedings of the International Symposium on Computer Network Protocols, February, 1978, 12 pages.

TO MY FORMER AND PRESENT RESEARCH ASSOCIATES
AND ASSISTANTS ON DBC WORK

FOREWORD

In a gross sense, direct user manipulation of data may be considered analogous to programming directly on an instruction processor, file manipulation may be compared with programming in assembly language, and the use of a data base management system may be likened to programming in a high-level language (HLL). The development of the instruction processors of contemporary information processing systems has been directed more toward the efficient execution of programs written in the most widely used high-level languages than toward the efficient manipulation of information as described, stored, and maintained by current data base management systems (DBMS).

The tailoring of the architectures of most computers toward HLL execution has paralleled the development and usage of high-level languages. Just as fifteen years ago there were still debates as to whether most programs should be written in a HLL, today we occasionally hear debates as to whether file systems should give way to DBMS. The answer regarding DBMS is as clear to this observer now as was the answer regarding HLL in the early 60's, so we must take steps to handle DBMS requirements in a more efficient manner.

Some features have been added to all modern instruction-processor architectures to aid in manipulating data, but all major systems remain inefficient for the task of supporting a data base management system. The processor architecture needed for executing user programs written in a high-level language, and for supporting a complex operating system, is not the same as the architecture needed for supporting a DBMS. The problem is becoming even more acute with the advent of new DBMS models. One possible solution is to create an architecture which is efficient for both DBMS manipulation and high-level language execution. Because of significant differences in requirements for these two environments, the obstacles to creating such an architecture are great--a fact attested to by the lack of any such architectures in existence today.

An alternative solution is to recognize that the differences between the requirements for HLL execution and DBMS manipulation are so great that the way to achieve the best overall efficiency is to use a different architecture for each task. This approach has been taken by several research teams over the last few years, and significant results have been achieved. The scheme generally proposed is to attach to an existing system a back-end processor with an architecture tailored to data base manipulations. This technique appears to have a lot of inherent merit. The significant time lag in the development of DBMS as compared with HLL is the main reason contemporary architectures are much more efficient at handling the latter. It is likely that had the development of data base management systems preceded the development of high-level languages, we would now be considering a scheme to attach to an existing system a front-end processor with an architecture tailored to high-level language execution.

One of the leaders of the investigations into architectures for efficiently manipulating data base management systems has been Professor David Hsiao of Ohio State University. Professor Hsiao has collected this set of nine papers on the general approach to data base computers which he has so successfully pioneered. I recommend the reading of these papers as an excellent tutorial on the subject of data base computers and as a reference source for further readings on other approaches to this topic.

B.R. Borgerson, Director
Research and Technical Planning
Sperry Univac

FOREWORD

Conventional database packages cannot meet the needs of many systems currently nearing completion. Electronic data processing hardware has been decreasing in cost, size and power requirements at a rate of a factor of two every two to three years. In command, control, communication and intelligence systems, the growing capability of digital hardware has made possible greatly expanded data acquisition and processing systems. Although these systems, in themselves are well designed, the increased availability of data is highly desirable. Unfortunately, the combined effect of increasing quantities of data results in longer access times, reduced data security, and increasingly complex problems in data correlation. Conventional database structures would be drowned in the deluge of data from new Navy systems.

The Office of Naval Research anticipated the above problems, and in 1973 began supporting basic research efforts directed toward the design of methods to implement very large databases which would access data rapidly, securely, and which could give the database system designer a very high-level view of the data. To be practical, the Navy required that this database system be implementable from hardware which would be available commercially in the near-term time frame. Finally, it was essential that existing application software written for relational, hierarchical, network or CODASYL type database structures should be transportable to the Data Base Machine without major rewriting.

The specific approach to the above problem which appeared most promising was to design a separate back-end database computer optimized for access to very large databases. The resulting machine, the Database Computer (DBC) uses a synergistic combination of cleverly chosen hardware to achieve the design goals. These structures are described in the various parts of this book.

By the end of 1977, the fundamental structure of the DBC has been specified and analyzed in detail. The primary remaining question was whether such a machine could actually be built, and whether it could meet the practical requirements of a production component of a large system. Questions such as cost, difficulty of maintenance, recovery in the event of hardware failure, and operations in an update-intensive environment, have eventually to be answered.

In an attempt to encourage some computer manufacturers to address the above questions and to prototype the results of the ONR funded basic research, Professor Hsiao visited many companies, described the Database Computer (DBC) and offered his assistance in getting such an effort off the ground.

It appears that Sperry-Univac has responded to this suggestion, and is now in the process of implementing a prototype machine. The Office of Naval Research is pleased at this transfer of technology from basic research to a prototype which will be available to solve the Navy's critical problems dealing with ever-increasing amounts of data.

Leonard Haynes,
Office of Naval Research

PREFACE

The issuance of this collection of reprints is indicative of a successful story. The success involves many people and organizations, ranges over a number of years and covers various localities. The one thing that threads through time, places and individuals towards the success is the notion of database computer.

It is the database computer notion that causes a federal research funding agency (in this case, the Office of Naval Research) to support initially a typical university (i.e., the Ohio State University) for pursuing basic research and study of database computer and to encourage subsequently a major computer industry (Sperry-UNIVAC) for active pursuit of the instrumentation of and prototype work on a database computer.

In 1975, when the research was first started, there were one professor (Dr. David K. Hsiao) and one student (Mr. Richard I. Baum) on the project. By 1979, the database computer research has produced three Ph.D.'s and one M.S. Dr. Baum has since joined IBM Poughkeepsie Lab, Dr Krishnamurthi Kannan is with IBM Thomas J. Watson Research Center and Mr. Fred Ng is with Bell Laboratories. Dr. Jayanta Banerjee received his Ph.D degree just this month. At the present time, the OSU research team consists of, again, a professor (Dr. David K. Hsiao) and a student (Mr. Jaishankar Menon). On the other hand, since 1978 there has been a large fusion of researchers from UNIVAC (Mr. Olin Bray, Dr. John Jordan, and Dr. Harvey Freeman) and a technical supervisor (Dr. George A. Champine). In addition, we have an energetic sponsor from ONR (Dr. Leonard Haynes) and a high-level manager from UNIVAC (Dr. Barry Borgerson). The amount of traffic among Washington, D.C. (ONR), Philadelphia (UNIVAC headquarters), Minneapolis (UNIVAC Advanced System Group), and Columbus (OSU) is heavy.

Despite the large number of people and organizations involved and despite the difficulty of logistics and the passing of time, the database computer notion has been crystalized, conceptual and functional designs of a database computer known as DBC have been advanced and a prototype implementation of a DBC-like database computer has been proposed. In all likelihood, a plan for prototype construction and experimentation is to follow immediately.

The collection of nine reprints is divided into three parts. In the first reprint of Part I, the arguments for a hardware architectural solution to database management are articulated. Both the conceptual and functional designs of the database computer (DBC) are advanced in the second and third reprints. In making the advances, it is pointed out that a viable database computer may have to eliminate the use of staging and memory hierarchies, to provide high-volume and low-cost content-addressable online database store and to utilize concurrently structural information about the database. It also argues for and proposes a design of a well-integrated hardware security mechanism for access control and a clustering mechanism for performance enhancement. In the last reprint of Part I, the use of database computers as back-end machines in a distributed computer network environment is envisioned.

In Part II, there are three sets of papers -- one for each type of database management system software, namely, the hierarchical (e.g., IBM IMS), relational (e.g., IBM System R) and the CODASYL (e.g., UNIVAC DMS 1100). The reprints attempt to show analytically the two most important factors involved in the replacement of existing system software with the new hardware machine. First, is the database computer a sufficiently high-level machine so that much of the existing database management system software can be replaced by the arrival of the database computer? The answer to this question is affirmative. Second, are there difficulties and penalties in transforming the existing databases and applications to the new machine? Transformation of existing databases in conventional format into the new storage format presents no problem; it is only a one-time overhead. Applications, on the other hand, need not be converted or reprogrammed at all. Although there may be no net savings in storage for the transformed database, the machine performance in the execution of typical transactions for the existing applications can be one or more orders of magnitude of improvement. These improvements are evident in hierarchical, relational and CODASYL types of database management.

In Part III, there is one paper. This paper represents the present thinking of a prototype design and configuration of a DBC-like database computer. The aim of the design is to come up with a prototype which can be constructed and completed in nine months. For the prototype machine, a number of experimentations are planned.

This collection of reprints does not deal with the use of DBC for new applications. It also does not provide a closer discussion of the design of the security mechanism, clustering mechanism and post-processing mechanism (such as the joint operation). A number of technical reports are being issued on these topics: they are available upon request. At this point of our endeavor, the collection is mainly aimed to serve as an orderly introduction to the DBC work. We would also like to use the issuance of the collection as a first testimonial to a successful story on DBC work.

G.A.C.
D.K.H.

ACKNOWLEDGEMENT

This copy of reprints is not made or distributed for direct commercial advantage. Its sole purpose is to place scatteredly published and unpublished papers on DBC written in different time periods by many people into a coherent collection for an orderly introduction to and comprehensive understanding of the DBC work.

The permission to copy reprints 1 through 9 is granted by the first author of each reprint. The permission to copy reprint 4 is also granted by the publisher, A. Danthine of the University of Liege and is gratefully acknowledged herein.

PART I

THE ARCHITECTURE OF DBC

Database Computers—A Step Towards Data Utilities

RICHARD I. BAUM AND DAVID K. HSIAO, MEMBER, IEEE

Abstract—**The concept of the data utility as a database system capable of supporting a data model, a large-scale on-line store, and concurrent user access has emerged in recent years. New technology can make the notion of database computers—a specialized hardware system for database use—a viable one. A summary of previous activities and a discussion of the state of the art of database computers is given. An attempt is made to suggest guidelines for future database computer architecture research.**

Index Terms—**Computer architecture, data models, database computers, database engineering.**

I. Introduction

IN RECENT years, the concept of *data utility* began to emerge. A data utility is a centralized, integrated database system which provides shared, concurrent access with security and integrity for a large number of on-line users. The acceptance of this concept was perhaps prompted by the following factors.

1) Major advances in memory and processor technology now make the on-line storage, access, and management of large databases ($>10^9$ bytes) feasible.

2) There is an acute need to build integrated data stores for many applications to eliminate unnecessary data redundancy, to facilitate data sharing, to maintain the integrity and consistency of the data, and to control access to the data.

3) There is a better understanding of how to develop database systems which provide interfaces having a high degree of independence from the physical structure of the database and the underlying hardware.

Database systems enabling users' application programs and users' "views" of the database to be immune from changes to the physical structure of the database are said to be *data independent*. The logical data structures and operations which provide such data independence are called a *data model*. To minimize the number and types of "mapping" information and operations which relate the logical structures of the data model to the physical structures of the database, the system designers attempt to make the physical database structure "close" in some sense to the logical database structure. A major benefit of a data model is that it provides a unified way to specify access to, and control of, the database. Once a query language is defined for accessing the database, it can be extended in a natural way to allow specification of security and integrity constraints [1], [8], [15], [19], [39]. A security constraint identifies which kind of accesses and manipulations may not be performed by one or more users on certain data elements. An integrity constraint indicates how the logical consistency of the database is to be maintained during user access operations.

A data utility must have the characteristics of a public utility. It must be able to support shared, concurrent access while enforcing the security and integrity constraints specified by its users. It must have adequate capacity to handle the needs of its users and it must be reliable, cost-effective, and responsive. To be a data utility, a database system must therefore meet these requirements.

1) It must provide adequate storage capacity. A contemporary large database system must support an on-line database of 1–10 billion bytes in fast access (i.e., nonarchival) storage devices.

2) It must support a high-level data model and an appropriate query and access control language.

3) It must support shared, concurrent access with adequate response time and with security and integrity enforcement.

4) It must be highly reliable.

Software-implemented database systems for conventional computers can ususally meet the first and second requirements without difficulty. The third and fourth requirements are difficult for conventional database systems to meet. Many of the problems of conventional database systems are due, in large part, to the requirement for mapping information and operations. Name mapping operations convert symbolic data names represented by a query into storage addresses where the data named by the query can be found. Since the query language is made of expressions of predicates, it is usually far more powerful than the addressing scheme implemented by the hardware. More involved mapping algorithms are therefore needed. Name mapping algorithms must be highly optimized if they are to perform well. In particular, these algorithms must minimize their secondary storage access requirements for the named data. To accomplish this most name mapping algorithms use very complex auxiliary data structures to aid their operation. These auxiliary data structures are also stored in secondary storage. The complexity of name mapping algorithms coupled with their requirements for secondary storage accesses can compromise both the performance and software reliability of the system.

Manuscript received April 12, 1976; revised June 28, 1976. The work reported herein was supported by the Office of Naval Research under Contract N00014-75-C-0573.

R. I. Baum was with the Department of Computer and Information Science, Ohio State University, Columbus, OH 43210. He is now with IBM, Poughkeepsie, NY.

D. K. Hsiao is with the Department of Computer and Information Science, Ohio State University, Columbus, OH 43210.

Powerful data security and data integrity facilities are a severe performance hindrance in contemporary systems. The most powerful data security and data integrity mechanisms allow specifications to be written in the query language of the system. To perform access operations it is therefore necessary to perform multiple name mapping operations—one for determining the required data and several for determining the data being affected by the security and integrity constraints. The repetitive use of name mapping algorithms to carry out security and integrity has increased considerably the performance penalty of present systems. This is one reason why data security and data integrity facilities are mostly primitive or nonexisting in contemporary systems.

Specialized database computers can be developed to overcome the problems discussed above. To do this, it is clear that such a computer must be designed with two key goals in mind: first, the provision of an addressing scheme which inherently simplifies the name mapping problem and thus simplifies the algorithms needed for name mapping. Second, the provision of hardware which speeds up some or all aspects of name mapping algorithms. An era when it will be possible to seriously consider the design and construction of database computers is upon us. This era is heralded by three important developments: first, the availability of new hardware technology; second, a better knowledge of the algorithms required to realize database systems; third, an understanding of data models for database systems. New technologies will allow dedicated, functionally specialized components to be built for database systems. Once database algorithms and users' needs in terms of data models are understood it will be possible to determine with some confidence where the limitations of conventional hardware lie and to overcome these limitations with new hardware architectures. The impact of new technology on database computer architectures will be discussed in the last section.

II. Past Database Computer Activities

It was recognized quite early that name mapping would be a problem in a data retrieval system and that some form of content addressing would be desirable to ameliorate the problem. This led to the pioneering work in associative memories [32]. The early work in database system hardware tended to be very device oriented; that is, the designs were centered around a single hardware component. Typically, these early machines were composed of a conventional CPU coupled to a memory hierarchy that had an associative memory or associative process at one end of the hierarchy [6], [13], [14], [26], [28]. The associative memories employed by these systems were not originally designed with database systems in mind (e.g., the designs lacked an overall data model) and so these designs generally had to make the database application fit the associative memory devices instead of designing an entirely new system to meet the application. Because of the inherent problems of memory hierarchy management these early systems could not offer much improvement over database systems implemented with conventional hardware.

After the appearance of the above devices it was recognized that much larger associative memories would be necessary to build large-scale database computers. To do this, Slotnick introduced the idea of the logic-head-per-track rotating storage [36]. Such a device could be used to realize a much larger—albeit much slower—associative memory than those used previously. Slotnick's idea was adopted in varying forms by Parker [33], Parhami [31], and Minsky [27]. This work also emphasized the hardware properties of the devices rather than how they could be used to support data models for database systems.

As database systems matured it became clear that associative memories for database management applications would have to support a much more elaborate form of data structures than did earlier devices. The CASSM project [11], [18], [38] was an attempt to build a logic-head-per-track system capable of handling very general data srructures such as hierarchies. This was the first associative memory hardware project to recognize the need for the data structures required for database systems and to design a device from the outset to support them. A general limitation of logic-head-per-track designs is the high cost of their fixed-head disk storage and associated logic. Nevertheless, the utilization of associative memories to minimize name-mapping complexity on database systems was an encouraging one.

To avoid the problems of expensive fixed-head disks, others have tried to build specialized components to augment conventional systems by performing one or more particular database system functions. Most of the work in this area has concentrated on directory memories [5], [12] and on hardware to process directory data [20]. Although the directory plays an important role in reducing name-mapping complexity, this work, like the early work in associative memories, devoted most of its effort to the design of the hardware component rather than to the construction of an overall computer to meet the requirements of database systems. Today's database computer designers have a much larger body of database system knowledge at their disposal than did their predecessors. As a result, it is possible for today's designers to grasp the problems of database systems and to then fabricate systems specifically for solving database system problems.

The rotating associative relational storage (RARES) design [25] is aimed at providing a high performance content-addressable memory for the realization of a relational database [9]. The RARES hardware operates in conjunction with a query optimizer such as SQUIRAL [37] to support a relational query language. Physically, RARES is connected to a CPU and buffer memory by a high speed channel. RARES uses a head-per-track rotating disk in which relational tuples (i.e., records) are stored orthogonally across tracks in "bands." A search module is associated with each band to perform access operations on the tuples in the band. The band organization greatly reduces the complexity of sending relational tuples to the CPU for

processing. This is just one example of how RARES was carefully laid out to facilitate the operation of other components. Another example of this is its ability to maintain relational tuples in sort order or to rapidly sort tuples on a domain (i.e., on a record attribute) to facilitate certain kinds of search operations. Due to its head-per-track architecture, RARES is practical only for databases smaller than 10^7 bytes in size.

The rotating associative processor (RAP) [30] was also designed for a relational database. This system is very similar to CASSM and is designed to run as a stand-alone system. The machine has a high-level "assembly language" that is used to write RAP programs which execute relational queries. Unlike RARES, RAP was not designed with an optimizer in mind. Like RARES and CASSM, the database handled by RAP is at most 10^8 bytes in size.

The database computer (DBC) is composed of functionally specialized components [3], [4] to realize an attribute-based database [21], [22]. Unlike earlier designs this one was explicitly aimed at supporting large-scale databases and at providing hardware support for security enforcement. The DBC contains specialized components for the storage of directory information, for the processing of directory information, for the storage of the database, and for security enforcement. The first three of these components were different forms of a block-oriented, content-addressable memory. This machine used hardware content addressability to facilitate not only high speed retrieval of data but also to allow fast updating of the database. The DBC actually stored the database in modified moving-head disks whose cylinders were individually content addressable. This may account for its ability to handle very large databases. The design also included an integrated component to enforce security specifications given in the query language of the system.

A trend may be apparent in the way successive database computers were designed. As database system knowledge increases, systems become less device oriented and more functionally specialized towards supporting a specific data model. Furthermore, it seemed that a database computer would require more than just one functionally specialized component. An implication of this trend is that database computer designers must be experts in both data models and hardware system architecture. In the next section, a projection of the future of database computer research is attempted.

III. The Future of Database Computers

A description of the likely advances in memory and device technology over the next decade and a discussion of the implications of this technology on database computer architecture is given here.

Device technology advances will occur primarily in three areas: processors, semiconductor random-access memory (RAM), and all-electronic bulk memories. The cost-to-performance ratio of CPU's will decline rapidly over the next ten years. Low cost CPU's with the performance capabilities of today's medium-priced minicomputers will probably be available for a few hundred dollars in five years and perhaps much less by the mid 1980's [35]. The cost of semiconductor RAM systems will also drop drastically. When 64K-bit RAM chips get into full production (perhaps, by the early 1980's) the price of main memory should decline to about 0.02–0.04 cents per bit [24]. These low cost CPU's and memory systems will make it quite feasible to dedicate processors to specific database functions. The low cost of these systems will completely obviate the need to keep their utilization high and thus they can be dedicated to tasks which are intermittent in nature.

All-electronic systems will replace fixed-head disks in the 1980's. These systems will probably use magnetic bubble memories [7], [10], electron beam memories [23], or semiconductor memories [24]. These all-electronic replacements for fixed-head disks will offer a one or two order of magnitude improvement in access time over rotating devices and could also be less costly. Electron beam memories are only feasible in large sizes and they could provide capacities of 10^8–10^9 bytes. Magnetic bubble memories are feasible in large or small sizes. They can therefore be used to replace fixed-head disks (with capacities of 10^8–10^9 bytes) and to implement logic-per-track devices in conjunction with microprocessors. The cost of the latter kind of memory would be somewhat higher than the cost of a direct fixed-head disk replacement due to the need for a large number of microprocessors and a more complex bussing structure. Semiconductor systems could be used to build stores of 10^7–10^8 bytes in the 1980's. Such memories could thus provide a fixed-head replacement with moderate capacities and very high performance. Moving-head disks will continue to be the mainstay of database bulk storage. Density improvements should allow at least 10^9 bytes per drive in the 1980's [16], [17]. Thus, systems with 10^{10}–10^{11} bytes of disk storage would be possible. Very large on-line archival systems will also be available; such systems will have very slow access times (around 10 s) but will have capacities perhaps exceeding 10^{12} bytes.

The design of a database computer is strongly influenced by the available technology. To take advantage of the device technology developments discussed above a number of guidelines for database computer architects are now given. Future database computer designs will be influenced by these observations: first, the high speed on-line bulk storage of the system will be moving-head disk storage and consequently disk accesses must be minimized for high system performance. Second, the low cost of processing power makes it quite feasible to use many independent functionally specialized components in the system to improve throughput.

The implications of moving-head disks on database computer architectures are wide-ranging and significant. To reduce disk accesses two things must be accomplished:

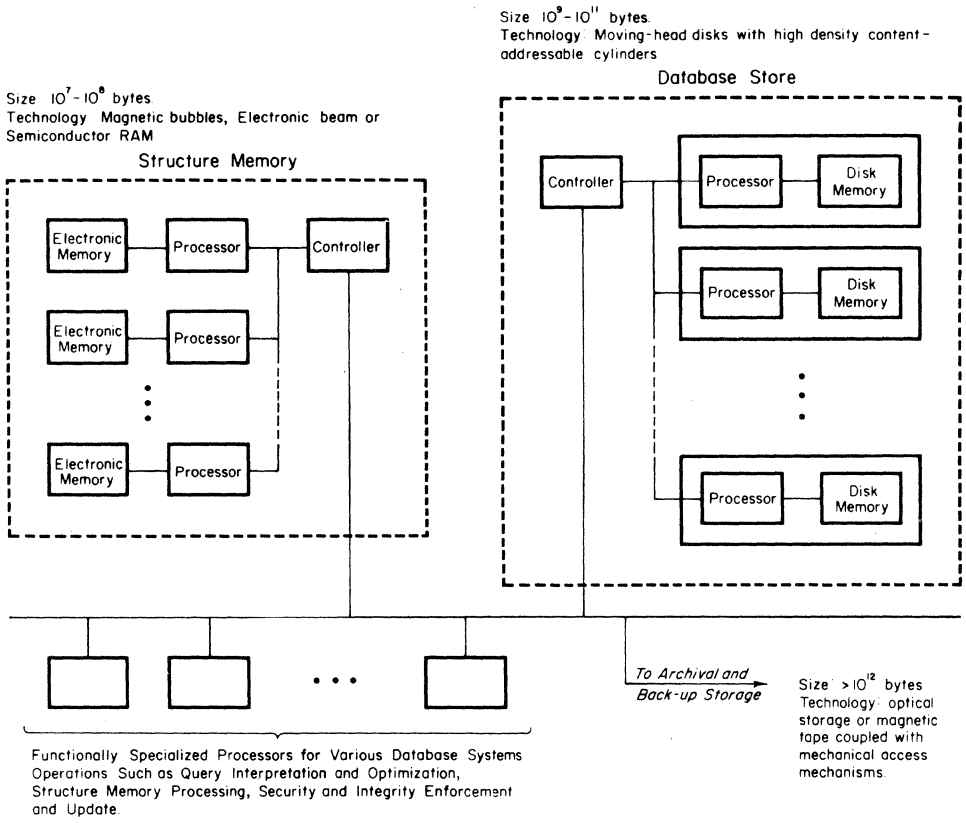


Fig. 1. Future database machine.

first, the amount of mapping information, such as pointers, on the disk storage should be made as small as possible. In this way, disk accesses will, for the most part, be executed to retrieve useful data rather than to retrieve intermediate information that is never needed by the user. Second, the need for clustering the contents of the database is great. A good clustering technique will place data that are likely to be simultaneously accessed, physically close together on the disk (say, in the same cylinders).

The need for clustering and the need to segregate the database from its mapping information may suggest several architectural principles for database computer architectures. Since mapping information would be accessed frequently to process queries, this information should be kept in a fast, functionally specialized "structure memory." This memory would be implemented with one of the all-electronic fixed-head disk replacements mentioned earlier. To facilitate clustering the amount of information obtained by each disk access should be as large as possible. This suggests that all disk I/O be carried out in a parallel-by-track mode for each cylinder accessed. Furthermore, use of several evenly spaced read/write heads per surface on the access arm could reduce arm movement latency [29] and, perhaps, allow data from several cylinders to participate simultaneously in an I/O operation. Because of the high data transfer rates of the structure memory and of disks with parallel-by-track I/O, it would be very desirable to provide localized, functionally specialized data search and manipulation logic for each memory device. This strategy would allow parallel operation of all memory search operations and avoid the need for a shared high-speed centralized processor and very high-speed data busses. The availability of low cost processors and RAM memory systems will make this design strategy quite reasonable.

Other functionally specialized components will be needed in a database computer. For example, processing elements could be designed to handle operations such as the relational "join operator" [9], [34], to sort information for storage or for processing, to process mapping information retrieved from the structure memory, and to enforce security and integrity specifications.

A promising strategy to minimize the need for mapping information and to gain other benefits as well is to provide a degree of hardware content addressability in the memory components. Such hardware can significantly aid update operations, as was shown in [3], and it also allows a closer correspondence between the logical and physical database structures [30].

The availability of large semiconductor memory systems can be of great benefit to a database computer designer. Such memories would allow a large amount of information about a user transaction to be kept in a readily available place. This could be an aid to rollback and recovery

schemes [2] and to query processing operations that require a large intermediate work space. Specialized processors to handle various aspects of update (e.g., locking, deadlock detection, and recovery) are also reasonable.

We conclude with a general description of the likely structure of a future database computer. The database computer (Fig. 1) would contain a hierarchy of memory where each level contains the mapping information required to efficiently access the next higher level as well as actual database information. Each level would consist of one or more functionally specialized memory units with localized search and manipulation capabilities. In addition, other specialized processors would be provided to handle query processing, update, security, and the user interface. To design such a computer three major problems must be overcome by future database system research.

1) How can mapping information be effectively segregated from the database and localized in a much faster structure memory? If all of the mapping information cannot be placed in the structure memory then how can they be clustered on the disk and be staged into the structure memory?

2) How can a database be clustered to minimize disk accesses? This will require a study of how users typically interact with a database by way of a data model.

3) How can a database system be decomposed into largely independent components capable of parallel operation and what is the nature of each of these components?

When these three issues are resolved, the path towards building a high performance and economically feasible database computer will be clear and the concept of data utility will become a reality.

REFERENCES

[1] M. M. Astrahan and D. C. Chamberlin, "Implementation of a structured english query language," *Commun. Ass. Comput. Mach.*, vol. 18, pp. 580–588, Oct. 1975.

[2] M. M. Astrahan *et al.*, "System R: A relational approach to database management," *ACM Trans. Database Systems*, vol. 1, pp. 1–25, Sept. 1976.

[3] R. I. Baum, "The architectural design of a secure database management system," Comput. and Inf. Sci. Res. Center, Ohio State Univ., Columbus, Tech. Rep. (OSU-CISRC-TR-75-8).

[4] R. I. Baum and D. K. Hsiao, "A data secure computer architecture," in *Dig. Papers COMPCON 76*, Feb. 1976.

[5] P. B. Berra and A. K. Singhania, "A multiple associative memory organization for pipelining a directory of a very large data base," in *Dig. Papers COMPCON 76*, Feb. 1976, pp. 109–112.

[6] P. B. Berra, "Some problems in associative processor applications to database management," in *Proc. 1974 AFIPS Nat. Comput. Conf.*, vol. 43, 1974, pp. 1–5.

[7] A. H. Bobeck, P. I. Bonyhard, and J. E. Geusic, "Magnetic bubbles – An emerging new memory technology," *Proc. IEEE*, vol. 63, pp. 1176–1195, Aug. 1975.

[8] D. Chamberlin, J. N. Gray, and I. L. Traiger, "Views, authorization and locking in a relational database system," in *Proc. 1975 AFIPS Nat. Comput. Conf.*, May 1975.

[9] E. F. Codd, "A relational model of data for large shared data banks," *Commun. Ass. Comput. Mach.*, vol. 13, pp. 377–387, June 1970.

[10] M. S. Cohen and H. Chang, "The frontier of magnetic bubble technology," *Proc. IEEE*, vol. 63, pp. 1196–1206, Aug. 1975.

[11] G. P. Copeland, G. J. Lipovski, and S. Y. W. Su, "The architecture of CASSM: A cellular system for non-numeric processing," in *Proc. First Annu. Symp. Comput. Architecture*, Dec. 1973, pp. 121–128.

[12] G. F. Coulouris, J. M. Evans, and R. W. Mitchell, "Towards content addressing in databases," *Comput. J.*, vol. 15, pp. 95–98, Feb. 1972.

[13] C. R. DeFiore and P. B. Berra, "A data management system utilizing an associative memory," in *Proc. 1973 AFIPS Nat. Comput. Conf.*, vol. 42, 1973, pp. 181–185.

[14] J. A. Dugan *et al.*, "A study of the utility of associative memory processors," in *Proc. 21st ACM Nat. Conf.*, 1966, pp. 347–360.

[15] J. N. Gray, R. A. Lorie, and G. R. Putzolu, "Granularity of locks in a shared database," in *Proc. Int. Conf. Very Large Databases*, Framingham, MA, Sept. 1975.

[16] J. M. Harker and C. Hsu, "Magnetic disks for bulk storage – past and future," in *AFIPS Conf. Proc.*, vol. 40, 1972, pp. 945–955.

[17] K. E. Haughton, "An overview of disk storage systems," *Proc. IEEE*, vol. 63, pp. 1148–1152, Aug. 1975.

[18] L. D. Healy, K. L. Doty, and G. J. Lipovski, "The architecture of a context addressed segment sequential storage," in *1972 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 41. Montvale, NJ: AFIPS Press, 1972, pp. 691–701.

[19] G. D. Held, M. Stonebraker, and E. Wong, "INGRES– A relational database management system," in *Proc. 1975 AFIPS Nat. Comput. Conf.*, May 1975.

[20] L. A. Hollaar, "A list merging processor for information retrieval systems," presented at the Workshop on Architecture for Non-Numerical Processing, Dallas, TX, Oct. 1974.

[21] D. K. Hsiao, *Systems Programming – Concepts of Operating and Database Systems*. Reading, MA: Addison-Wesley, 1975, ch. 6.

[22] D. K. Hsiao and F. Harary, "A formal system for information retrieval from files," *Commun. Ass. Comput. Mach.*, vol. 13, Feb. 1970; Corrigenda, *Commun. Ass. Comput. Mach.*, vol. 13, Mar. 1970.

[23] W. C. Hughes *et al.*, "A semiconductor nonvolatile electron-beam accessed mass memory," *Proc. IEEE*, vol. 63, pp. 1230–1240, Aug. 1975.

[24] D. A. Hodges, "A review and projection of semiconductor components for digital storage," *Proc. IEEE*, vol. 63, pp. 1136–1147, Aug. 1975.

[25] S. C. Lin, D. C. P. Smith, and J. M. Smith, "The design of a rotating associative memory for relational database applications," *ACM Trans. Database Systems*, vol. 1, pp. 53–75, Mar. 1976.

[26] R. R. Linde, R. Gates, and T. Peng, "Associative processor applications to real time data management," in *Proc. 1973 AFIPS Nat. Comput. Conf.*, vol. 42, 1973, pp. 187–195.

[27] N. Minsky, "Rotating storage devices as partially associative memories," in *1972 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 41. Montvale, NJ: AFIPS Press, 1972, pp. 587–596.

[28] R. Moulder, "An implementation of a data management system on an associative processor," in *Proc. 1973 AFIPS Nat. Comput. Conf.*, vol. 42, 1973, pp. 171–176.

[29] R. B. Mulvany, "Engineering design of a disk storage facility with data modules," *IBM J. Res. Development*, vol. 18, Nov. 1974.

[30] E. A. Ozkarahan, S. A. Schuster, and K. C. Smith, "RAP – Associative processor for database management," in *AFIPS Conf. Proc.*, vol. 44, 1975, pp. 379–388.

[31] B. Parhami, "A highly parallel computer system for information retrieval," in *1972 Fall Joint Computer Conf., AFIPS Conf. Proc.*, vol. 41. Montvale, NJ: AFIPS Press, 1972, pp. 681–690.

[32] ———, "Associative memories and processors: An overview and selected bibliography," *Proc. IEEE*, vol. 61, pp. 722–730, June 1973.

[33] J. L. Parker, "A logic per track retrieval system," in *Proc. IFIP Congr. 1971*. Amsterdam, The Netherlands: North-Holland, 1971, Ta4-146–TA4-150.

[34] J. B. Rothnie, "Evaluating inter-entry retrieval expressions in a relational database management system," in *Proc. 1975 AFIPS Nat. Comput. Conf.*, May 1975.

[35] H. Schmid, "Monolithic processors," *Comput. Design*, pp. 88–95, Oct. 1974.

[36] D. L. Slotnick, "Logic per track devices," in *Advances in Computers*, vol. 10, F. Alt, Ed. New York: Academic, 1970, pp. 291–296.

[37] J. M. Smith and P. Y. Chang, "Optimizing the performance of a relational algebra database interface," *Commun. Ass. Comput. Mach.*, vol. 18, pp. 568–579, Oct. 1975.
[38] S. Y. W. Su and G. J. Lipovski, "CASSM: A cellular system for very large databases," in *Proc. Int. Conf. Very Large Databases*, Sept. 1975, pp. 456–472.
[39] M. Stonebraker, "Implementation of integrity constraints and views by query modification," in *Proc. 1975 SIGMOD Workshop on Management of Data*, San Jose, CA, May 1975.

Richard I. Baum received the B.S. degree with distinction in engineering physics from Cornell University, Ithaca, NY, in 1971, and the M.S. and Ph.D. degrees in computer and information science from Ohio State University, Columbus, in 1974 and 1975, respectively.

From October 1971 to September 1972 he held a Fellowship at Ohio State University. From October 1972 to May 1976 he was a Research Associate at Ohio State University, during which time he developed new architectural models for secure database systems culminating in his Ph.D. dissertation in the same area. He is currently involved in the architectural design of computers at IBM, Poughkeepsie, NY.

Dr. Baum is a member of the Association for Computing Machinery and the IEEE Computer Society.

David K. Hsiao (M'68) received the Ph.D. degree in computer and information sciences from the University of Pennsylvania, Philadelphia, in 1968.

He has previously conducted research at the Honeywell Information Sciences Research Center, and taught at the Moore School of Electrical Engineering, University of Pennsylvania. In the Fall of 1975 he was a Visiting Professor of Management at the Sloan School of the Massachusetts Institute of Technology. He is presently a Faculty Associate at the IBM Research Laboratory, San Jose, CA, and an Associate Professor of Computer and Information Science at Ohio State University, Columbus. He has published widely in the area of database systems design and engineering and is the author of a recent textbook, *Systems Programming—Concepts of Operating and DatabaseSystems* (Reading, MA: Addison-Wesley). Currently, he serves as the Editor of the *ACM Transactions on Database Systems* (TODS) and is the Chairman of the Technical Committee on Database Engineering of the IEEE Computer Society.

Concepts and Capabilities of a Database Computer

JAYANTA BANERJEE and DAVID K. HSIAO
The Ohio State University
RICHARD I. BAUM
IBM Poughkeepsie Laboratory

The concepts and capabilities of a database computer (DBC) are given in this paper. The proposed design overcomes many of the traditional problems of database system software and is one of the first to describe a complete data-secure computer capable of handling large databases.

This paper begins by characterizing the major problems facing today's database system designers. These problems are intrinsically related to the nature of conventional hardware and can only be solved by introducing new architectural concepts. Several such concepts are brought to bear in the later sections of this paper. These architectural principles have a major impact upon the design of the system and so they are discussed in some detail. A key aspect of these principles is that they can be implemented with near-term technology. The rest of the paper is devoted to the functional characteristics and the theory of operation of the DBC. The theory of operation is based on a series of abstract models of the components and data structures employed by the DBC. These models are used to illustrate how the DBC performs access operations, manages data structures and security specifications, and enforces security requirements. Short Algol-like algorithms are used to show how these operations are carried out. This part of the paper concludes with a high-level description of the DBC organization. The actual details of the DBC hardware are quite involved and so their presentation is not the subject of this paper.

A sample database is included in the Appendix to illustrate the working of the security and clustering mechanisms of the DBC.

Key Words and Phrases: database computers, content-addressable memory, structure memory, mass memory, keywords, security, clustering, performance
CR Categories: 6.22, 4.33

1. PROBLEMS AND SOLUTIONS

A number of major problems have been faced by database designers for a long time. These problems are of a very general nature and have frequently plagued builders of both hardware and software database systems. This section of the paper contains a discussion of these problems and of the architectural principles adopted in the DBC design which solve them.

The work reported herein was performed at the Ohio State University, Columbus, OH.
Authors' addresses: J. Banerjee and D. K. Hsiao, Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210; R. I. Baum, IBM Poughkeepsie Lab, P.O. Box 390, Poughkeepsie, NY 12602.

1.1 The Problems of Database System Software

A. Name-Mapping Complexity

The complexity of database system software is due, in large part, to the requirement for *name-mapping operations*. Name-mapping operations convert symbolic data names, called a query, into memory addresses which identify where the data named by the query can be found. Since the language which is used to name data is usually far more powerful than the addressing scheme implemented by the hardware, it is normally necessary to have rather involved name-mapping algorithms. Name-mapping algorithms must be highly optimized if they are to perform well. In particular, these algorithms must minimize their secondary storage access requirements. To accomplish this, most name-mapping algorithms use very complex auxiliary data structures to guide their operation.

To illustrate these problems consider the difficulties of the following typical name-mapping scenario. First, the query is used to access a "directory." The directory contains information which allows the algorithm to determine the approximate location of the requested data (this information thus potentially reduces the number of secondary storage accesses required by the algorithm). The information retrieved from the directory is then processed in some manner to yield secondary storage addresses. Finally, the secondary storage is accessed and the data are located. This software name-mapping algorithm requires auxiliary data structure in both the directory and the secondary storage. These auxiliary data structures, which include elements such as pointers, allow rapid retrieval of data from the secondary storage and the directory. All of these auxiliary data structures must be properly maintained. This last requirement is the underlying cause for the great difficulty that most contemporary systems have in executing update operations. Update operations make changes to auxiliary structures and so they are frequently very time-consuming. A classic example is the process of modifying a network of address pointers.

B. Performance Bottlenecks

Database system software normally consists of several distinct functional parts which perform specific tasks. For example, separate software modules which perform query parsing, directory access, directory processing, data retrieval and update, and data security are usually found in contemporary database management systems. To have a well-balanced system with high throughput it is necessary for these modules to have diverse performance capabilities. Such diverse capabilities are difficult to achieve when all these software modules are implemented on the same underlying hardware. When such performance capabilities cannot be met because of inherent hardware constraints, the system develops bottlenecks and its performance is consequently degraded. Contemporary database management systems are usually plagued by many such bottlenecks.

C. Data Security Overhead

Powerful data security facilities are generally a performance hindrance on contemporary systems. The most powerful data security mechanisms allow security specifications to be written in the query language of the system. To

authenticate access operations it is therefore necessary to perform multiple name-mapping operations—one for determining the requested data and several for determining the data being affected by the security specifications. The use of name-mapping algorithms to carry out security enforcement is generally too much of a performance burden to be seriously considered in present systems.

D. Add-On Approach to Security

Security capabilities are frequently just an "add-on" to present systems. This kind of design philosphy opens the way to not only performance difficulties but also to questionable reliability. With the high degree of complexity of current systems it is extremely difficult to add on a security mechanism which will guarantee that all "backdoors" are, in fact, closed.

1.2 The Problems of Building Database Machine Hardware

A. The Need for Distant Technology

Attempts to build database machine hardware have been made before [10, 11, 25, 26, 28]. These efforts have been plagued by a number of critical drawbacks. The most serious shortcoming in these systems has been their reliance on monolithic fully associative memories. Such memories are not feasible for supporting a large online database (i.e., at least 10^9 bytes) unless we are ready to wait for a major advancement in technology.

B. Incomplete Hardware Designs

Many hardware design attempts [6, 10, 13, 21, 24, 27] have led to machines that could not perform all of the functions necessary to support a viable database management system. In particular, some of them can support just one database management function in hardware such as directory processing or data retrieval; others cannot adequately support such critical functions as the update function. Previous hardware design approaches have almost always lacked a data security capability—such an omission makes the use of computers in a data sharing environment very questionable indeed. A viable database machine must support all database management functions equally well.

1.3 Problem Solving Concepts

To overcome the problems described above a number of key design concepts were used in the DBC. These design concepts include both architectural principles and design philosophy.

A. Partitioned Content-Addressable Memories

The use of hardware content addressing can significantly reduce the need for name-mapping data structures. Content-addressable memories eliminate the need for knowing the actual location of a data item. In such a memory the notion of "actual location" is nonexistent; instead, data are accessed by specifying their content. This kind of access gives us a very important capability: data items may be moved about without any need to modify name-mapping data structures. This is because few, if any, name-mapping data structures are needed in a content-addressable memory. This characteristic greatly facilitates update operations.

A fully associative memory large enough to hold a complete database is not yet

feasible. However, a storage system consisting of many blocks (called *partitions*) of memory each of which is content-addressable is quite feasible. We call this memory concept a *partitioned content-addressable memory* (PCAM). It is possible to build PCAM's of widely varying performance characteristics. In particular, it is possible to design the access speed and capacity of a PCAM to meet a particular performance requirement. This flexibility allows us to design these PCAM's for use in the proposed DBC architecture with very different speeds and capacities. As will be seen later, a PCAM of gigabyte capacity is feasible with current technology.

B. Structure and Mass Memories

Since PCAM's are block-oriented, it is necessary to have some name-mapping data structures in the system. Our goal, of course, is to minimize their use as much as possible. This leads to the architectural concept of the *structure memory*. A DBC employing this concept has two memories. The *mass memory* contains the information making up the database and is by far the larger of the two memories. The mass memory contains only *update invariant* name-mapping data structures. Once an update invariant data structure is created for a data item it need never be modified so long as that data item continues to exist *anywhere* in the database. The data structures in conventional mass storage are not update invariant; they must be modified whenever the location of a data item changes. The structure memory contains all of the nonupdate invariant name-mapping information necessary to locate data in the mass memory. To access the database the system first accesses the structure memory, obtains mapping information, processes it and then accesses the mass memory.

The proposed DBC employs the structure memory concept. Both the mass memory and the structure memory are PCAM's. They, of course, have very different functional characteristics.

C. Area Pointers

To simplify the name-mapping data structures that are still required by the DBC, a concept called the *area pointer* is used. An area pointer indicates the PCAM partition in which a data item may be found by employing content-addressing. Unlike the location pointers used in contemporary systems, area pointers need not be modified when data items are moved around within a partition.

Conventional mass memories do not support the area pointer concept. Our mass memory, on the other hand, is a PCAM and so area pointer support comes naturally. Area pointers are stored in and managed by the structure memory.

D. Functional Specialization

The DBC contains a number of components with considerably different processing speed and memory capacity requirements. The mass storage and structure memory are examples of two such components. To keep any component from becoming a bottleneck we employ the architectural concept of *functional specialization.*[1] In a functionally specialized system, the components are individually

[1] This term was suggested to us by E. Feustel.

designed to be optimally adapted to their function. The processing power and memory capacity of each component is determined by its role in the system. Because all major components are specialized (i.e., functionally separate from other components), estimation of their required processing power and memory capacity is much easier. In the proposed DBC each of the major components is a physically separate hardware component. This approach allows us to build a relatively well-balanced system and to avoid bottlenecks by providing each component with the right amount of processing power and memory capacity.

The proposed DBC has seven major functionally specialized components; the keyword transformation unit (KXU), the structure memory (SM), the mass memory (MM), the structure memory information processor (SMIP), the index translation unit (IXU), the database command and control processor (DBCCP), and the security filter processor (SFP). These seven components are the heart of a database computer that is able to support gigabyte database capacities while providing full retrieval, update, and security capabilities.

E. Look-Aside Buffering

When an update operation occurs it is sometimes necessary to modify name-mapping data structures. To insure the correct execution of the queries which follow the update, the execution of queries is normally postponed until the update operation and all of its related changes to data structures are complete. This is because the data involved in an update operation could very well be the data the next operation depends on. Thus update operations can become bottlenecks in contemporary systems.

In the DBC, changes to name-mapping data structures induced by an update operation will be very few in number because the partitions in the mass memory are large, making the number of different indices small. Nevertheless, even these changes in the structure memory require some time. To reduce the wait-time of other more frequent commands, a fast *look-aside buffer* is used. The changes (consequent of the update commands) are recorded in this buffer and are referred to by all subsequent commands until the necessary changes are permanently recorded in the structure memory. The permanent updates to the structure memory are postponed until the system reaches a relatively slack period. In this way, queries following an update operation do not have to wait for the permanent effects of that update operation to be actually recorded before they are executed.

F. An Integral Data Security Mechanism

At the outset the security mechanism was made an integral part of the DBC design. This design philosophy not only allows us to construct a system that has no "backdoors" but also insures that all access requests are, in fact, controlled by the DBC's security mechanism. We achieved this by designing the security mechanism first and then designing the rest of the system around it. The DBC supports a security specification language that is the same as the DBC's query language.

Security in the DBC is provided in terms of two distinct protection mechanisms. The first mechanism, based on the security atom concept [23], requires some form of cooperation from the creator of the database. This mechanism achieves enforcement in a rapid and elegant manner and is incorporated in the database

command and control processor (DBCCP). The second enforcement mechanism allows the database creator wide latitude in the manner in which he can specify security-related information. Since it generally requires more (and different) processing than the first, the second mechanism is incorporated in a functionally specialized component, the security filter processor (SFP). Such an architecture tends to lead to good performance while ensuring that security is not compromised.

G. Performance Enhancement by Clustering Techniques

A powerful clustering technique has been incorporated in the DBC, which allows the creator of the database to optimize access times. The placement of every record into the DBC can be controlled (in terms of its properties) by the creator of the database in such a way that retrieval of records with similar properties may be accomplished with minimal access delays.

H. Emerging and Existing Technology

A database computer for the near future should take maximum advantage of the emerging technology. This design philosophy is especially important in an era of rapidly developing technology such as the present one. The significant developments expected in the area of high-speed secondary storage (semiconductors: CCD's and dense RAM's, magnetic bubbles and electron beam memories) and low-cost processing power (microprocessors) dictate a major rethinking of conventional machine architectures.

For example, an all-electronic storage component may replace the fixed-head disk as the fastest secondary storage device in the system. Since these all-electronic fixed-head disk replacements will offer at least an order of magnitude improvement in access time, they will allow powerful data organizations as well as a significant increase in the throughput of certain database system components. Low-cost random access memory will allow the widespread use of very large data buffers and independent, functionally specialized memories throughout the system. Low-cost microprocessors coupled with low-cost moving-head disks with tracks-in-parallel read-out modification will allow high-volume processing with content-addressable capabilities.

1.4 Approaches to Database Machine Design

With regard to the basic database functions of searching, retrieving, and updating of data, it is possible to identify three different approaches toward a hardware solution. The first is the *logic-per-track* approach where the entire database is stored in the tracks of a fixed-head disk device and enough processing logic is incorporated into the device so that all the tracks can be simultaneously content-addressed and processed. The second approach is the *high-speed processor* approach. The database is still stored in a relatively inexpensive secondary memory (such as moving-head disks) and only a portion of the database is staged into a high-speed parallel processing device. Processors with parallel processing capability are used to carry out a variety of database management operations on the information resident in the device. Finally, there is the *back-end 'machine'* approach where the operating system is relieved from the data management responsibilities and the database management is carried out by a dedicated

general-purpose computer with appropriate software which also manages the I/O devices for the database.

The context-addressed segment sequential memory (CASSM) [28] and the rotating associative relational store (RARES) [21] are examples of database machines designed with the logic-per-track approach. CASSM is based on a cellular architecture. It consists of an array of cells with the provision for communication between adjacent cells. Each cell is made up of a segment (which is a track of a head-per-track device), a pair of physically separated read-write heads, and an arithmetic unit. Data structures, such as trees, graphs, and tables may be mapped to the database store. The instruction set is powerful enough to allow for context-search of tree-like data structures.

In RARES, proposals were made to manage relational databases. A relational tuple (i.e., record) is stored across a band of tracks instead of on a single track. Logic is associated with each band of tracks rather than with each track.

The disadvantage of the logic-per-track approach is that it is not cost-effective to implement large databases on logic-per-track devices. Even though RARES only requires a head-per-track and logic-per-band device, it is still quite expensive to support large databases on such devices. Furthermore, there is a definite synchronization problem when a single unit of data is read (written) by more than one read (write) mechanism.

An example of the high-speed processor approach is the relational associative processor (RAP) [26]. RAP, a machine based on the relational model of data, makes use of a relatively small number of parallel processing cells for storing the active relations (files) or the active portions of one or more relations (subfiles). The main database is still stored in conventional online secondary memory. The primary disadvantage of this approach is that certain relational operations such as the equality join, as well as sequences of operations referring to a large number of relations, will require frequent staging of data to the processor. Suggestions were made to use a CCD-implemented RAP as a logic-per-track device for database store, since RAP is similar to CASSM in architectural design. However, CCD-implemented RAP as a main database store is only viable for very small databases. Another example of high-speed processor approach is the use of an array processor such as STARAN as a staging device for database management [11].

The third database machine approach is that of a back-end 'machine'. The idea of a back-end machine for performing specialized data management tasks was originated in [8]. In this approach, the operating system and user applications are resident in a front-end, general-purpose computer, while the database and the data management software reside in the back-end, general-purpose computer and its storage devices. The importance of this approach is that data management tasks are logically, as well as physically, separated from other activities such as compilation and assembly. This provides for a greater reliability in database management and a more flexible sharing of the centralized database.

The Datacomputer [22] is another example of the back-end machine approach. The Datacomputer is a large-scale data management software running on a medium-size computer such as PDP-10. The Datacomputer provides facilities for data sharing of a centralized database among dissimilar front-end computers in a network environment.

The disadvantage of both the Datacomputer and the back-end 'machine' of [8] is that they are conventional computers performing data management tasks by software means. Therefore, the difficulties of name-mapping and data updating still remain an integral part of these machines. Consequently, the performance of these software-laden systems remains low.

The database computer (DBC) is an example of the third design approach with the difference that the back-end machine is almost entirely devoid of software and has specialized hardware for data management. It can support a very large online database, say of 10^{10} bytes in size. The economy of such a large content-addressable storage is due to a PCAM (partitioned content-addressable memory) organization of the database store. The DBC, therefore, does not require staging of data between levels of memories of various speeds. Further advantages of the DBC accrue from a high-level query language for interface with the front-end computer, and from a content-based security mechanism for access control.

2. THE FUNCTIONAL CHARACTERISTICS OF THE DBC

The DBC must communicate with external systems and so a DBC interface must be defined. The functional characteristics of the DBC provide such an interface. The DBC functional characteristics define the data management and security features supported by the DBC and show how commands are sent to and executed by it.

2.1 A Back-End Machine

The DBC is not a general-purpose computer and does not have a typical operating system. Instead, it is a separate machine dedicated to database operations. Other computers and systems communicate with the DBC by using DBC access commands and by sending or receiving database information. The decision to design the DBC as a back-end machine to support database operations in a general-purpose computer system is a consequence of applying the concept of functional specialization. A number of advantages accrue from this decision [8]. First, the DBC is not constrained to be used with any particular kind of general-purpose computer system. Second, more than one system can share a DBC. In this way, the back-end DBC can serve many front-end computer systems. Third, several DBC's can become a part of a general-purpose computer system to facilitate distributed database applications [3]. This interconnection could be done with a geographically distributed communications network. Finally, all DBC access channels can be identified and controlled. This is necessary to insure that no "backdoors" into or out of the DBC exist (see Figure 1).

We shall collectively call all of the systems which communicate with the DBC the *program execution system* (PES). We aggregate all these systems into one conceptual entity so that it will be easier to describe the operation of the DBC.

2.2 The Functional Model

The DBC proposed here implements the attribute-based model. This model has been extensively studied and is particularly well-suited to supporting contemporary database functions [14, 15, 23].

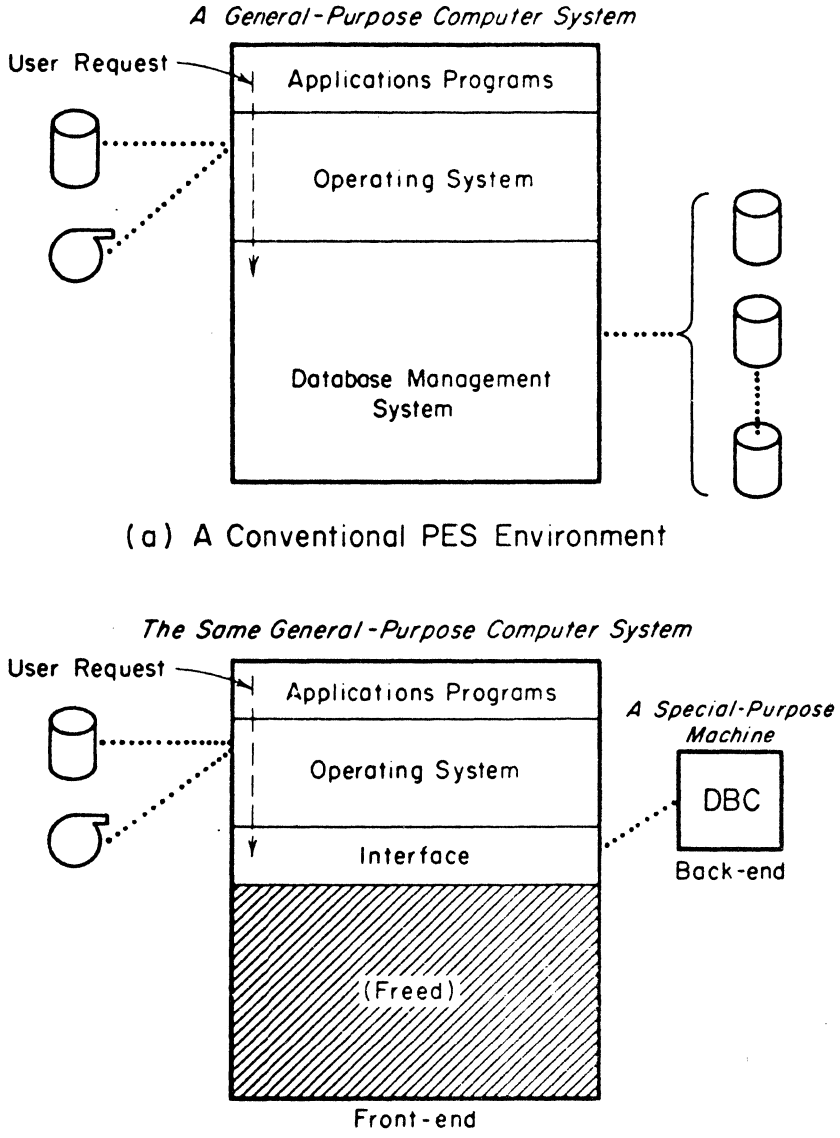


Fig. 1. The relationship of a database computer with its front-end computer

A. Queries—The Symbolic Data Names Used by the DBC

Our definition of a database starts with two terms: a set AT of "attributes" and a set VA of "values". These are left undefined to allow the broadest possible interpretation. We shall denote a member of AT by at and a member of VA by v.

A *record* R is a subset of the cartesian product $AT \times VA$. To simplify the notation we will assume without loss of generality that all attributes in a record are distinct. Thus R is a set of ordered pairs of the form

(an attribute, a value).

Records are physically stored in the mass memory. The set of all records in the mass memory is called a *database* (DB). The database may be partitioned into subsets called files. To distinguish among several files, each file is given a unique name F, called its *file name*.

The *keywords* of a record are those attribute-value pairs which characterize the record. In practice it is useful to consider only succinct keywords. We shall denote a keyword by the notation K.

A *keyword predicate* $P(K)$ is true for a keyword K if K satisfies the condition specified by P. The most commonly used keyword predicate is the *equality predicate* $E(K)$ which is true for K when K is the same as a certain keyword, say, K'. For this special case, we shall denote the keyword predicate by simply K'. Another common keyword predicate is the *less-than predicate* $LT_{at}(K)$. This predicate is true for K when the attribute of K is at and the value of at is less than some value, say, v. This keyword predicate shall be denoted by $(at < v)$. This predicate can be easily generalized to handle other relational operators such as '>', '≠', '≥', and '≤'.

A keyword predicate is *true* for a record R if some keyword K in R satisfies the keyword predicate. A *query* is a proposition given by a Boolean expression of keyword predicates. A query is *true* for R if this proposition holds for the keywords in R; such a record is said to *satisfy* the query. The set of all records in the database (or in a file of DB) that satisfy a query Q will be called its *response set* and will be denoted by $Q(\text{DB})$ (or $Q(F)$). Every query is written in disjunctive normal form

$$Q^1 \vee Q^2 \vee \ldots \vee Q^k$$

where each Q^i of the query has the form

$$P_1{}^i \wedge P_2{}^i \wedge \ldots \wedge P_n{}^i$$

and each $P_j{}^i$ is a keyword predicate. Some examples of queries follow. The query $K_1 \wedge K_2$ is true for R when K_1 and K_2 are both in R. The query $K_1 \wedge (Salary < 10{,}000)$ is true for R when K_1 is in R and there is a keyword in R whose attribute is *Salary* and whose value is less than 10,000. More elaborate queries can be formed if they are in disjunctive normal form.

B. Security Specifications—The Protection of Data

A *database access* or simply an *access* is the DBC operation which transfers information to or extracts information from the database. Examples of accesses are retrieve, insert, and delete. Let ACC denote the set of names of all the accesses available in the DBC. Let a member of ACC be represented by a and a subset of ACC by A.

A *security specification* is a relation

$$S\text{: DB} \rightarrow 2^{ACC} \text{ where } 2^{ACC} \text{ is the power set of } ACC.$$

Thus for a record R in DB, the security specification $S(R) = A$ indicates which subset A of accesses is permitted on R.

A *file sanction* or simply a *sanction* is defined as the couple (Q, A) where Q

is a query, and A is a subset of ACC. A sanction (Q, A) induces a security specification $S.FS_{Q,A}$ over records R of the database such that

$$S.FS_{Q,A}(R) = \begin{cases} A, \text{ if } R \text{ satisfies } Q. \\ ACC, \text{ otherwise.} \end{cases}$$

Thus a sanction indicates that only the accesses in A may be performed on the records satisfying Q. When R does not satisfy Q, all accesses may be performed on it. In this case, we say that no sanctions of (Q, A) are applicable to R. The sanction is a very powerful type of security specification, since it allows the full power of the query language (i.e., Q) to be used to specify records to be protected.

Consider a file named F and a set of sanctions T where

$$T = \{(Q_1, A_1), (Q_2, A_2), \ldots, (Q_m, A_m)\}.$$

A *database capability* (F, T) induces a security specification $S.DC_{F,T}$ over the records of F such that

$$S.DC_{F,T}(R) = \bigcap_{i=1}^{m} S.FS_{Q_i,A_i}(R)$$

where R is a record of F. In words, $S.DC_{F,T}(R)$ is the set of all accesses granted for R by one or more file sanctions in T and not denied by any sanction T. Security specifications are therefore stored in the DBC as database capabilities. The database capabilities specify exactly when access operations are allowed on records. The DBC maintains a database capability for each active user of every file.

For example, consider the database capability having the file sanctions, $T = \{(Q_1, A_1), (Q_2, A_2)\}$. Suppose Q_1 and Q_2 specify overlapping sets of records as shown in Figure 2. Then the records in the intersection of Q_1 and Q_2 have the access privileges $A_1 \cap A_2$ associated with them.

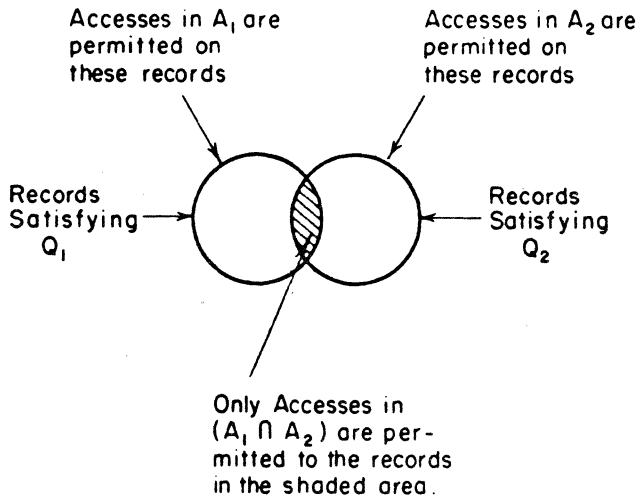


Fig. 2. The security specification induced by $\{(Q_1, A_1), (Q_2, A_2)\}$

C. Command Execution—The Processing of Access Requests

An access command has the form $\langle U, (F, Q), a \rangle$ or the form $\langle U, (F, R), a \rangle$. U represents the name of the user issuing the command, a is an access, (F, Q) represents the response set $Q(F)$ on which the access is to be performed, and (F, R) represents a record R of F that is to be used in the access. Before an access is executed, file F must be protected from unauthorized access by the user U. This is accomplished by first employing U to locate the appropriate database capability (F, T). Then for the command $\langle U, (F, Q), a \rangle$, the access a is performed on each record R of $Q(F)$ for which $S.DC_{F,T}(R)$ contains a. For the command $\langle U, (F, R), a \rangle$, the access a is performed on R if a is in $S.DC_{F,T}(R)$. If any data need be sent to the user as a result of the access command, it is sent to the front-end program execution system (PES) to be routed to that user.

2.3 The Need for Front-End Support

Before a user issues any access commands for a file, the database capability specifying the user's access rights to that file is sent to the DBC by the PES. An access command is rejected by the DBC unless the appropriate database capability is found. It is the responsibility of the PES to send the correct database capabilities to the DBC and to authorize the use of access operations to users by constructing appropriate database capabilities. In this way our DBC design does not impose any restriction on the nature of the PES's security mechanisms or on the authorization policies it supports.

3. THEORY OF OPERATION

A model which describes the basic components of the DBC and how these components interact to realize the DBC's functional characteristics is now given. In the presentation we do not emphasize the intricacies of hardware design. Instead, we describe the operation of the components at a conceptual level. We have shown in [16, 19, 20] how these components can actually be implemented with existing and emerging technology.

The theory of operation is presented in two sections. In the first section a data model is developed. In the next section we show how the data model described above is realized by the DBC with the aid of functionally specialized components.

3.1 The Data Model

The need for auxiliary data structures arises from the fact that the mass memory is not fully associative. Therefore, a technique to minimize mass memory accesses is required to insure high performance. We shall employ a PCAM-based mass memory. The mass memory's content-addressability allows it to contain only update-invariant mapping structures. The data model will allow us to determine the nature of the information to be kept in the structure memory.

When a PCAM partition is used to store records, record placement within the partition does not affect the system's performance. When a set of records is not placed in the same partition, the system's performance can be affected since multiple PCAM accesses may be required to retrieve the records. To address this problem a database is normally partitioned into groups of records that need to be

stored physically close to each other. The exact nature of "closeness" is dependent on the properties of the memory. For example, on a disk with movable read/write heads, records could be considered close if they are stored in the same cylinder. This seems reasonable since the cost of initially accessing a cylinder of the disk is usually much greater than the cost of immediately following subsequent accesses to tracks of the same cylinder. The underlying reason for this is the requirement for mechanical motion to access a new cylinder. In our data model we shall consider records to be close when they are stored in the same partition of the PCAM mass memory. To distinguish partitions in the mass memory PCAM from those in other PCAM's, we shall call each of these partitions a *minimal access unit* (MAU).

There are many reasons for placing one record close to another record. A basic reason, related to performance, is the likelihood that these records will be accessed simultaneously. There are other reasons for grouping records. For example, compartmentalization of records for security reasons is one. Precisely what features of these records allow the designer to deduce a particular record grouping does not concern us at this time. Our goal as builders of generalized hardware to support a database system is not to choose a specific way to partition the database but instead to provide a general mechanism with which many possible groupings may be realized. Such a mechanism will be presented shortly.

A. Storage Structure

Let there be L MAU's in the mass memory and let L be called the *minimal access unit count.* All L MAU's are of fixed size. We denote the *minimal access unit size* by $|\,\text{MAU}\,|$. Associated with the database DB is the set of records denoted by $M(\text{DB})$ and defined as $\{R{:}R \text{ is in DB}\}$.

If the set $M(\text{DB})$ is further partitioned into L subsets and each of these subsets represents the records which are placed in an MAU, then the union of the subsets is called a *database configuration* of $M(\text{DB})$. The size of a record, i.e., the number of bits needed to represent it in memory, is denoted by $|\,R\,|$. A database configuration is *valid* if each subset X of $M(\text{DB})$ satisfies the constraint

$$\left(\sum_{R \in X} |\,R\,|\right) \leq |\,\text{MAU}\,|.$$

In other words, a database configuration is valid if all of the records of $M(\text{DB})$ fit into MAU's of the mass memory. A valid database configuration results in a *memory map* which describes how the records are placed in the mass memory.

Each MAU is represented by a unique name called the *minimal access unit number* (MAU number), denoted by f, where $0 \leq f < L$. Let M_f represent the contents of the MAU numbered f.

The DB *storage structure* is defined as the ordered sequence

$$(M_0, M_1, \ldots, M_{L-1}).$$

This sequence represents the distribution of records in the MAU's.

Let F be a file whose records contain just m different keywords denoted by K_1, $K_2, \ldots, K_m$. To keep track of the MAU's in which records containing the keyword K_i are to be found, we form the set $D(F, K_i)$ defined as

$$\{f \mid R \text{ is in } F \text{ and } K_i \text{ is in } R \text{ and } R \epsilon M_f\}.$$

$D(F, K_i)$ is called a *directory entry* and each element f of $D(F, K_i)$ is called an *index term.* In words, $D(F, K_i)$ is the set of MAU numbers of MAU's which contain one or more records with the keyword K_i.

The *directory* of file F is defined as the set DIR(F) defined as

$$\{D(F, K_1), D(F, K_2), \ldots, D(F, K_m)\}.$$

The directory of a file represents the structural information needed to access the mass storage. We shall see how it is used shortly.

As mentioned earlier, the DBC allows the creator of a file to enhance performance by allowing records of the file to be identified as a cluster and by accessing such records with minimal access delay. Let us motivate the concept of clustering and the resulting performance improvement by a simple example. Let a file F (to be placed in the DBC) have n records of which we choose four records for our discussion. These four are shown in Figure 3a.

In Figure 3b we have shown an arbitrary placement of records in the two MAU's that have been made available in the database for the file F. Now, if a query for retrieval is received in the form "Retrieve records which satisfy the conjunction $(K_1 \wedge K_3)$," then the DBC has to make two MAU accesses. However, if the records are placed in the MAU's grouped according to the occurrence of keywords (K_1, K_2, and K_3) in a record, then the resulting configuration will be as shown in Figure 3c. Such a configuration will result in the retrieval of all records which satisfy the given query in a single access to the mass memory.

The above discussion implies two things: First, the creator of the file has an idea of the type of queries that will be made on the file. Second, the DBC provides him with a mechanism for effectively conveying that knowledge to the DBC. While we, as system designers, cannot predict how much knowledge a creator may have of his file usage, we must ensure that he is provided with an easy yet powerful mechanism to utilize that knowledge to his best advantage. The mechanism that we have adopted and shall describe here is intended for such purpose.

A more elaborate illustration of the clustering and security mechanisms of the DBC is included in the Appendix. In studying the following sections on clustering and security processes, the reader may refer to the Appendix for clarification and demonstration.

B. The Clustering Process

A file is associated (by the file creator) with a single *primary clustering attribute* and one or more *secondary clustering attributes.* The latter attributes are specified in an *order of importance.* In case a record for insertion has more than one secondary clustering attribute, then the most important one is considered during the clustering process. On the other hand, if a record for insertion has no secondary clustering attribute then a null secondary clustering attribute is assumed for clustering purposes.

A keyword whose attribute is the primary (secondary) clustering attribute is called a *primary (secondary) clustering keyword.* Each primary clustering keyword, during file creation time, is associated with a maximum space requirement (in terms of the number of MAU's) which indicates the estimated amount of

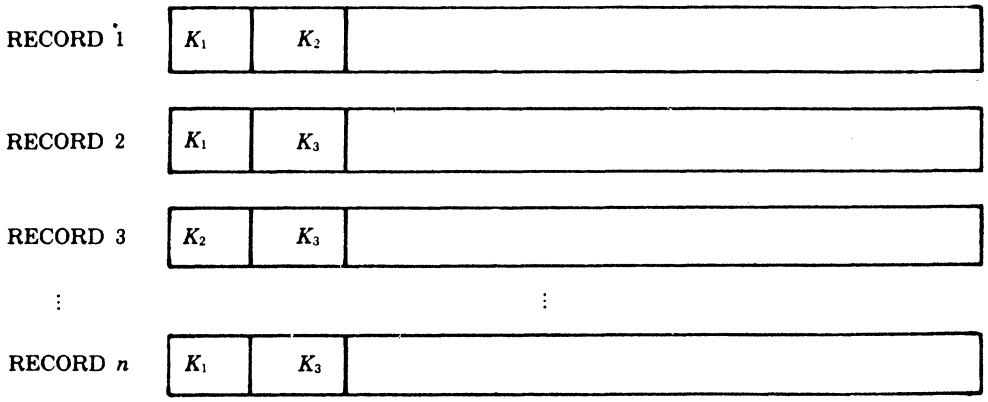


Fig. 3a. Records belonging to a file

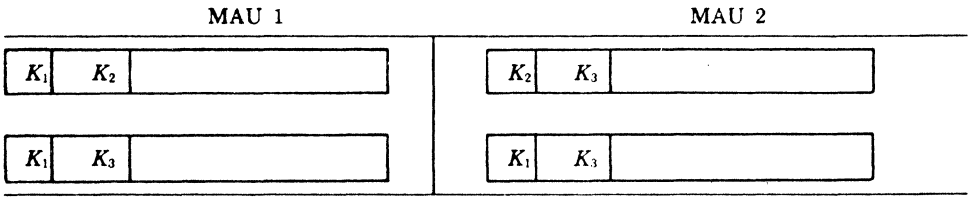


Fig. 3b. An arbitrary assignment of records to MAU's

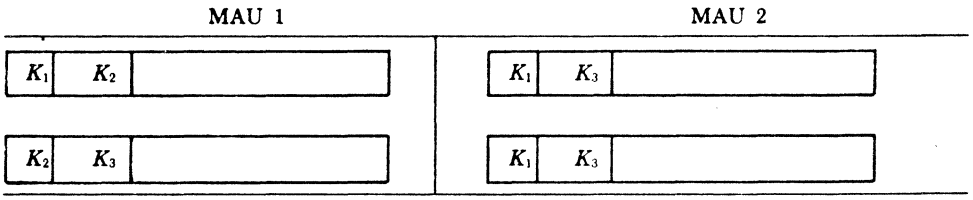


Fig. 3c. An assignment of records to MAU's which results in a minimum number of accesses for certain queries

storage required in the mass memory for all records having this keyword. The estimate is only approximate, but better performance is obtained if the estimate closely reflects the actual maximum storage requirement.

A *cluster c* is defined as the set of records each of which contains exactly the same primary and secondary clustering keywords. Notice that for every record, only one secondary clustering keyword (the most important one) is considered. Each cluster is identified by a unique number called the *cluster identifier.*

When loading into the database a record with primary clustering keyword K and cluster identifier c, an attempt is made to place the record in the same MAU (or one of the MAU's) as is occupied by other records of the cluster c. In case there exist no other records in the database belonging to cluster c or if all MAU's so far used by the cluster are nearly full, then it is first checked whether the number n of MAU's currently used by records with primary clustering keyword

K is less than the corresponding estimated number n_k provided by the file creator. If $n \geq n_k$, then an attempt is made to place the given record in one of the MAU's already being used by records with keyword K. If $n < n_k$, then the given record is loaded in one of the relatively empty MAU's allocated to the file in which the record belongs. In the entire process stated above, whenever there is a choice among a number of MAU's, the most empty MAU is chosen for loading the record. The reason for such a decision is that, in the long run, with such a choice, the clusters may be expected to stay physically together even after many updates to the database. However, there is a provision for database reorganization if clusters tend to disperse within the mass memory.

To support the record clustering operation, a *cluster table* C keeps track of the clusters and MAU's to which those records belong that have the same primary clustering keyword. Each entry in cluster table C is a quadruple

$$(F, K, c, f)$$

where F is a file name, K is a primary clustering keyword of the file, c is a cluster with primary clustering keyword K, and f is an MAU in which there is at least one record of cluster c. Notice that even though a cluster should normally be totally accommodated in a single MAU, the cluster table allows for the event that a cluster is distributed among more than one MAU.

With every file F is associated a *MAU space table* L_F with entries of the form

$$(f, l)$$

where f is an MAU allocated to file F and l is the space available in that MAU.

Given a record to be loaded in the database, it is possible to determine an MAU for loading the record, by making use of the tables C and L_F. Let F be the file to which the record belongs, let K be the record's primary clustering keyword, c_r its cluster identifier, and g_r its length. Let n_k be the estimated number of MAU's required by records with keyword K. The algorithm given in Figure 4 will now determine the number m of MAU in which the given record may be loaded.

In the algorithm, h represents the amount of space remaining in the MAU chosen for consideration. In line 1, ω represents the set of MAU's that are currently being used by the cluster c_r. If this set is not empty (i.e., $|\omega| \neq 0$), then an attempt is made in line 4 to place the record in one of the MAU's in ω. If that is not possible, then it determines in line 7 the set ω' of MAU's used by records with the primary clustering keyword K. If the number of MAU's in ω' is at least as large as the estimated number n_k (as checked in line 8), then an attempt is made to select an MAU from ω'. If that attempt fails, then, in line 11, an MAU is selected from all the MAU's allocated to the file.

C. The Security Process

We now show how the DBC can group records for security purposes. Certain attributes of a file may be designated as *security attributes* by the creator of the file. A *security keyword* is a keyword whose attribute is a security attribute. Each record belonging to a file with security attributes contains a set of security keywords (possibly empty). This set defines a *security atom*. A record is said to belong to a security atom if and only if its security keywords define the security

Input: 1. primary clustering keyword K
2. cluster identifier c_r
3. file identifier F
4. record length g_r
5. estimated number n_k of MAU's required by records with keyword K.

Output: MAU number m in which the given record may be loaded.

```
0.  begin
1.    ω ≡ {f|(F, K, c_r, f) ∈ C}
2.    h ≡ 0
3.    if |ω| ≠ 0 then
4.      (m, h) ≡ max2 ({(f, l)|f ∈ ω and (f, l) ∈ L_F})
5.    if |ω| = 0 or h ≤ g_r then do
6.      begin
7.        ω' = {f|(F, K, c, f) ∈ C and c is any cluster id}
8.        if |ω'| ≥ n_k then
9.          (m, h) ≡ max2 ({(f, l)|f ∈ ω' and (f, l) ∈ L_F})
10.       if |ω'| < n_k or h ≤ g_r then
11.         (m, h) ≡ max2 ({(f, l)|(f, l) ∈ L_F})
12.     end
13. end
```

Note: 1. For any set α, $|\alpha|$ denotes the number of elements in α.
2. The function max2 operates on a nonempty set α of pairs and determines the tuple whose second component is the maximum among the second components of all the tuples of α.

Fig. 4. An algorithm to select an MAU for a record

atom in question. The concept of security atoms is due to [23]. In Figures 5a and 5b, we illustrate this concept by means of an example [5].

In Figure 5a, we notice that there are three different security keywords $K4$, $K5$, $K6$ and there are eighteen different records. Assuming that the attributes of $K4$, $K5$, and $K6$ are all different, there is a theoretical possibility of having $2^3 = 8$ different security atoms, since there are 2^3 different combinations of the security keywords. In Figure 5b, however, we notice that the eighteen given records are partitioned into only six security atoms, on the basis of the combination of security keywords that they contain. Notice that security atom 5 corresponds to the null combination of security keywords.

We observe that security atoms are disjoint sets (i.e., a record belongs to one and only one security atom). This is because each security atom represents a unique combination of security keywords, and any given record can have only one of these combinations of keywords. We further observe that a query made up of security keywords alone will apply to all records of an atom or to none at all. Therefore, if file sanctions are restricted to having queries made up only of security keywords, then it is clear that each file sanction will be applicable to a group of complete security atoms, instead of only to a set of unrelated records. A database capability will now induce a security specification over entire atoms of a file F. That is, every security atom is associated with an access privilege set, thus establishing an *atomic access privilege list* (AAPL) for every user.

Assuming a user query consists of conjunctions which have at least one predicate that corresponds to a directory keyword, for every directory keyword

{K4, K6}	{K1, K3, K5, K6}	{K1, K4, K6}
{K2, K3, K5, K6}	{K2, K3, K4, K5, K6}	{K2, K4}
{K2, K3, K4}	{K1, K3, K4, K6}	{K3, K5, K6}
{K1, K3, K5}	{K3, K5}	{K1, K2, K4}
{K5}	{K1, K2, K4, K5, K6}	{K1, K2}
{K4, K5, K6}	{K2, K5}	{K1, K3}

Fig. 5a. Records (only keywords in the records are shown) to be partitioned into security atoms. Keywords K4, K5, K6 are security keywords

Security Atom 0	Security Atom 1	Security Atom 2
{K4, K6} {K1, K3, K4, K6} {K1, K4, K6}	{K1, K2, K4} {K2, K3, K4} {K2, K4}	{K2, K3, K5, K6} {K1, K3, K5, K6} {K3, K5, K6}

Security Atom 3	Security Atom 4	Security Atom 5
{K2, K5} {K3, K5} {K1, K3, K5} {K5}	{K2, K3, K4, K5, K6} {K4, K5, K6} {K1, K2, K4, K5, K6}	{K1, K2} {K1, K3}

Security atoms and their corresponding security keywords:

Security atom 0	{K4, K6}	Security atom 3	{K5}
Security atom 1	{K4}	Security atom 4	{K4, K5, K6}
Security atom 2	{K5, K6}	Security atom 5	{ }

Fig. 5b. The security atoms of the records of Figure 5a

K, the index terms include not only the MAU numbers, but also the identifiers of the security atoms that contain the records with keyword K.

Pursuing the example given in Figure 5, the directory entries are shown in Figure 6a, where only the security atom numbers are included in the index terms. We have assumed that K2, K3, K4, K5, and K6 are directory keywords. Let us further assume that for a given user, the database capability is such that we get the atomic access privilege list of Figure 6b.

Given a request for access $a2$ on all records containing keywords K1 and K4, the directory is examined for keyword K4. Only atoms 0, 1, and 4 contain K4, as shown in Figure 6a. Looking now at the atomic access privilege list of Figure 6b, it is determined that access $a2$ is not allowed on atoms 0 and 4, but is allowed on atom 1. Therefore, in response to the user's request, only atom 1 is accessed.

A complete example of the security and clustering mechanisms applied to a very small but realistic database is given in the Appendix.

It may be argued that a creator may wish to protect his records at the subatomic level or in a manner which affects portions of different atoms. In such cases, full search of the file sanctions is necessary to determine which of the file sanctions are applicable to an access request. Thus the data model supports two protection mechanisms. The first is geared towards reducing security costs to a minimum, while the other aims at providing maximum flexibility to the user. For the sake of convenience, we shall refer to the protection mechanism based on

Directory Keyword	Security Atom Numbers
*K*2	1, 2, 3, 4, 5
*K*3	0, 1, 2, 3, 4, 5
*K*4	0, 1, 4
*K*5	2, 3, 4
*K*6	0, 2, 4

Fig. 6a. Directory entries showing only the security atom numbers as index terms

Security Atom Number	Access Rights
0	*a*1
1	*a*1, *a*2
2	*a*3
3	*a*1, *a*2, *a*3
4	*a*1
5	*a*1, *a*2, *a*3, *a*4, *a*5

Fig. 6b. Atomic access privilege list of a given user

security atoms as *Type A* protection mechanisms. The other protection mechanism based on full file sanctions search will be called a *Type B* protection mechanism.

From the above discussions, we conclude that the data model specifies two steps by which a record may be evaluated for placement. First, the MAU where the record is to be placed is determined based on the following: the clustering attributes specified by the file creator, the existing clusters in the database, and the clustering keywords of the given record. Secondly, the security atom (if the creator has chosen to specify file sanction in terms of security keywords) to which a record belongs is determined by the set of security keywords appearing in the record.

3.2 The Basic DBC Operations

The basic DBC operations are security enforcement, record insertion, record retrieval, and record deletion. We first give a brief description of these operations and relate them to their supporting components. Then we show in some depth the data structures and algorithms involved in the operations.

3.2.1 The Role of Security Enforcement

The security filter processor (SFP) and the database command and control processor (DBCCP) jointly maintain the database capabilities for the active users of the system. In order for them to correctly enforce a security policy, the proper database capabilities must be provided by the program execution system (PES). A table is kept for each user U with the database capabilities for each active file F. Let each table entry have the form:

$$(F, \{(Q_1, A_1), (Q_2, A_2), \ldots, (Q_n, A_n)\})$$

where each Q_i is a query, each A_i is an access set, and the set of couples is a database capability.

Commands of the form

$$(U, (F, Q), a) \text{ and } (U, (F, R), a)$$

pass through the SFP or the DBCCP depending on the type of protection mechanism chosen by the user. If the creator has chosen Type A protection mechanism, the DBCCP converts the file sanctions into an atomic access privilege list (AAPL). The AAPL has the form

$$(\mathrm{U}, \mathrm{F}, \{(\mathrm{SAN}_1, \mathrm{APD}_1), (\mathrm{SAN}_2, \mathrm{APD}_2), \ldots, (\mathrm{SAN}_P, \mathrm{APD}_P)\})$$

where SAN_i is the name of the ith security atom of the file F and APD_i is the access privilege set associated with SAN_i for the user U. In forming the AAPL, the DBCCP makes use of all the DBC components except the mass memory (MM) and security filter processor (SFP). If the creator of the file has chosen Type B protection mechanism, the SFP takes over the maintenance and usage of the file sanctions. In contrast to Type A protection mechanism, Type B requires the access of all records satisfying a query and then the checking of every record, one at a time, for security violation. Type A mechanism is, therefore, less time-consuming and should be the preferred type of security enforcement mechanism.

Records are sent into the DBC by way of commands of the form

$$(U, (F, R), \text{"insert"}).$$

When such a command is received by the DBCCP, the record to be inserted is checked for security clearance with the aid of the AAPL (Type A protection mechanism) or the file sanctions. If the result of the check indicates that the record may be inserted, then the DBCCP proceeds with the actual insertion process.

When a command $(U, (F, Q), \text{"retrieve"})$ is received by the DBCCP, the query Q undergoes a similar check. If the check is successful, the mass memory is instructed to retrieve the relevant records which form the response set $Q(F)$. Each record in the set $Q(F)$ is tagged with the user identification and file name (F, U, R). If the user has specified Type B protection mechanism, then the retrieved records are subjected to a security check by the security filter processor (SFP) before the records are passed on to the front-end PES. This is because the records may contain keywords (in addition to and including those that are required to satisfy the query Q) which satisfy the query parts of file sanctions. The access privilege sets of such file sanctions then become applicable to the records. As a result, some of the retrieved records may not be passed onto the user. Such a drop in precision is part of the price a user pays for the wide latitude the system provides in specifying security information.

To execute the command $(U, (F, Q), \text{"delete"})$ the query Q is put through a similar check. If the access "delete" is not granted, the command is rejected. If the access is granted, the mass memory is instructed to proceed with the access. In case of Type B protection mechanism, as each record is accessed, it is sent to the SFP for a check against the set of file sanctions. The rationale for this check is the same as the one given for the "retrieve" command. If the check is successful, the mass memory proceeds to delete the record from the database; otherwise, the record is not deleted.

3.2.2 Name-Mapping and the System Components

The retrieve and delete commands both employ the query Q as a parameter. The subsequent processing of Q required for the execution of these commands has the greatest effect in determining the architectural components of the system. We shall now provide an introduction to these components.

A query Q in these commands is in a disjunctive normal form as follows:

$$(P_1^1 \wedge \ldots \wedge P_{n_1}^1) \vee \ldots \vee (P_1^m \wedge \ldots \wedge P_{n_m}^m)$$

where P_j^i are keyword predicates. The ith conjunction of this query is denoted by Q^i. To form the response set $Q(F)$, the mass memory must be given at least two arguments: a query Q and an MAU number f. Given these arguments the mass memory will locate all records in M_f that satisfy the query Q. We had earlier seen that each of the index terms in the directory entry of a keyword defines an MAU number f. In the discussion in Section 3.1 on the security atom concept, it became apparent that the index terms must carry information not only about MAU numbers, but also about security atom numbers. Thus an *augmented directory entry* for a keyword K of file F is defined as

$$D(F, K) \equiv \{(f, s) \mid \exists\, R \ni R \epsilon M_f, R \epsilon \text{ security atom } s, \text{ and } K \epsilon R\}.$$

The pair (f, s) will be called an *augmented index term.* In cases where the user has chosen Type B protection mechanism, the security atom concept is not applicable and the second member of an augmented index term is null. In future discussions, by index terms we will always mean augmented index terms.

In order to protect the security of the database, it is necessary that the response set $Q(F)$ of a query Q should include only those records that not only satisfy Q but also belong to the security atoms on which the required access is allowed to the user. Therefore, the mass memory (MM) must be sent the query Q and a list of index terms (f, s) of which the first component determines an MAU in which one or more records satisfying Q may reside, and the second component indicates to the mass memory (MM) that such records may be accessed if they belong to security atom s. For each unique value f of the first component of the above index terms, the mass memory makes one access to MAU numbered f and finds those records satisfying Q and belonging to one of the security atoms that appear as the second component of the index terms for f.

To obtain the (f, s) pairs for a conjunction Q^i, all index terms for keywords satisfying each P_j^i of the conjunction must be found. Once found, a set intersection operation is performed over the index terms. The resulting index terms are those whose keywords will make the conjunction Q^i true. These index terms are then used as arguments to retrieve records from the mass memory.

An algorithm which forms the response set $Q(F)$ is given in Figure 7. In line 5 of this algorithm, the index terms are fetched from all directory entries $(D(F, K))$ whose keyword K satisfies P_j^i and are placed in a set $\omega(j)$. In lines 3–6, one set $\omega(j)$ is formed for each keyword predicate P_j^i in Q^i. Then in line 7 these sets are intersected to give the set $\theta(i)$. Line 7 carries out an intersection operation since the keyword predicates of Q^i are ANDed together. In lines 1–8, a set $\theta(i)$ is formed for each conjunction Q^i. In line 9, it is ensured that the set to be sent to the mass memory (MM) contains only those security atoms that are accessible

```
0.  begin
1.    For i = 1, 2, ..., m do
2.      begin
3.        For j = 1, 2, ..., n_i do
4.          begin
5.            ω(j) = {(f, s) | K satisfies P_j' and (f, s) ∈ D(F, K)}
6.          end
7.        θ(i) = ∩_{k=1}^{n_i} ω(k)
8.      end
9.    Σ = {(Q^k, f, s) | (f, s) ∈ θ'(k) and given access is allowed on atom s}
10.   Q(F) = ∪_{(Q^k, f, s)∈Σ} {R | R ∈ M_f, R satisfies Q^k and R is in atom s}
11. end;
```

Fig. 7. A name mapping algorithm

by the user. (This job of determining the accessible atoms is done by the control processor DBCCP.) In line 10, the response set is finally formed by the mass memory (MM). In this algorithm, assumption is made that the attributes of all the predicates in the query Q are directory attributes (those attributes on which directory entries are maintained). This is not always the case, but it is required that the attribute of at least one predicate of each conjunction should be a directory attribute. This constraint is easy to maintain (as demonstrated in [1, 2]) since the attributes of security keywords and clustering keywords are also directory attributes. In any case, the query Q may be modified to Q' by deleting all predicates whose attributes are not directory attributes. The modified query Q' may now be used in lines 0-8 of the algorithm (during directory search) and the original query Q may be used in lines 9 and 10 when the actual response set is being determined.

This algorithm shows how the data structures defined in the data model are used for name-mapping. The content-addressability employed by the DBC will, in fact, allow the actual realization of the data structures to be just as simple as those illustrated here.

This algorithm shows us what the structure memory must do. The structure memory must store directory entries and be able to accept a keyword predicate P_i and retrieve all index terms for all keywords which satisfy P_i (as in line 5). Clearly, the structure memory will also have to be able to add, delete, and modify directory entries as well. It also shows us the nature of the structure memory information processing, namely, set manipulation (line 7). These observations help us outline the architecture of the DBC. The DBC contains at least five functionally specialized components: the database command and control processor (DBCCP), the security filter processor (SFP), the mass memory (MM), the structure memory (SM), and the structure memory information processor (SMIP). The DBCCP is responsible for translating DBMS commands into lower level commands for the mass memory and coordinating the actions of the other components. The MM contains the database, the SM stores the directory entries, and the SMIP is a set operation processor. The organization of these components to a first order detail is shown in Figure 8. Two other components, namely the index translation unit (IXU) and the keyword transformation unit (KXU) [20], are needed from the point of view of an efficient physical realization of the DBC.

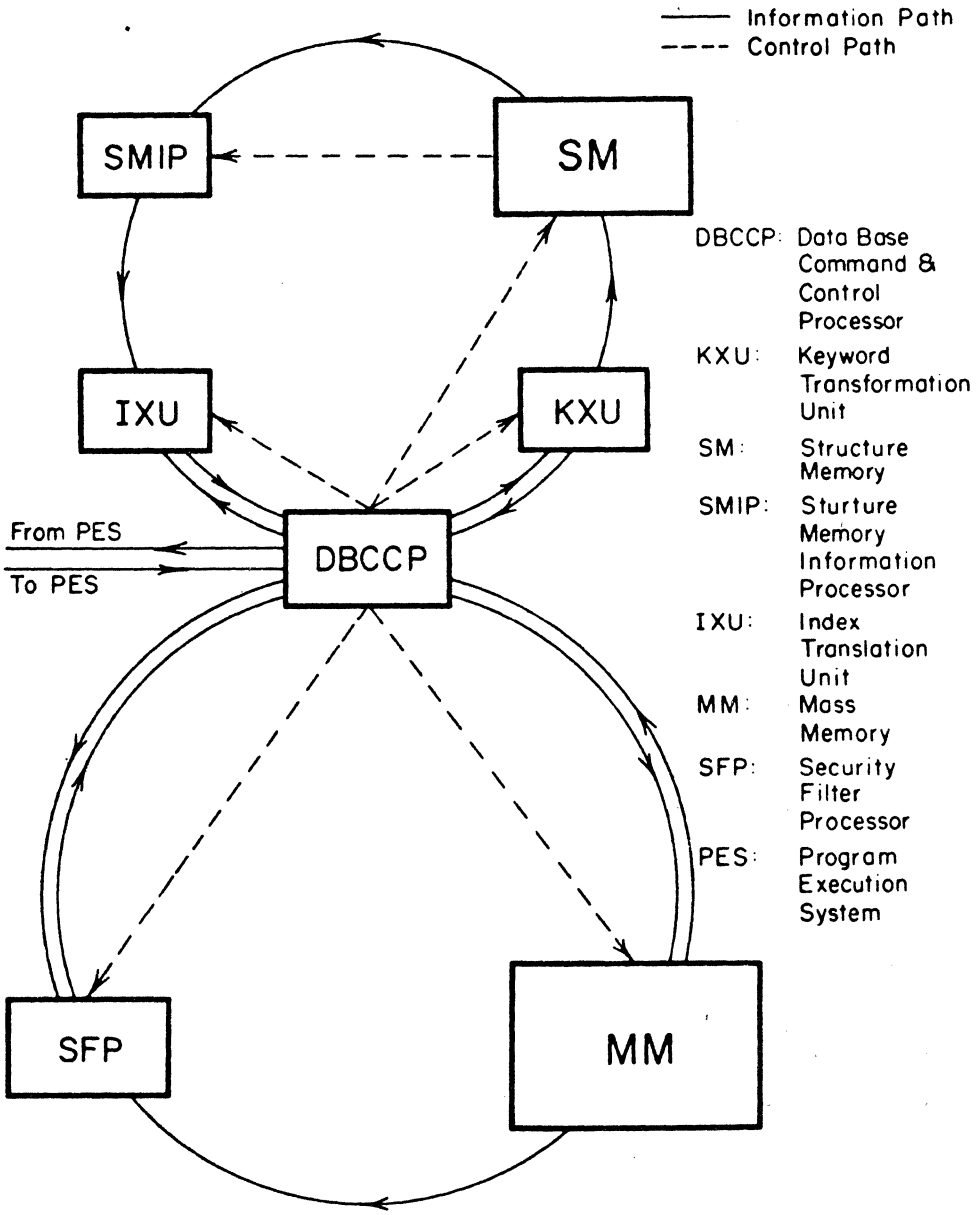


Fig. 8. The architecture of the DBC

3.2.3 The Operation of the SM, SMIP, MM, and DBCCP

The theory of operation continues with an exposition of the operating principles of the structure memory, structure memory information processor (SMIP), the mass memory, and the database command and control processor (DBCCP). The carefully tailored functional characteristics of these components allow them to readily carry out the DBC algorithms. The description of the components that

follows is conceptual in nature; the actual hardware organization used to realize them is given in [16, 19, 20].

A. Structure Memory (SM)

The SM is the repository of the directories of the files in the DB. Each index term (f, s) of $D(F, K)$ is stored in the SM as the tuple (F, K, f, s). The contents of the SM may therefore be viewed as a set (known as *structural memory basis* SMB) of such tuples defining the directories of all files.

The SM retrieve command has the form SM⟨retrieve, (F, P)⟩ where F is a file name and P is a keyword predicate. The command is carried out by constructing a set containing all index terms of each directory entry $D(F, K)$ whose keyword K satisfies P. Formally, the SM executes the command SM⟨retrieve, (F, P)⟩ by outputting the set

$$\{(f, s) \mid (F, K, f, s) \in \text{SMB and } K \text{ satisfies } P\}.$$

The insert command has the form SM⟨insert, (F, K, f, s)⟩ and is executed by adding (f, s) to the set $D(F, K)$. In other words, the insert command is executed by replacing SMB with SMB $\cup$ (F, K, f, s).

The delete command has the form SM⟨delete, (F, K, f, s)⟩ and is executed by removing (f, s) from $D(F, K)$. Formally, the deletion command is executed by replacing SMB with SMB$-(F, K, f, s)$.

To model its operations the SM can be viewed as a PCAM with M content-addressable blocks. The SM partitions the set SMB into N subsets, designated SMB_i, $0 \leq i < N$, where $N \leq M$. Each subset is stored in one or more blocks of the PCAM.

The retrieve command is executed by first applying to predicate P a hash function which maps it into an integer j where $0 \leqq j < N$. Then the set SMB_j is searched by accessing the appropriate block(s) of the PCAM to locate and retrieve the tuples (F, K, f, s) whose keyword K satisfies P.

Insert and delete commands are executed by applying to K a hash function which maps it into an integer j. The tuple (F, K, f, s) is then added to or removed from the subset SMB_j by accessing the appropriate block of the PCAM.

The nature of the hash function will strongly influence the kinds of keyword predicates that may be used by the system. This issue along with a description of how the sets SMB_j are stored in the PCAM and how the SM and its PCAM are realized are addressed in [20].

Consideration is now given to the fact that the SM is a two-level system containing a directory entry storage and a look-aside buffer (LKA). We now extend the aforementioned operations to the two-level SM. Let the directory entry storage be represented by the set SMB defined above. The time required to update this set (i.e., add or delete an element) is fairly long compared to the time required to update, say, a fast access semiconductor RAM. The look-aside buffer allows SM update operations to appear as though they were executed immediately.

The look-aside buffer may be conceptually represented by an ordered set of SM update commands

$$\text{command}_1, \text{command}_2, \ldots, \text{command}_k$$

where command$_i$ precedes command$_{i+1}$ in time. The look-aside buffer has two functions: it acts as a command queue for the SM and it contains the information which allows the SM to appear updated. The look-aside buffer (LKA) would be realized with high-speed random access memory and so its access time would be much less than that of the directory entry storage.

Whenever an update command is received by the SM it is placed in LKA. If an insert (delete) command negates the effect of a previous delete (insert) command then the insert (delete) command is not added to LKA and the negated delete (insert) command is removed from LKA.

To execute a retrieval command, the two-level SM first examines LKA for commands which add index term (f, s) to directory entry $D(F, K)$ whose keyword K satisfies P. All index terms so found are output. Then the set SMB is searched for additional index terms. When an index term (f, s) of a directory entry $D(F, K)$ whose K satisfies P is retrieved from SMB it is checked in the following way: If there is a command in LKA to delete (f, s), then that index term is not output from SM.

B. Structure Memory Information Processor (SMIP)

The SMIP is a processor for set manipulation. Set manipulation operations are performed by maintaining an intermediate set in the SMIP while the argument sets which modify it are passed through the SMIP. The SMIP's intermediate set is designated SW and consists of couples (m, d) called *SMIP data units*. The first part m of the couple is called the *key* and the second part d is called the *data*. Operations are performed on SW by identifying a SMIP data unit and by performing an operation on it. There are two kinds of SMIP commands. The first kind of SMIP command is represented by SMIP$\langle m, g \rangle$ where m is a key and g is a manipulation function. The manipulation function can do two things: first, it can specify how the data part of a SMIP data unit (m, d) with key m should be modified; and second, it can specify what should be done if no SMIP data unit with key m is in SW. When no SMIP data unit with key m is found and no action is specified by g then SMIP takes no action. The second kind of SMIP command has the form SMIP$\langle g \rangle$ where g specifies an action that is to occur.

To illustrate the set manipulation functions, let us show how the SMIP can be used to perform an N-set intersection. Let X_i represent one of these N sets and let x_{ij} represent an element of X_i. The algorithm which performs the intersection is shown in Figure 9.

```
0.  begin
1.  For each element x1j of X1 do
2.    begin
3.      execute the command SMIP<x1j, "create (x1j, 1)">
4.    end
5.  For i = 2, 3, ..., N do
6.    begin
7.      For each element xij of Xi do
8.        begin
9.          execute SMIP<xij, "replace (xij, d) with (xij, d + 1)">
10.       end
11.   end
12. Execute the SMIP<"retrieve the key m from all (m,d) where d = N">
13. end
```

Fig. 9. An N-set intersection algorithm using the SMIP

In lines 1–4 of the algorithm a SMIP data unit of the form $(x_{1j}, 1)$ is created for each element of X_1. In steps 5–11 each element of the sets $X_2, X_3, \ldots, X_n$ is examined and whenever a matching SMIP data unit is found its data part is incremented by 1. When these steps are completed, SW contains SMIP data units which indicate in how many of the sets X_i each element of X_1 appears. Those elements appearing in all sets make up the set $X_1, \ldots, X_n$. In line 12 all such elements are retrieved from the SMIP.

The SMIP is also realized with a PCAM. To model the operation of the SMIP, a PCAM with M content-addressable blocks is used. The SMIP partitions the set SW into N subsets designated SWB_i where $N \leq M$. Each subset is stored in one or more blocks of the PCAM.

The command SMIP$\langle m, g \rangle$ is executed by applying to m a hash function which maps it into an integer j where $0 \leq j < N$. Then SWB_j is searched for a SMIP data unit with the key m. If it is found, g is applied to its data part. If no SMIP data unit is found, then any other action that g specifies is carried out on SWB_j. The command SMIP$\langle g \rangle$ is executed by ordering each block of the SMIP PCAM to perform the operation specified by g.

C. Mass Memory (MM)

The MM is the repository of the database itself. The storage is organized as a partitioned content-addressable memory with enough processors to simultaneously content-address each partition. To access records in the database, queries, MAU numbers, and security atom identifiers are provided to the MM. These addresses and identifiers are provided by the structure memory, the structure memory information processor, and the index translation unit after processing a given query Q.

Mass memory commands have three forms. The first form, MM$\langle a, Q, f, s \rangle$, specifies an access type a, a query Q, an MAU number f, and a security atom identifier s. This form of access request is used for records with Type A security specification. The MM executes the command by performing access a on the records in MAU numbered f satisfying query Q and belonging to atom s. (The atom number being a part of each record, it is easy to check if a record belongs to a given atom.) The second form, MM$\langle a, U, F, Q, f \rangle$, specifies an access type a, a user U, a file F, a query Q, and an MAU number f. This form of access request is used for records with Type B security specification. The MM executes the command by first performing access a on the records in MAU f satisfying query Q, and then sending these records to the security filter processor (SFP). The SFP performs Type B security check on these records and returns to the control processor (DBCCP) only those records that are allowed to be accessed by the given user. Finally, to insert a record R into the MAU f, a command of the form MM$\langle$'insert', $U, R, f\rangle$ is sent to the MM.

D. Database Command and Control Processor (DBCCP)

The DBCCP regulates the operation of the entire system. Its basic function is to receive commands from the front-end program execution system (PES), execute these commands by properly using the various components of the DBC, and sending response data back to the PES. The DBCCP ensures that all commands move smoothly through the system in the form of a pipeline, so that

all the DBC components can be executing specific functions for different commands at any instant of time.

The DBCCP maintains a number of table memories. Among these tables are the cluster table C, the MAU space table L_F for every file F and the database capabilities of the active users. Using these tables, the DBCCP performs some important functions such as clustering of the records (algorithm in Figure 4) and computation of the atomic access privilege lists of the active users. The latter computation is done only once for each combination of file and user.

Complete details of the functions of the various DBC components as well as the detailed algorithms for the execution of the DBC commands may be found in [16, 19, 20].

4. THE TECHNOLOGY OF THE DBC

The database command and control processor (DBCCP), the security filter processor (SFP), and the index translation unit (IXU) are conventional processors that would be specially microprogrammed for their tasks. The structure memory (SM) can also be built with available technology, but the more powerful SM organizations must employ emerging technology.

The SM is most dependent on technological developments. Its PCAM (partitioned content-addressable memory) could be built today by using a fixed-head disk as the storage medium. Each block of the PCAM would be stored on one or more tracks of the disk. The memory would be accessed by reading and searching the track(s) representing a block. This organization would have two limitations: First, the block access time would be relatively slow (5 msec or greater); this is a potential system bottleneck. Second, the PCAM would consist of many relatively small blocks and so only equality predicates could be readily handled by the SM. This is because the small block size implies small hash table buckets which, in turn, implies that the hash function must be used for exact-match searches. This is because inequality searches would cause access to a large number of small blocks.

The rapid development of electronic bulk memory technologies (CCD's, RAM's [12], magnetic bubbles [7, 9], and electron beam memories [18]) may make an all-electronic fixed-head disk replacement available very soon. This would allow the construction of a much faster PCAM-based SM which would not be a bottleneck. An "electronic-disk" PCAM would still, however, have many small blocks and so would suffer from the same keyword predicate limitations as a fixed-head disk PCAM.

The availability of cheap and very powerful microprocessors opens the way to a very powerful PCAM organization. This PCAM consists of a small number of very large content-addressable blocks and is realized by a large number of microprocessor-memory pairs as shown in Figure 10. This kind of PCAM would be capable of supporting a much greater variety of keyword predicates. This is because all keywords of a given attribute could probably be stored in a single PCAM block and so any predicate could be applied to all keywords of that attribute with a single access. In [16], we proposed three design alternatives for the PCAM organization of SM. The design based on magnetic bubbles is of

major-minor loop variation and the design based on charge-coupled devices is of line-addressable RAM organization. Both of these are good for SM of $10^7 \sim 10^8$ bytes in size. For larger SM of 10^9 bytes, the design based on an electron-beam-addressed memory system is given.

The structure memory information processor (SMIP) (see Figure 11) is primarily a processing element and is consequently not limited by memory technology. The small amount of memory required by this component can be realized with current technology. The SMIP achieves very high speed by using many processor-memory pairs to execute operations in parallel. The SMIP is feasible with today's technology and could become quite inexpensive in the future as RAM's and microprocessors become cheaper.

The mass memory (see Figure 12) uses a moving-head disk to realize a PCAM. Each cylinder of the disk represents one PCAM block. For high performance, all of the data on a cylinder is accessed in parallel, and is searched in a single disk revolution. The mass memory therefore uses multiple read/write assembly registers and a set of fast processing units (the number of units is equal to the number of tracks of a cylinder). This requires modification and improvements of current technology. A detailed description of the logical and physical structure of the mass memory is given in [19].

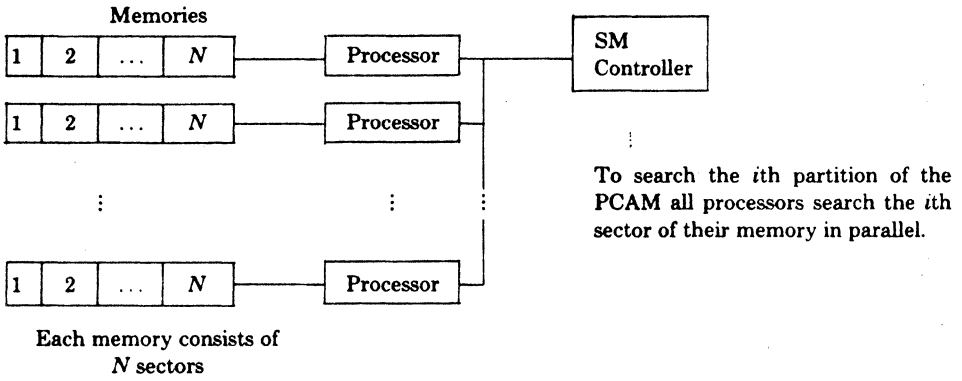


Fig. 10. The architecture of the structure memory

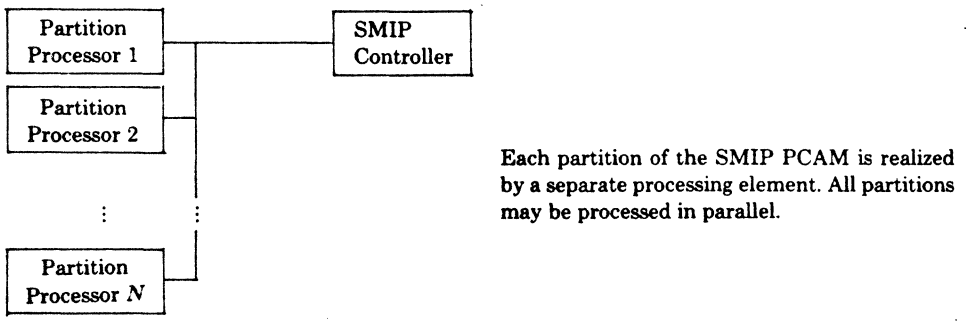


Fig. 11. The architecture of the SMIP

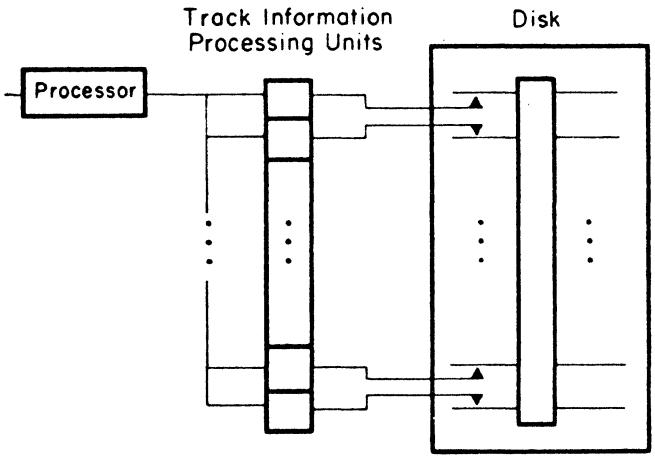


Fig. 12. The architecture of the MM

5. CONCLUDING REMARKS

The emergence of large data bases and complex software for the conventional database management systems has prompted the search for hardware solutions for database management. It is expected that database machines can contribute to a considerable improvement in reliability and performance of the existing database management systems [4].

This paper lays down the conceptual framework for the design of such a machine, called the DBC. The DBC design allows for a number of functionally specialized modules which can all work concurrently. A large degree of parallelism is provided within each module (including the mass memory) by employing a set of processors to simultaneously perform a content-search operation and by providing tracks-in-parallel read-out. Yet, the number of such processors is kept low by having a PCAM (partitioned content-addressable memory) design for the major DBC memory components, namely the mass memory (MM) and the structure memory (SM).

The DBC directly implements the attribute-based data model and a very powerful query language based on Boolean expressions of keyword predicates. The DBC, thereby, provides a natural way of expressing database management needs.

Since the mass memory of the DBC is made up of content-addressable partitions called minimal access units (or MAU's), it is necessary that the more frequent requests to the DBC be answered with the fewest possible MAU accesses. This is made possible by the application of the data clustering mechanism. A file creator is allowed to use his knowledge of the characteristics of the records and the frequency of the various types of requests, in determining a set of clustering attributes. Since MAU's are normally very large (at least a disk cylinder), few files will ever require a storage of more than 100 to 1000 MAU's. Two levels of clustering, therefore, in the form of a primary clustering attribute and a secondary clustering attribute should be more than adequate for all

purposes. Effective application of this clustering mechanism has already been made in the simulation of network [1] and relational [2] databases.

The DBC also provides a content-based security enforcement mechansim. Each file is logically partitioned into a number of security atoms, which are defined in terms of security keywords. A database capability is then defined for every user of the file. The database capability of a user consists of a set of file sanctions, which, in turn, consist of pairs of the form (query, access privilege set). Using the database capability of a user, the DBC can determine the user's atomic access privilege list. This is done once for every user of the file. Later, during file access, it is easy to determine for each query given by a user what atoms the query might refer to. The user is then allowed the requested access to only those atoms that are permitted by his atomic access privilege list.

Detailed specifications of the data and instruction formats of the database computer and its components, the structure, speeds, and capacities of the components and the technology required to build the machine are given in [16, 19, 20]. It is our belief that the architectural principles used in the DBC do not require distant technology and so can be realized in the near future. Preliminary studies on how the DBC should support higher level data models such as the hierarchical, network, and relational have been completed [1, 2, 17]. In all these studies it has been observed that compared to a conventional system supporting a particular data model, the DBC shows a much improved performance. In fact, a general conclusion may be drawn from these studies that even though the mass memory requirement of the DBC is comparable to or slightly greater than that of a conventional system, the directory storage requirement as well as the query execution time (or time to execute a database transaction) of the DBC are one or more orders of magnitude less than those of a conventional system.

APPENDIX: AN ILLUSTRATION OF THE SECURITY AND CLUSTERING MECHANISMS

We shall illustrate here, with an example, the manner in which records are clustered. We also illustrate how the same records are grouped into security atoms in order to protect them from unauthorized access. Even though the DBC allows range specifications (instead of only simple keywords) as directory entries and as security and clustering descriptors, we shall consider, for simplicity, only simple keywords for such purposes. Further, the Type A protection mechanism being the more important and interesting one, our example will assume the use of the Type A protection mechanism alone. The reader may be forewarned that the sample database being necessarily small, the security atoms and clusters will be quite small. What we hope to achieve, however, is to provide the reader with a clearer utilization of the concepts presented earlier in this paper.

(1) A Sample Relational Database

We consider for illustration a relational database with two relations EMP(ENO, NAME, DNO, JOB, PNO), and DEPT(DNO, MGR, FLOOR), where the column names, i.e., attributes of each relation, are enclosed in parentheses. Every employee record, i.e., a row in relation EMP, consists of an employee number ENO,

an employer NAME, a department number DNO, a JOB designation, and a project number PNO. Every department record, i.e., a row in relation DEPT, consists of a department number DNO, a manager name MGR, and the FLOOR in which the department is located.

A. Data Adjacency Requirements—Based on the frequency of various types of user requests, it has been determined that all records of a single relation are to be kept physically close to one another. Furthermore, secondary clustering may be done in terms of the JOB and DNO attributes. Records of the two relations are to be placed in a single file *F*. We assume that the file creator has specified the expected storage requirement for EMP relation to be 4 MAU's and for DEPT relation 2 MAU's. In order to keep the discussion simple, it is assumed that an MAU can accommodate up to five records (instead of some realistic size, such as 500,000 bytes).

B. Basic Security Requirements—It has been determined that all records belonging to an engineer or manager, to project number 10 or 20, or to department 100 are security sensitive.

C. The DBC Representation of the Sample Relational Database—The relational database is given in Figure 13. In representing a relation in the DBC database, a relational tuple is converted to a set of DBC keywords, i.e., attribute-value pairs, and a special keyword is created for the relation name. For example,

EMP Relation

	ENO	NAME	DNO	JOB	PNO
R1.	1	HAYES	100	MGR	10
R2.	2	NAYAK	100	ENGG	20
R3.	3	ROSEN	100	ENGG	20
R4.	4	KERNS	100	TECH	10
R5.	5	GROVE	100	SEC	10
R6.	6	PERRY	100	SEC	20
R7.	7	GHOSH	200	MGR	30
R8.	8	SLOAN	200	ENGG	30
R9.	9	PARDO	300	MGR	30
R10.	10	PRICE	300	TECH	20
R11.	11	WHITE	300	TECH	30
R12.	12	KLINE	300	SEC	30
R13.	13	HSIAO	400	MGR	40
R14.	14	PRATT	400	ENGG	40
R15.	15	BOONE	400	SEC	40

DEPT Relation

	DNO	MGR	FLOOR
R16.	100	HAYES	1
R17.	100	NKOMO	2
R18.	200	GHOSH	1
R19.	200	GHOSH	2
R20.	300	PARDO	1
R21.	400	HSIAO	1
R22.	400	HSIAO	2

Fig. 13. A sample relational database

the first tuple in the EMP relation is represented as a DBC record:

(<RELATION, EMP>, <ENO, 1>, <NAME, HAYES>,
<DNO, 100>, <JOB, MGR>, <PNO, 10>).

For the purpose of later reference we call the tuples and equivalent DBC records as R1, R2, R3, etc.

(2) New Users and Their Access Privileges

Considering a specific user U of the database, it is necessary that he be allowed no access to the employee records belonging to a manager, only 'read' access to all records with DNO=100 and PNO=20, and only 'read-and-modify' access to all records with JOB=ENGG.

(3) The Process of Creating a Database in the DBC

The given records, in the form of attribute-value pairs, are loaded into the mass memory (MM). This process starts with the front-end program execution system (PES) specifying the names of the files to be created, their clustering attributes, storage requirements, and their security keywords and other directory keywords. The database command and control processor (DBCCP) keeps track of them by means of an attribute table.

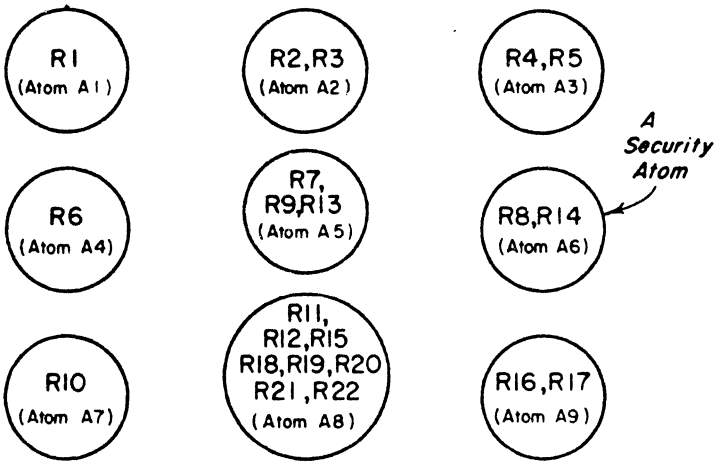
The records are then supplied by the PES, one at a time, to the DBCCP. The DBCCP determines, for each record, the security atom and cluster in which it belongs and the MAU in which the record should be placed. The record is then physically placed in the MAU determined. While the record is being loaded in the mass memory (MM), the DBCCP updates the directory in the structure memory (SM). A new entry is created in the directory for every directory keyword appearing in the record in consideration.

For this example, the processes are summarized in the following paragraphs:

(A) Determining the Security Atom for a Record—The combination of security keywords in the given record uniquely determines the security atom in which it should belong. Any new security atom generated during the loading process is assigned a unique security atom identification (id). The security keywords for the given file are <JOB, MGR>, <JOB, ENGG>, <PNO, 10>, <PNO, 20>, and <DNO, 100>. The security atoms created for the database are shown in Figure 14.

(B) Determining the Cluster id of a Record—The combination of the primary clustering keyword and the most important secondary clustering keyword uniquely determine the number of the cluster in which the record should belong. For the given file, the primary clustering attribute is RELATION and the secondary clustering attributes, in order of importance, are JOB and DNO. The clusters created for the database are shown in Figure 15.

(C) Assigning an MAU to a Record—Every record is stored in an MAU determined by the cluster to which it belongs and the current space availability in the MAU's. In the example, the space requirements of the primary clustering keywords <RELATION, EMP> and <RELATION, DEPT> are 4 MAU's and 2 MAU's, respectively. The MAU number for each record, loaded in the order



The security keywords given are ⟨JOB, MGR⟩, ⟨JOB, ENGG⟩, ⟨PNO, 10⟩, ⟨PNO, 20⟩ and ⟨DNO, 100⟩.

Security Atom id	Security Keywords in Atom		
A1	⟨DNO, 100⟩,	⟨JOB, MGR⟩,	⟨PNO, 10⟩
A2	⟨DNO, 100⟩,	⟨JOB, ENGG⟩,	⟨PNO, 20⟩
A3	⟨DNO, 100⟩,	⟨PNO, 10⟩	
A4	⟨DNO, 100⟩,	⟨PNO, 20⟩	
A5	⟨JOB, MGR⟩		
A6	⟨JOB, ENGG⟩		
A7	⟨PNO, 20⟩		
A8			
A9	⟨DNO, 100⟩		

Notice that atom A8 corresponds to the null combination of security keywords.

Fig. 14. Security atoms formed by the records of Figure 13

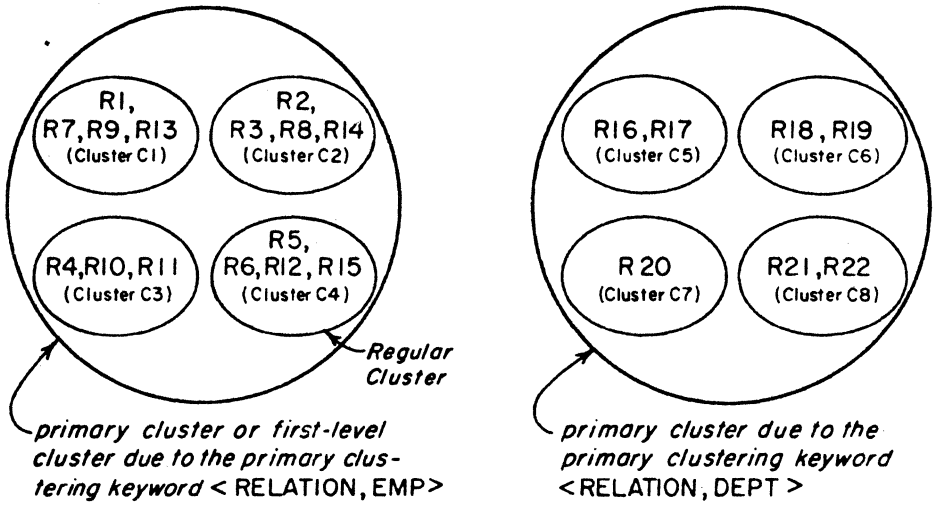
R1, R2, R3, etc., is determined by means of the algorithm of Figure 4. After creation of the database, therefore, the MAU map appears as shown in Figure 16.

(D) Creating the Keyword Directory—The security and clustering keywords are specified to be the only directory keywords for the given file. For each directory keyword in a record, an entry is made in the structure memory (SM). The index term of the entry is of the form (f, s) where f is the number of the MAU in which the record is stored and s is the security atom to which the record belongs. The keyword directory created for the sample database is shown in Figure 17.

(4) Access Privileges of the User *U*

Consider now the security requirements of the particular user U. His database capability consists of the following file sanctions:

((JOB=MGR), no-access)
(((DNO=100) ∧ (PNO=20)), read-only)
((JOB=ENGG), read-and-modify)



Cluster id	Clustering Keywords	
C1	⟨RELATION, EMP⟩,	⟨JOB, MGR⟩
C2	⟨RELATION, EMP⟩,	⟨JOB, ENGG⟩
C3	⟨RELATION, EMP⟩,	⟨JOB, TECH⟩
C4	⟨RELATION, EMP⟩,	⟨JOB, SEC⟩
C5	⟨RELATION, DEPT⟩,	⟨DNO, 100⟩
C6	⟨RELATION, DEPT⟩,	⟨DNO, 200⟩
C7	⟨RELATION, DEPT⟩,	⟨DNO, 300⟩
C8	⟨RELATION, DEPT⟩,	⟨DNO, 400⟩

It has been given that the primary clustering attribute is RELATION and the secondary clustering attributes are JOB and DNO.

Fig. 15. Clusters formed by the records of Figure 13

The user has all access rights on records that do not violate any of the file sanctions. Using the security atom definitions and the database capability of the user, the DBCCP creates an atomic access privilege list for the user according to the discussion in Section 3.1. This is done before he starts accessing the file. The atomic access privilege list of the user is shown in Figure 18. The atom definitions (i.e., the list of security keywords of the atom) are also included in the list, so that the reader can make convenient reference. The user's file sanctions are also reproduced in the figure.

(5) Executing the Requests of User *U*

We finally consider the role of the various data structures, such as the atomic access privilege list and the keyword directory, in the retrieval of records from the database. Three typical access requests, made by user *U*, will illustrate this process.

Example 1: Modify the project number of all secretaries to 20.
This is an example of a request where the physical clustering of the records becomes very useful for efficient execution of the request. The request requires that all records be first retrieved that satisfy the query

((RELATION = EMP) ∧ (JOB = SEC)).

• Since both the predicates of the query correspond to directory keywords, the index terms for each predicate are easily determined by referencing the directory (Figure 17). The two sets of index terms are now intersected by the structure memory information

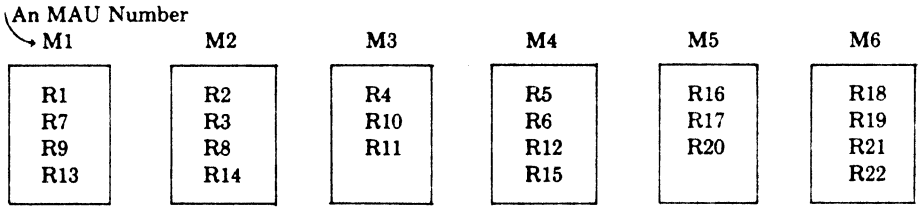


Fig. 16. MAU map of file *F*

Keyword	Index Terms
<RELATION, EMP>	(M1,A1), (M1,A5), (M2,A2), (M2,A6), (M3,A3), (M3,A7), (M3,A8), (M4,A3), (M4,A4), (M4,A8)
<RELATION, DEPT>	(M5,A8), (M5,A9), (M6,A8)
<JOB, MGR>	(M1,A1), (M1,A5)
<JOB, ENGG>	(M2,A2), (M2,A6)
<JOB, TECH>	(M3,A3), (M3,A7), (M3,A8)
<JOB, SEC>	(M4,A3), (M4,A4), (M4,A8)
<DNO, 100>	(M1,A1), (M2,A2), (M3,A3), (M4,A3), (M4,A4), (M5,A9)
<DNO, 200>	(M1,A5), (M2,A6), (M6,A8)
<DNO, 300>	(M1,A5), (M3,A7), (M3,A8), (M4,A8), (M5,A8)
<DNO, 400>	(M1,A5), (M2,A6), (M4,A8), (M6,A8)
<PNO, 10>	(M1,A1), (M3,A3), (M4,A3)
<PNO, 20>	(M2,A2), (M3,A7), (M4,A4)

Fig. 17. Keyword directory of file *F*

File sanctions of user *U*

((JOB=MGR), no-access)
(((DNO=100) ∧ (PNO=20)), read-only)
((JOB=ENGG), read-and-modify)

Atomic access privilege list of user *U*

Security Atom	Security Keywords	Accesses Authorized
A1	<DNO, 100>, <JOB, MGR>, <PNO, 10>	no-access
A2	<DNO, 100>, <JOB, ENGG>, <PNO, 20>	read-and-modify
A3	<DNO, 100>, <PNO, 10>	all accesses
A4	<DNO, 100>, <PNO, 20>	read-only
A5	<JOB, MGR>	no-access
A6	<JOB, ENGG>	read-and-modify
A7	<PNO, 20>	all accesses
A8		all accesses
A9	<DNO, 100>	all accesses

Fig. 18. File sanctions and atomic access privilege list of user *U*

processor (SMIP) to produce the following index terms. The intersection algorithm implemented in the SMIP is listed in Figure 9.

(M4, A3), (M4, A4), (M4, A8).

Notice, as an aside, that even though there are many records satisfying the query, only one MAU, namely M4, contains all of them. This is because of clustering by the RELATION and JOB attributes. Coming back to the problem, we notice that three atoms A3, A4, and A8 need to be accessed. However, looking at the atomic access privilege list (Figure 18), it is determined that the user is allowed the 'modify' access only on atoms A3 and A8. Hence, the index terms sent to the mass memory (together with the query) are:

(M4,A3) and (M4,A8).

After the response set is delivered by the mass memory, the records may be modified and reinserted in the database.

Example 2: Read the records of all employees in project 10.

This is an example of a request where the response set is not clustered. The request requires that all records be retrieved that satisfy the query

((RELATION = EMP) $\wedge$ (PNO = 10)).

Once again, since both predicates correspond to directory keywords, two sets of index terms are retrieved from the directory and intersected to produce the list:

(M1, A1), (M3, A3), (M4, A3).

Since 'read' access is not allowed on atom A1, only the following index terms are sent to the mass memory (together with the query):

(M3, A3) and (M4, A3).

Example 3: Find the employee record of KERNS.

This is an example of a request where only the first-level clustering, by attribute RELATION, can be used. The request requires that all records be retrieved that satisfy the query:

((RELATION = EMP) $\wedge$ (NAME = KERNS)).

The second predicate does not correspond to a directory keyword, hence only the first one is used for directory search. The index terms for <RELATION, EMP> are found, and after removing the terms with atoms that are not permitted for 'read' access, the following index terms (together with the query) are sent to the MM:

(M2, A2), (M2, A6), (M3, A3), (M3, A7), (M3, A8), (M4, A3), (M4, A4), (M4, A8).

ACKNOWLEDGMENTS

The work reported here is the result of a research initiated by D. K. Hsiao, contributed first by R. I. Baum, expanded by K. Kannan, and continued by J. Banerjee under the supervision of D. K. Hsiao. The entire work was conducted at The Ohio State University and supported by The Office of Naval Research through contract N00014-75-C-0573. The authors would like to thank K. Kannan for his contribution to the database computer project. A major portion of this paper is derived from a project report [5] available also through NTIS under AD-A034154. The authors would also like to thank the referees whose comments and suggestions helped to improve the manuscript considerably. Thanks also are due D. Kuck who as the editor devoted considerable patience and care in handling the manuscript.

REFERENCES

1. BANERJEE, J., HSIAO, D.K., AND KERR, D.S. DBC software requirements for supporting network databases. Tech. Rep. OSU-CISRC-TR-77-4, Ohio State U., Columbus, Ohio, June 1977; also available in BANERJEE, J., AND HSIAO, D. K. A methodology for supporting existing CODASYL database with new database machines. Proc. ACM 78 Conf., Washington, D. C., Dec. 1978.
2. BANERJEE, J., HSIAO, D.K., AND KERR, D.S. DBC software requirements for supporting relational databases. Tech. Rep. OSU-CISRC-TR-77-7, Ohio State U., Columbus, Ohio, Sept. 1977; also available in two papers by J. Banerjee and D.K. Hsiao: The use of a database computer for supporting relational databases, Proc. Fourth Comptr. Architecture Workshop for Non-Numerical Processing, Syracuse, N.Y., Aug. 1978, and Performance study of a database machine in supporting relational databases, Proc. Fourth Int. Conf. Very Large Data Bases, Berlin, Germany, Sept. 1978.
3. BANERJEE, J., HSIAO, D.K., AND NG, F.K. Data network—a computer network of general-purpose front-end computers and special-purpose backend database machines. Proc. Int. Symp. on Comptr. Network Protocols, A. Danthine, Ed., Liege, Belgium, Feb. 1978, pp. D6-1–D6-12.
4. BAUM, R.I., AND HSIAO, D.K. Database computers—a step towards data utilities. *IEEE Trans. Comptrs. C-25*, 12 (Dec. 1976), 1254–1259.
5. BAUM, R.I., HSIAO, D.K., AND KANNAN, K. The architecture of a database computer, Pt. I: Concepts & capabilities. Tech. Rep. OSU-CISRC-TR-76-1, Ohio State U., Columbus, Ohio, Sept. 1976.
6. BERRA, P.B., AND SINGHANIA, A.K. A multiple associative memory organization for pipelining a directory to a very large data base. Dig. of Papers COMPCON 76, Washington, D.C., pp. 109–112.
7. BOBECK, A.H., BONYHARD, P.I., AND GEUSIC, J.E. Magnetic bubbles—an emerging new memory technology. *Proc. IEEE 63*, 8 (Aug. 1975), 1176–1195.
8. CANADAY, R.H., HARRISON, R.D., IVIE, E.L., RYDER, J.L., AND WEHR, L.A. A back-end computer for data management. *Comm. ACM 17*, 10 (Oct. 1974), 575–582.
9. COHEN, M.S., AND CHANG, H. The frontier of magnetic bubble technology. *Proc. IEEE 63*, 8 (Aug. 1975), 1196–1206.
10. COULOURIS, G.F., EVANS, J.M., AND MITCHELL, R.W. Towards content addressing in data bases. *Comptr. J. 15*, 2 (Feb. 1972), 95–98.
11. DEFIORE, C.R., AND BERRA, P.B. A data management system utilizing an associative memory. Proc. AFIPS 1973 NCC, Vol. 42, AFIPS Press, Montvale, N.J., pp. 181–185.
12. HODGES, D.A. A review and projection of semiconductor components for digital storage. *Proc. IEEE 63*, 8 (Aug. 1975), 1136–1147.
13. HOLLAAR, L.A. A list merging processor for information retrieval systems. Presented at Workshop on Architecture for Non-Numerical Processing, Dallas, Tex., Oct. 1974.
14. HSIAO, D.K. *Systems Programming—Concepts of Operating and Data Base Systems.* Addison-Wesley, Reading, Mass., 1975, chap. 6.
15. HSIAO, D.K., AND HARARY, F. A formal system for information retrieval from files. *Comm. ACM 13*, 4 (April 1970), 266. Corrigenda. *Comm. ACM 13*, 6 (June 1970).
16. HSIAO, D.K., KANNAN, K., AND KERR, D.S. Structure memory designs for a database computer. *Proc. ACM 77 Conf.*, Seattle, Wash., Oct. 1977, pp. 343–350.

17. HSIAO, D.K., KERR, D.S., AND NG, F.K. DBC software requirements for supporting hierarchical databases. Tech. Rep. OSU-CISRC-TR-77-1, Ohio State U., Columbus, Ohio, April 1977.
18. HUGHES, W.C., et al. A semiconductor nonvolatile electron-beam accessed mass memory. *Proc. IEEE 63,* 8 (Aug. 1975), 1230–1240.
19. KANNAN, K. The design of a mass memory for a database computer. *Proc. Fifth Annual Symp. Comptr. Architecture,* Palo Alto, Calif., April 1978, pp. 44–50.
20. KANNAN, K., HSIAO, D.K., AND KERR, D.S. A microprogrammed keyword transformation unit for a database computer. Proc. Tenth Annual Workshop Microprogramming, Niagra Falls, N.Y., Oct. 1977, pp. 71–79.
21. LIN, C.S., SMITH, D.C.P., AND SMITH, J.M. The design of a rotating associative memory for relational database applications. *ACM Trans. Database Syst. 1,* 1 (March 1976), 53–65.
22. MARILL, T., AND STERN, D. The datacomputer—a network data utility. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 389–395.
23. MCCAULEY, III, E.J. Highly secure attribute-based file organization. Proc. Second USA-Japan Comptr. Conf., Aug. 1975, pp. 497–501.
24. MINSKY, N. Rotating storage devices as partially associative memories. Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N.J., pp. 587–596.
25. MOULDER, R. An implementation of a data management system on an associative processor. Proc. AFIPS 1973 NCC, Vol. 42, AFIPS Press, Montvale, N.J., pp. 171–176.
26. OZKARAHAN, E.A., SCHUSTER, S.A., AND SMITH, K.C. RAP—associative processor for data base management. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 379–388.
27. STELLHORN, W.H. A specialized computer for information retrieval. Rep. No. R-74-637, Dept. Comptr. Sci., U. of Illinois, Urbana, Ill., Oct. 1974.
28. SU, S.Y.W., AND LIPOVSKI, G.J. CASSM: A cellular system for very large data bases. Proc. Int. Conf. Very Large Data Bases, Sept. 1975, pp. 456–472.

Received July 1977; revised April 1978; rerevised August 1978

DBC - A DATABASE COMPUTER FOR VERY LARGE DATABASES

Jayanta Banerjee and David K. Hsiao
The Ohio State University

Krishnamurthi Kannan
IBM Thomas J. Watson Research Center

ABSTRACT

Design considerations of a database computer are presented in this paper. The overall architecture of the computer as well as the organization of its individual components are discussed. Several key concepts which are vital to database management are incorporated in the design and organization of the components. The concepts of tracks-in-parallel read-out and logic-per-some-track processing are provided in an on-line database store for the purpose of achieving high-volume content-addressability. The use of auxiliary information about the database for access precision and control has resulted in the design of a structure memory, an array of content-addressable memory and processor pairs for large collections of indices. The choice of technologies for the implementation of these components are considered in terms of their cost and performance. Modified moving-head disk technology is chosen in order to support the very large on-line database store. Emerging technologies such as magnetic bubles and CCDs are chosen for the structure memory on the basis of their matching performance with the on-line database store and their capability for parallel-in-blocks-and-serial-within-block processing. Five other important components are also discussed in the paper. Their role in the database computer and relationship with the structure memory and on-line database store are delineated.

The database computer is meant to be a back-end machine which interfaces with the front-end general-purpose computers. To this end, the paper attempts to show that the database computer provides a very high-level instruction repertoire for interfacing with the front-end, a set of elaborate security mechanisms, and an effective cluster mechanism. These built-in capabilities tend to allow the database computer to support existing and new database applications with better throughput and higher security.

1. BASIC DESIGN GOALS

Database machines are special-purpose computers which may have been prompted [1] in recent years by the following factors:

(1) The change of data-processing-oriented information management to database-management-oriented information management -- The traditional data processing is essentially a closed-shop operation which is supported and managed by computer professionals. The user of information must interface with the computer professionals for problem solving and decision making. Essentially, the computer professional's attempt to understand the problems and needs of the user, devise programs to solve the problems, run the programs for the user, and return the results to the user. This entire cycle is repeated many times until the information needs of the user are met. Modern database management is not meant to be a closed-shop operation. Instead, it allows multiple users to have access to a shared database. Although computer professionals are still needed to support and manage the facility, they are primarily involved in the design and creation of the shareable database, development of high-level data languages and software aids for the ease of user-database interactions, and incorporation of effective access control measure and reliable security provisions so that access to sensitive information can be regulated and protected. This multi-access operation requires considerable new software development and hardware support.

This change also requires that the off-line mode of operations be replaced by an on-line one. In other words, the software must be capable of supporting on-line databases and interacting with the user in real-time.

(2) The availability and variety of memory and processor technology -- Typically, the software-laden database management system has been large in size and complex in structure which not only overtaxes the hosting hardware but also overshadows the hosting operating system. Excluding the database, they are still large relative to the hosting operating systems and thereby utilize considerable main memory and auxiliary storage as the operating systems do. It is therefore not surprising that attempts have been made to remove the software-laden database management system from the general-purpose computer and replace it with a specialized machine [3,4,6,7,8]. In addition, we have at present, a wide choice of emerging technology such as charge-coupled devices, magnetic bubles, electron beam addressable memories, dynamic RAMs and modifiable moving-head disks [17,18,20]. It may thus be possible to design and configure a special-purpose computer which can perform database management tasks cost-effectively. By eliminating much of its software, the database management system can perhaps now interface with the host computer and the host operating system more reliably with better response time and throughput.

The database computer (DBC) [9,10,11] to be discussed in this paper is an attempt to incorporate as much specialized hardware for data

management as possible. As a back-end machine, the DBC attempts to achieve high performance and low cost. There are five basic goals in the design of the database computer (DBC). The first goal is to design it with the capability of handling a very large on-line database of 10^{10} bytes or beyond, since special-purpose machines are not likely to be cost-effective for small databases. The second goal is to build the database computer now. This implies that only emerging technology and modifications of the existing technology may be considered for the hardware design. No reliance is to be placed on distant technology. The third goal is that the DBC must compete favorably with existing software-laden database management systems (which are run on general-purpose computers) in terms of system throughput and cost of database storage. The fourth goal is to design at the outset a security mechanism as an integral part of the DBC, since a modern database must have security and control for sharing and protection. The final goal is that the DBC, working as a back-end computer, must provide a repertoire of very high-level commands to interface with the front-end computers and support different types of database management applications (in particular, those applications utilizing the hierarchical [12], CODASYL [13] and relational [5] data models). As we progress through the remaining sections in this paper, we will attempt to show how the DBC design meets the first four goals. We will not elaborate on the DBC design in meeting the fifth goal. This study is voluminous [14,15,16] and is being published elsewhere.

2. AN OVERVIEW OF THE DBC ARCHITECTURE

Figure 1 is a complete diagram of the major DBC components. The DBC acts as a back-end machine to one or more front-end general-purpose computers, which are jointly referred to as the program execution system (PES). Users' programs reside in the PES, and are executed by the PES using the DBC as one of its various resources. The PES communicates with the DBC by way of DBC commands and the DBC responds either by returning a group of records or parts of such records (i.e., the response set), or by indicating successful or unsuccessful execution of a command.

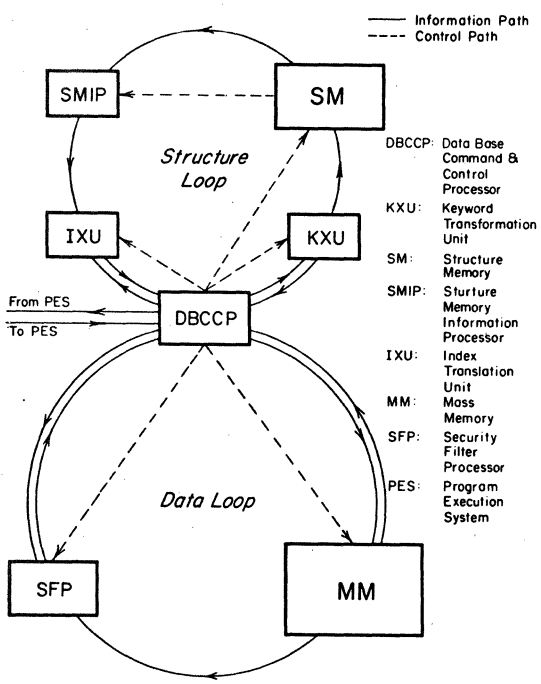


FIGURE 1. Architecture of the DBC

The DBC makes use of two loops of processors and memories in executing the commands. The data loop, which consists of the database command and control processor (DBCCP), mass memory (MM), and security filter processor (SFP), is used for storing and accessing the database, for post-processing of retrieved records and for enforcing field-level security (known as the type B control). The structure loop, which consists of the database command and control processor (DBCCP), keyword transformation unit (KXU), structure memory (SM), structure memory information processor (SMIP) and index translation unit (IXU), is used for limiting the mass memory search space (through the determination of cylinder numbers), for determining the authorized records for accesses (known as the type A control) and for clustering records received for insertion into the database.

The DBC design exploits both existing and emerging technologies. The on-line mass memory (MM) is made from moving-head disks, perhaps the least expensive of all large-capacity on-line storage devices. The disks, however, are modified to allow parallel read-out of an entire cylinder in one revolution time, instead of one track at a time. The parallel readout capability of the DBC provides rapid access to a relatively large block of data. This data can now be content-addressed simultaneously by a set of track information processors (TIPs) in the same revolution. It seems adequate that access is limited to one or a few cylinders, since single user transactions seldom refer to data beyond mega-bytes in size. As long as data is not physically scattered, sweeping of a large number of disk cylinders can be avoided. The physical dispersion of related data is prevented by a built-in clustering mechanism in the database command and control processor (DBCCP) which uses information provided by the creators of the database via the program execution system (PES).

The DBC needs the use of some structural information about the database. Without the help of such information, every request would require all the cylinders (that constitute the database) to be accessed whether there is any clustering or not. Furthermore, pre-processing of the user's access authorization in determining well-compartmentalized data aggregates for security purpose may not be possible (known as type A control). Although both the access and security-related information are likely to be at most 1% of the size of the database [14,15,16], they are still quite large since the database itself is of 10^{10} bytes. Furthermore, since there may be a number of accesses to the information for every access to the database, it must be possible to access them very fast. Therefore, the

structure memory (SM), which is the repository of all structural information, has to provide a large capacity and good access speed. Such a performance can be achieved through the use of emerging technology, such as charge-coupled devices or magnetic bubble memory devices.

The DBC is the first database machine with security mechanisms being incorporated in it at the outset. Generality in security enforcement is allowed through the record-at-a-time post-checking for field-level security in the security filter processor (SFP) and the more efficiently implemented security control for compartmentalizing records of the same security specifications. Post-processing of records and data items constitute some other functions provided by the SFP.

Other components such as the structure memory information processor (SMIP), the index translation unit (IXU) and the keyword transformation unit (KXU) are functionally specialized in the DBC. They are pipelined and multiprocessed by the database command and control processor (DBCCP) for concurrency that enhances the overall performance of the DBC. The DBCCP is therefore charged with the synchronization and control of all the DBC components, so that they can work concurrently on one or more commands. The variable-length commands are sent to the DBCCP by the front-end program execution system (PES). The DBCCP interfaces with the PES by receiving commands and returning appropriate responses, such as sets of records, diagnostic messages, etc. Other functions of the DBCCP include the clustering of records during insertion, pre-processing of the record-level (type A) and field-level (type B) security specifications, coordinating the task of security checking during database accesses, instructing the SFP to post-check the response set for the field-level (type B) control, and performing certain essential bookkeeping chores.

Without belaboring the terminology and details of the various components which will be provided in later sections, let us first gain an overview of the flow of command execution of the DBC. The database stored in the mass memory (MM) is made of records. Every record consists of a record body, a set of attribute-value pairs (known as keywords), and a number representing the record set (known as security atom) of which all the records satisfy the same security specifications. The set of all security atoms makes a logical partition of the database such that all records belonging to an atom are protected in an identical manner with respect to a given user. Since the database resides on many cylinders and one cylinder is searched at a time, keyword indices are maintained in the structure memory (SM). For a keyword K, an entry of the structure memory consists of a list of index terms of the form (f,s), where the cylinder f and security atom s contain records with keyword K.

Given the boolean conjunctions of keyword predicates (known as query conjunctions), as part of an input command, the database command and control processor (DBCCP) considers each query conjunction in turn. For each predicate of the conjunction, the KXU uses the attribute of the predicate and the file name to determine the whereabouts (in the structure memory) of the keywords that satisfy the predicate. The aggregates of all index terms for the keywords satisfying a predicate in a query conjunction are retrieved from the structure memory and transmitted to the structure memory information processor (SMIP). The SMIP, then, intersects the aggregates of index terms. There are as many aggregates as there are predicates in the query conjunction. After the intersection, the resultant set of index terms are further filtered by the DBCCP. The DBCCP deletes all those index terms that have numbers of the atoms to which the user (i.e., the issuer of the query conjunction) does not have the authorized access right. This final set of index terms, together with the complete query conjunction, are now sent to the mass memory for content search. Output from the mass memory may be post-processed by the SFP before routing to the front-end PES.

As depicted in Figure 2, there are two classes of input commands recognized by the DBCCP: access commands and preparatory commands. (For a complete repertoire of DBC commands, see [14].) Access commands are those that require accesses to the

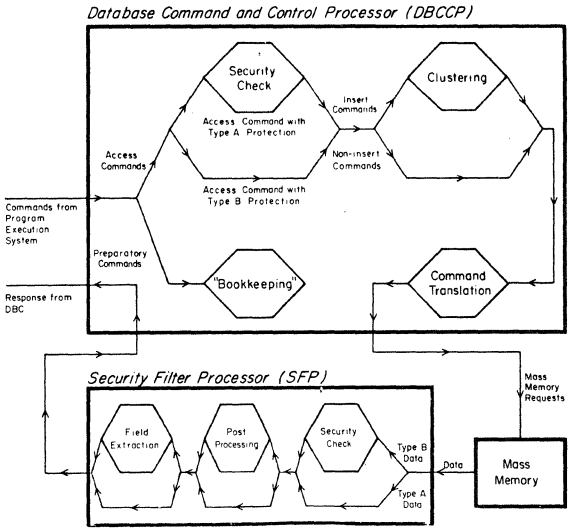


FIGURE 2. Execution of Commands Received from a Front-end Computer

mass memory. Preparatory commands, on the other hand, convey information about the database such as the names and attributes of files to be created, characteristics of the attributes, space requirement of files and security specifications. Each access command is executed in a pipelined fashion by the various components of the DBC. The DBCCP coordinates the operation of the other components and keeps track of the status of the commands that are currently being executed. The information received in the preparatory commands are organized in a random access memory of the DBCCP. This information is referenced frequently during the execution of access commands.

Records to be inserted in the database are physically clustered by the DBCCP according to

their primary and secondary clustering attributes. We will return to Figures 1 and 2 in later sections when we discuss the individual components of the DBC.

Both the clustering and security mechanisms are illustrated by way of an example in the appendix of [9]. In that appendix, the execution of a number of queries through major stages of the DBC is also illustrated. The reader may refer to [9] for a more theoretical discussion on DBC concepts.

3. DESIGN CONSIDERATIONS OF THE ON-LINE MASS MEMORY

The design of the mass memory (MM) is heavily dictated by the storage and processor technologies, database size and processing characteristics. Let us consider each of these factors in the sequel.

3.1 The Use of Moving-Head Disks

A survey of the current and emerging technologies indicates that the various on-line memory technologies may be divided into three major classes, on the basis of their cost and performance. At the higher end of the cost-performance spectrum, there are the magnetic core, MOS and bipolar technologies. In the middle, there are the fixed-head disk technology and its potential replacements, namely, the charge-coupled devices (CCDs), dynamic RAMs, magnetic bubbles and electron beam addressable memories (EBAMs). In terms of low cost per bit and high storage capacity, however, there is no known and emerging technology in sight that can compete with the moving-head disk technology which occupies the lower end of the cost-performance spectrum. Thus, moving head disks seem to be the only alternative for large on-line database store. We have thus chosen moving-head disks for the DBC mass memory.

Once the technology is chosen, we then ask what kind of modifications of the moving-head disk is necessary in order to support database management. The performance gain due to such modifications must be cost-and-performance-effective so that the cost-performance projection of the modified disks will not exceed either the fixed-head disk or its replacements.

Typical database management operations require the processing of 90-95% of related data for the purpose of producing 5-10% of useful information (known as the 90-10 rule). It is desirable that the mass memory should process the related data rapidly so that the results can be obtained without being delayed by the sheer volume of the related data. This calls for high-volume read-out and processing capabilities.

3.2 The Tracks-in-Parallel Read-Out Capability

Conventional moving-head disks, as well as fixed-head disks, allow the read-out of only one track per disk revolution. By modifying the read-out mechanism of moving-head disks, the mass memory can read, instead of one track per disk revolution, all the tracks of a cylinder in the same revolution. This modification is called tracks-in-parallel read-out. Such modification is known, at the time of this writing, to be feasible and relatively low in cost [17], since some of the read/write electronics are already a part of the moving-head disks. Modifications are necessary so that all the read/write heads can be triggered to read simultaneously and that the data buses are enlarged for accommodating the increased data rate.

3.3 The Dynamically Associated Logic-per-Track Approach

With the moving-head disks modified for high-volume read-out, the mass memory must now provide high-volume processing. The mass memory information processor (MMIP) obtains and processes an entire cylinder of information in one disk rotation time. Since the rotation speed of the disks is relatively slow, it is possible to process information 'on the fly'. Processing on the fly is possible because every track of the cylinder is actually processed by a separate processing unit called a track information processor (TIP) having some amount of buffer space. For instance, considering a disk rotation speed of 3,000 revolutions per minute and a track capacity of 30,000 bytes, we require a processing speed (for comparison-type operations) of no more than 1.5 Mbytes per second from each track information processor. This is within the present state of the art of microprocessor technology. Furthermore, if there are forty tracks in a cylinder, then there will be forty TIPs in the MMIP. The MMIP is time-shared among all the cylinders of the mass memory.

3.4 The Content-Addressable Capability

In data management, processing means content-addressable search, retrieval and update. With the mass memory modified for high-volume readout and with the high-performance processors, we now illustrate how the mass memory (MM) performs content-addressing. For this discussion, we must introduce some notions and terminology.

The DBC accepts and stores a database as a collection of records. Each record consists of a record body and a set of variable-length attribute-value pairs, where the attribute may represent the type, quality, or characteristic of the value. The record body is composed of a (possibly empty) string of characters which are ignored by the DBC for search purposes. For logical reasons, all the attributes in a record are required to be distinct. An example of a record is shown below:

(<RELATION,EMP>,<JOB,MGR>,<DEPT,TOY>,<SALARY,15000>)

The record consists of four attribute-value pairs. The value of the attribute JOB, for instance, is MGR. Attribute-value pairs are called, for short, keywords. They obviously characterize records and may be used as 'keys' in a search operation.

The DBC interfaces with the front-end computers by accepting a large repertoire of high-level database management commands [14], by delivering collections of records as response sets, and by indicating successful or unsuccessful execution of the commands in messages. Some of the commands, called record access commands, may be used for specifying a collection of records in the database and for carrying out an intended operation on these records, such as retrieval, deletion and modification. Other commands may be used for database loading, record insertion, initialization, etc.

An important feature of the DBC record access commands is that they allow natural expressions for specifying a record collection. A record

collection may be specified in terms of a keyword predicate, or simply, predicate, which is a triple consisting of an attribute, a relational operator (such as, =, ≠, >, ≥, ≤, <) and a value. For example, the predicate

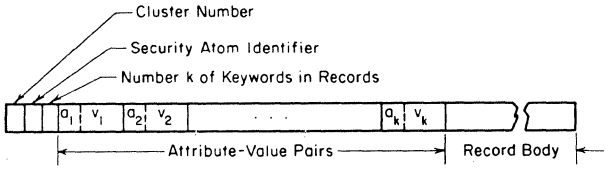
(SALARY > 10000)

may be used to indicate all records that have SALARY as one of the attributes, the value of that attribute being greater than 10,000.

A record collection may also be specified in terms of a conjunction of predicates called the query conjunction. An example of a query conjunction is

(SALARY ≥ 25000) ∧ (JOB ≠ MGR) ∧ (RELATION = EMP).

Carefully planned physical layouts of the record are used in the DBC to eliminate unnecessary disk revolutions and to reduce the cost and size of the TIPs' buffers. Each attribute is first encoded by the DBC, so that it has a unique numerical identifier. The attribute-value pairs (keywords) in a record as shown in Figure 3a are now arranged in an ascending order of the attribute identifiers. The cluster number and the security atom number of a record, seen in the record layout of Figure 3a, will be discussed later in this paper. The layout of a query conjunction is depicted in Figure 3b.

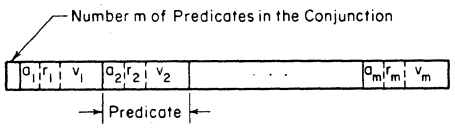


a_i : Fixed-Length Attribute Identifier of the i-th Keyword of the Record

v_i : Variable-Length Value with Length Indicator of the i-th Keyword

$a_1 < a_2 < a_3 < \cdots < a_k$.

(a) The Format of a Record R in the Mass Memory



a_i : Fixed-Length Attribute Identifier of the i-th Predicate of the Conjunction

r_i : Relational Operator of the i-th Predicate

v_i : Variable-Length Value with Length Indicator of the i-th Predicate

$a_1 \le a_2 \le a_3 \le \cdots \le a_m$.

(b) The Format of a Query Conjunction

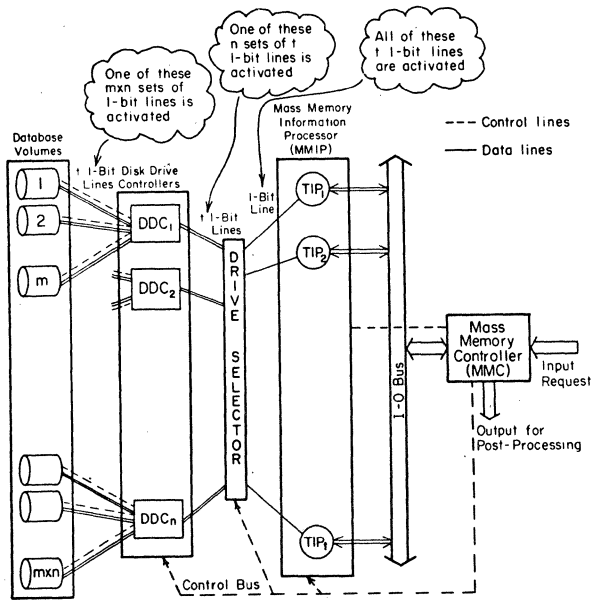
FIGURE 3. Internal Formats of Records and Query Conjunctions

The predicates in a query conjunction, like the keywords in a record, are arranged in an ascending order based on the attribute identifiers. A query conjunction is stored in a sequentially accessed memory. The track information processor (TIP) reads a record from the track as a part of one data stream and the query conjunction from the sequentially-accessed buffer as another data stream and carries out a simple bit-by-bit comparison of the two streams. Whenever there is a match between an attribute identifier in the record and an attribute identifier in the conjunction, the TIP then compares the value parts to determine if the corresponding predicate is satisfied. If the attribute identifier in the record is less than the attribute identifier in the conjunction, then the TIP skips over the corresponding value to the next attribute identifier of the same record. If the attribute identifier in the record is greater than the one in the conjunction, then the TIP skips the entire record. The above logic is repeated until either all predicates in the conjunction are satisfied or the record does not satisfy the conjunction. The scheme just described will result in a simple serial-by-bit comparison.

A conjunction Q, after it is broadcasted by the mass memory controller, is stored in each of the TIPs. All the track information processors (TIPs) simultaneously evaluate the query conjunction against their corresponding incoming record streams. For example, the first TIP searches the records of the first track of the cylinder. At the same time, the i-th TIP searched all the records in the i-th track of the same cylinder. In one disk revolution, all tracks of an entire cylinder are thus searched in parallel by the TIPs.

4. THE OVERALL ORGANIZATION OF THE MASS MEMORY

The overall organization of the mass memory is shown in Figure 4. The database resides in



DDC: Disk Drive Controller
TIP: Track Information Processor
t = # of tracks per cylinder
m = # of disk drives per disk drive controller
n = # of disk drive controllers for the entire database

FIGURE 4. The Mass Memory Organization

data volumes mounted on moving-head disks drives. A volume is composed of 200-400 cylinders. Data transfer to/from a cylinder is achieved by activating all the read/write heads of the access mechanism concurrently.

Although other attempts [3] have taken advantage of the fact that the read and write heads on a track could be positioned a short distance from each other, we do not favor such an arrangement. This is because, at high track densities (1000 tracks per inch or higher), the required mechanical tolerances for sustaining separate read and write heads may well deprive the disk technology of much of the cost-effectiveness brought about by the higher densities [18]. In this design, a combined read/write mechanism is assumed. The implication of such a decision is that a disk device in the mass memory can either be read from or written into at a given time. Reading and writing cannot be performed simultaneously.

The set of disk drives is partitioned into groups of 8-16 drives for access and control purposes. Each group of disk drives is controlled by a disk drive controller (DDC). A drive selector determines at any instant a particular disk drive controller, which, in turn, determines a disk drive from/to which data is being transferred. The drive selector also routes data in parallel to/from all the track information processors (TIPs) that consitute the mass memory information processor (MMIP). Finally, there is a mass memory controller (MMC) to receive requests, broadcast query conjunctions and commands to the track information processors, and control the operations of the mass memory information processor, the drive selector and the disk drive controllers. Data is transferred between the mass memory controller (MMC) and the track information processors (TIPs) via the I-O bus.

Recall that, in this design, a single cylinder is content-addressed at a time. Therefore, assuming that there are t tracks to a disk cylinder, a data transfer path consists of a 1-bit line from each of the t tracks of a cylinder belonging to a particular disk drive, t 1-bit lines from the corresponding disk drive controller and all the t 1-bit lines from the drive selector to the individual track information processors. The above approach provides for a simple design of the disk drive controllers. In fact, t 1-bit registers are all that is needed in each disk drive controller for buffering the data between the drive selector and a selected disk drive.

Although a bit- (or byte-) length buffer in each TIP is sufficient for the evaluation of a query conjunction, a record-length random access buffer is provided in each TIP. This is necessary for performing update as well as for holding on to a record during the query evaluation process. If a record satisfies the qeury conjunction, then it may be transferred to the mass memory controller (MMC). During insertion, a record received from the MMC is held in the record-length buffer before being written into the track. During updates, a record is modified in the buffer only if it satisfies a selection criterion (i.e., query conjunction). The updated record is written back in place during the next revolution, as long as it does not increase in size. If the record does increase in size, then the original record is tagged for deletion in the next revolution. The updated record is then sent from the TIP buffer to the MMC for insertion. (Since record insertion involves clustering, it is dealt with in more detail later in this paper.) We note that if no more than one record from each track of the content-addressable cylinder requires update, the the process is usually completed in two disk revolutions. If some track has more than one record for update, then more revolutions will be required. To approach an update speed of two disk revolutions per cylinder, it may be desirable to increase the buffer size in each TIP, perhaps to a multiple of the record length.

4.1 Two Modes of Operation

The mass memory operates in two basic modes -- the normal mode and the compaction mode. In the normal mode, input requests are decoded by the mass memory controller (MMC) and are queued according to the cylinders referenced by the requests. For each cylinder for which a queue of requests exists, the MMC asks the appropriate disk drive controller (if free) to position the read/write heads to the cylinder. When the cylinder is thus accessed, the MMC sends the requests one at a time to the mass memory information processor (MMIP). While the track information processors (TIPs) of the MMIP are executing the requests, the MMC can ask the disk drive controllers to position the read/write mechanisms to other cylinders for which there are non-empty queues. Thus the access time with respect to a cylinder is at least partly overlapped by useful work performed by the MMIP. The extent of overlap is determined by such factors as the average number of different cylinders for which there are non-empty queues.

Records which are identified by a delete command under the normal mode are tagged by the track information processors (TIPs) for later removal during the compaction mode. Since reading and writing are not done simultaneously, the record deletion process involves two disk revolutions per cylinder. During the first revolution, each TIP creates a bit-map of tag bits (there is a bit position in the bit-map for each record position in a track). The bit-maps are created by the TIPs and inserted in the beginning of the tracks during the second revolution. When the mass memory controller is ordered to reclaim the space occupied by tagged record, it enters the compaction mode. During this mode, cylinders with tagged records (this information being maintained by the mass memory controller using a bit-map, with one bit for each cylinder) are read into the mass memory via the TIPs. The mass memory controller then sends back to the TIPs only the untagged records.

There are two reasons for handling deletions in this manner. First, if reclamation of space were to be attempted in the normal mode, one of two undesirable things will occur: (1) we will have to provide a track-size buffer with each TIP resulting in low utilization of the buffer during retrieval, (2) we will have to reclaim space in segments of the track, each segment size being equal to the size of a TIP buffer. In the

latter case, the number of revolutions required to 'sweep' the entire track for reclamation will be a multiple of the ratio of the track size to the TIP buffer size. During the normal mode of operation, a single delete operation could hold up retrievals for several revolutions. This is undesirable. On the other hand, we might expect during the course of a 24-hour day, periods of light load. Such periods usually result in low utilization of system resources. By operating the mass memory in the compaction mode during these intervals of light load, we may be able to achieve a more equitable distribution of load on the mass memory.

4.2 The Need for Search Space Reduction

Despite all the improvements that can be made to the moving-head disk technology, there is still one fundamental limitation of the technology -- the time delay in repositioning the read/write heads from one content-addressable cylinder to another. This delay is particularly acute if the number of cylinders to be addressed is large. There are two factors which may cause the unnecessary search of a large number of cylinders: (1) the database creator inadvertently scatters his records over a large number of cylinders, thus, requiring the mass memory (MM) to 'sweep' through all those cylinders, (2) for a given query conjunction, the MM does not have any knowledge of those records which may satisfy the query conjunction. If, on the other hand, it knows which cylinders may contain the desired records, then the MM can restrict its content-addressable search to just those cylinders, instead of the entire cylinder space.

4.2.1 The Clustering Mechanism

To eliminate problem (1), the database computer (DBC) provides a clustering mechanism in the database command and control processor (DBCCP). With the clustering mechanism, the DBC allows physical grouping of records, that are likely to be retrieved and updated together, into as few content-addressable cylinders of the MM as possible. The DBC provides two levels of clustering: first, by a primary clustering attribute and second, by a secondary clustering attribute. Clustering attributes are supplied by the front-end system (PES) based on a knowledge of the access pattern. In other words, clustering attributes are chosen on the basis of the frequency of access of various record collections. This choice may be straightforward, as demonstrated in [14,15,16].

The DBC attempts to store all records with the same value for the primary clustering attribute into as few cylinders as possible. Therefore, given a query conjunction inolving a primary clustering attribute, the search space is limited to a very few cylinders, even if there is no further knowledge about the database. For example, in the DBC implementation of a relational database, each record (corresponding to a relational tuple) contains a keyword <RELATION,relation-name> where RELATION is an attribute and relation-name is the relation to which the record (tuple) belongs. If RELATION is declared as a primary clustering attribute, then every single-relation query can be executed by searching adjacently only as few cylinders as are required to store the entire relation.

At the second level of clustering, the secondary clustering attribute provides a further degree of search precision. In fact, since the cylinder size is very large (say, 1/2 megabyte), the two levels of clustering should allow most queries to be executed in only one cylinder access. A more detailed example of the clustering process is included in [9].

4.2.2 The Maintenance of Indices

To address problem (2), the database computer (DBC) maintains some auxiliary information about the database in a separate component known as the structure memory (SM). Indices are maintained in the SM on selected attributes of the records and their value ranges. Clustering attributes are likely candidates for indices, since most queries are expected to refer to these attributes. Furthermore, each query conjunction is recommended to include at least the primary clustering attribute.

An index term for a selected attribute-value (range) pair consists of, among other items, the cylinder number of the cylinder containing at least one record having the selected attribute-value pair. For a query conjunction, it is now feasible to consult the SM for the purpose of obtaining just those cylinder numbers of the index terms whose attribute-value (range) pairs satisfy the query conjunction.

5. DESIGN CONSIDERATIONS OF THE STRUCTURE MEMORY

The structure memory (SM) is the repository of auxiliary information about database. This information is concerned with search precision and access control. For improving search precision, the SM is employed by the database computer (DBC) to determine the mass memory cylinders that need be content-addressed. For access control, the SM is again used by the DBC to determine whether an access operation is an authorized one and whether access is permitted to the records involved. The use of cylinder numbers as a part of the index term for search precision has been discussed in the previous section. In the following section, we will concentrate on the discussion of the access control feature of the SM.

5.1 Pre- and Post-Checking for Access Control

The DBC provides two types of access control. Access requests with the type B control are slower to execute, because such requests require post-checking of every retrieved record for field-level security clearance. This type of security enforcement is performed by a special processor known as the security filter processor (SFP) which also does some other post-processing of records retrieved from the mass memory (see Section 7). Further, these requests may result in access imprecision since some of the retrieved records may have to be discarded by the SFP due to security violation. The type A control, on the other hand, requires no post-checking of records. It works solely on the basis of the access control-related information stored in the structure memory (SM) and in the database command and control processor (DBCCP). During database creation time, the access-control related information are extracted from the new records and stored in the

structure memory. The effect is that of pre-checking of records. Thus, at query execution time, security clearance may be made even before records are actually retrieved from the mass memory. Since the type A control incurs no access imprecision, it should be used regularly. However, to use type A control, the database creator must understand the notion of security atoms and be willing to designate certain keywords of his records as security keywords. With the security atoms and keywords, the DBC can then construct access control-related information and place the information in the SM and DBCCP for subsequent use.

5.2 The Notion of Security Atom

A security keyword of a record is a keyword of the record which is designated by the database creator to reflect his security requirements. All records having the same canonical expression of security keywords form a record set called a security atom. The advantageous properties of the security atom [21] are as follows:

(1) Security atoms represent disjoint record sets, i.e., a record belongs to one and only one security atom.
(2) The database can be partitioned into security atoms, with all records in an atom having the same security attributes.
(3) With proper choice of security attributes, the partitioning (i.e., the sizes of security atoms) can be made from very fine to very coarse, depending on the security requirements.
(4) Usually the total number of security atoms in the database is much smaller than the total number of records in the database.
(5) For any arbitrary query conjunction made up of security keywords, the records of a security atom will have the following exclusive property: Either all or none of the records of the security atom will satisfy the query conjunction.

For this type of access control, a user of the database is always provided with a database capability. Each element of the capability consists of a query conjunction (made up of security keywords) and a set of access rights. A security atom expression may satisfy a number of query conjunctions in the database capability. The access rights on a security atom for the user is therefore the intersection of the sets of access rights corresponding to the query conjunctions that are satisfied by the atom expression. Consequently, for each user, a list can be created indicating the access rights on each security atom. This list is called the atomic access privilege list of the user. Using this list, the database computer can now process a user request by first determining whether there is any atom expression that satisfies the request. If there is such an expression, then the access requested by the user is compared with the access rights assigned to the atom. If the requested access is an authorized one, then access to the atom (i.e., record set) is permitted. Subsequently, the record set is accessed by the mass memory (MM). A detailed illustration of the security atom concept for access control is included in [9].

5.3 The Structure Information

For every keyword designated for indexing, there is an entry in the structure memory (SM) consisting of the keyword itself and a list of index terms. An index term is composed of a cylinder number f and a security atom number s. An index term (f,s) for a keyword K, therefore, indicates that there exists one or more records containing the keyword K that are residing in the cylinder f of the mass memory (MM), and that are belonging to the security atom s.

For type A control, the query conjunction of a user is processed as follows: For each predicate with an indexed attribute, the structure memory (SM) determines all those keywords which satisfy the predicate. Corresponding to each of the satisfying keywords, a set of index terms is retrieved. The sets of index terms for all such predicates are then intersected (by the structure memory information processor to be discussed in Section 7). The result of the intersection is a list L of index terms for the given query conjunction.

This list L of index terms is compared (by the database command and control processor (DBCCP)) against the user's atomic access privilege list to determine the final list L'. The list L' includes only those (f,s)-pairs of L where the required access is permitted on the security atom s. The list L' together with the query conjunction and the requested access are now forwarded to the mass memory (MM).

As we stated earlier, the mass memory stores a record as variable-length attribute-value pairs, together with a record body. For the purpose of identifying the security atom to which it belongs, each record is also tagged with the security atom number as depicted earlier in Figure 3a. Given a query conjunction Q and a list L' of index terms (f,s), the mass memory can then narrow its content-addressable search to those cylinders whose numbers appear in L'. For each unique cylinder number f in L', the mass memory will access cylinder f, disregard those records that are not tagged with one of the corresponding security atom numbers s, and output only those that satisfy the conjunction.

5.4 The Performance Requirement and Choice of Technology

Typically, indices for conventional databases range from 1% to 10% of the size of the database [22]. In the DBC, the database needs to be indexed to the level of cylinders (instead of, tracks, pages and offsets within pages as in conventional systems). The total number of index terms for the database is therefore smaller. In fact, the size of the indices in the SM should not exceed 1% of the size of the database. This has been verified for realistic applications on the DBC [14,15,16]. Therefore, the capacity requirement of the SM for a 10^{10}-byte database is at most 10^{8} bytes.

Another important feature required in the SM is that it should provide sufficient search and retrieval speed, so that query conjunctions may

be processed at a rate commensurate with that of the mass memory. While the mass memory is working on the current request, the structure memory can work on the next request. Normally, a query conjunction contains no more than two predicates of index attributes, as seen in [14,15,16]. If each of these predicates is satisfied by 5 to 10 keywords, then at most 10 or 20 sets of index terms need be referenced per query conjunction. Consequently, for accessing a set of index terms, the structure memory requires a speed of 1 to 2 milliseconds, since all the 10 or 20 sets of index terms must be accessed in 20 milliseconds, which corresponds to the time required for one disk revolution.

The above performance requirement can be met at a relatively low cost by using one of the emerging technologies such as the bubble memories and charge-coupled devices (CCDs). According to a recent survey [20], CCDs can access a random block in 100 μsec and their costs are projected to be 50 millicents per bit. Bubble memories can also access random blocks, but in 1 msec and their costs are coming down to 10 or 20 millicents per bit. At the system level, the cost of CCD memories is about 250 millicents per bit while the cost of bubble memories may be 30 to 50 millicents per bit. Since the block-oriented bubble memories provide the required access speed at a lower cost than CCDs, they are a very good choice for the structure memory technology. Electron beam addressable memories (EBAMs) have also been studied in [10] for their applicability in structure memory design. Although such memories are expected to provide the lowest cost per bit (about 10 to 20 millicents per bit), the reliability of these memories is still uncertain. Furthermore, to absorb the high cost of their complex circuitry, EBAMs are cost-effective only for very large memories. In our implementation of the DBC, either bubble memories or CCDs is the present choice for the structure memory design.

6. THE OVERALL ORGANIZATION OF THE STRUCTURE MEMORY

From our discussion in Section 5.4, it is apparent that the structure memory should provide for a high search speed at a low cost. With the total size of the structure memory being of the order of 100M bytes, the speed requirement implies that the memory must be content-addressable and that the content-search operation should be carried out by multiple processing elements. The structure memory may, therefore, be split up into a number of sections (later called memory units) and each section may be assigned to a separate processor.

The structure memory is made up of a segmented sequential memory (e.g., CCDs or bubbles). Hence, any search on such a memory can be carried out no sooner than the data transfer time of a single physical segment. The larger the number of segments to be serially searched, the longer will be the total search time. It is, therefore, reasonable to try and assign a separate processor to each physical segment. Unfortunately, a segment is normally quite small, say up to 2K bytes, while the entire structure memory size is up to 100M bytes. Consequently, the above assignment would call for an extremely large number of processing elements. On the other hand, it would be cost effective (1) to utilize a small number of processors, (2) to assign a number of segments (later called memory modules) to each processor and (3) to provide a mechanism to identify a single segment (if possible) for search by each processor in response to an index search request. The structure memory organization presented below adheres to these guidelines.

The structure memory is organized as an array of memory unit-processor pairs which are managed by a controller. A memory unit, in turn, is composed of a set of memory modules. All memory modules are of the same fixed size. A processor can address any memory module within its memory unit, and then content-address the entire module. Furthermore, the structure memory controller can trigger all the processors to content-address their corresponding modules simultaneously.

6.1 The Notion of Bucket and Parallel Array of Memory Unit-Processor Pairs

Whenever possible, searching of the structure memory on the basis of a given keyword should be restricted to at most one module from each memory unit. To achieve this goal, all keywords and their index terms corresponding to a particular attribute (and lying within a given value range) will constitute a bucket. Each bucket is physically distributed among the various memory units in order that it may be searched in parallel by all the processors. Ideally a bucket is placed in n modules, one from each of the n different memory units.

Unfortunately, buckets are not necessarily equal in size. Therefore, a mechanism needs to be provided for dynamically varying the amount of physical space that is to be assigned to each bucket. The above structure memory organization with small module size allows for such variability of bucket size. Each bucket may be placed in one or more modules (as many as necessary) evenly distributed among different memory units. The concept is illustrated in Figure 5, where the 'shaded' modules contain a single bucket.

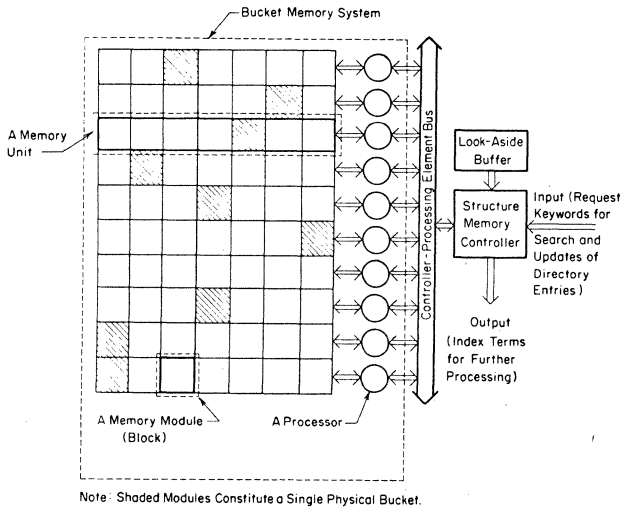


FIGURE 5. Organization of the Structure Memory

The bucket to which a keyword and its index terms belong is determined by a separate component of the DBC, called the keyword transformation unit (KXU) which we will discuss in Section 7. One of the functions of the structure memory is to map a bucket name into the memory modules allocated to the bucket. For this purpose, the controller has a small random access memory in which it records a bucket name and stores the corresponding module numbers. Thus, given a bucket name, all the processors can work simultaneously on the modules which contain the bucket.

6.2 The Use of Emerging Technologies

The processors of the bucket memory system must be sufficiently fast so that the data in each memory module can be processed on the fly. Shift register memories, made of bubble memories or CCDs, commonly have a module size of 2K bytes. For an access time of 1 msec, each processor must, therefore, be able to process data (with comparison-type operations) at the rate of 0.5 μsec per byte. This speed should be easily achievable with realtively powerful microprocessors (or a few of them working in parallel as a single processing element). If module size is larger, then data may be processed in a buffered mode, with each processing element having a random access store equal in size to a module of the bucket memory.

6.3 The Look-Aside Buffer

The look-aside buffer is used for enhancing the performance of the structure memory. During normal operations of the database, the retrieval of information from the structure memory is likely to be more frequent than the update of information in the structure memory, especially because update operations are also preceded by search and retrieval. However, it is conceivable that during short intervals of time, a large number of updates may have to be carried out. Such an event may adversely affect the average retrieval rate. The use of a look-aside buffer, implemented with fast random-access memory, is aimed at alleviating such a degradation in structure memory performance.

When an update request is received by the structure memory, it is temporarily placed in the look-aside buffer. The information in the bucket memory is not immediately updated. The contents of the look-aside buffer, therefore, represent pending updates which are yet to be permanently recorded in the bucket memory system. Execution of the requests in the buffer is delayed until either of the following two conditions occurs: (1) the loading of the buffer reaches a certain threshold value; (2) the structure memory encounters a slack period with no new requests awaiting execution.

Execution of a retrieval request, then, is carried out in the following manner. Given a keyword K, the processors are simultaneously activated to determine the set of index terms of K stored in the bucket memory. The structure memory controller then adds to this set, if necessary, extra index terms as a consequence of the insert requests stored in the look-aside buffer that affect K. Similarly, delete requests stored in the look-aside buffer may cause the deletion of some index terms from the final set of index terms prepared for output.

In summary, the complete structure memory organization is also shown in Figure 5. It consists of a bucket memory system, a structure memory controller and a look-aside buffer. Input requests are received by the structure memory controller either in the form of keywords for subsequent search for their index terms, or in the form of keyword-index term pairs for intended update. Output from the structure memory consists of one or more sets of index terms for further processing. Thus, the responsibility of the structure memory controller consists of maintaining the bucket-to-module maps, controlling the bucket memory system, maintaining the look-aside buffer, taking input requests from the data-base command and control processor (DBCCP) and transferring index terms to another DBC component, namely, the structure memory information processor (SMIP) (to be discussed in Section 7). In response to requests for keyword search, the structure memory controller activates the processors and then broadcasts the keyword to them for the required content-search of index terms.

7. THE FIVE OTHER COMPONENTS OF THE DATABASE COMPUTER

We have so far discussed the organization of the mass memory (MM), and the structure memory (SM). But from time to time we have made reference to the fact that some other components are also necessary. In particular, we have referred to the database command and control processor (DBCCP), the security filter processor (SFP), the keyword transformation unit (KXU), the structure memory information processor (SMIP) and the index translation unit (IXU). In referring to Figure 1, we note that the structure loop, which consists of the KXU, SM, SMIP, IXU, and DBCCP, is used for limiting the mass memory search space, for determining the security atoms allowed for accesses with the type A control, and for clustering records received for insertion into the database.

7.1 The Keyword Transformation Unit

The keyword transformation unit (KXU) allows the structure memory first to readily identify the modules which contain the index terms of the keywords by providing the associated bucket name, and then to process index terms and keywords rapidly since KXU transforms all information to be stored in the structure memory into fixed-length fields.

Each attribute in the database has a unique identifier. Information about the various attributes, supplied by the program execution system (PES), is sotred in a table of the KXU, called the attribute information table. It includes for each attribute the minimum and maximum values, the type of these values (numeric, floating point, alpha-numeric, etc.) and the number of ranges into which these values may be divided. For different attributes, different hash algorithms may be used to hash the variable-length values into fixed-length codes. These hash algorithms constitute a hash algorithm library. We observe that in the above process, a keyword, which is a variable-length

attribute-value pair, is transformed into a fixed-length triple (a,r,v) where a is the attribute identifier, r is the range number in which the value belongs, and v is the hash code of the value. The pair (a,r) is the bucket name of the keyword. Due to hashing, the structure memory may not be able to distinguish between values of two keywords whose attribute and range number are identical. However, this will not result in the retrieval of unnecessary records by the mass memory since the values of the keywords are used and stored in the mass memory in their complete variable-length form.

The organization of the KXU is shown in Figure 6. It consists of a quasi-random access

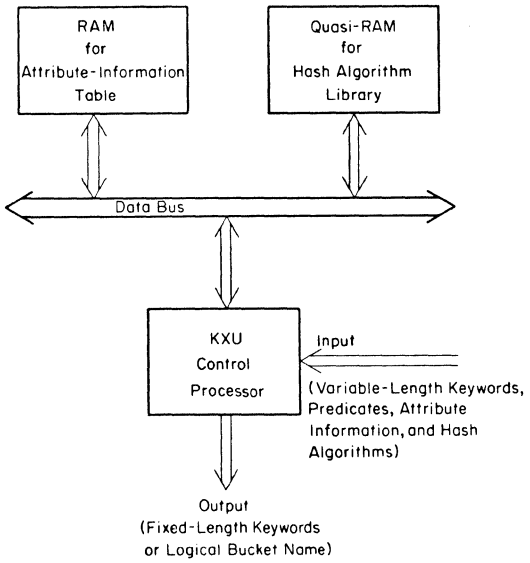


FIGURE 6. Organization of the Keyword Transformation Unit (KXU)

memory for storing the hash algorithm library; a random access memory for storing the attribute information table; and the KXU control processor for performing keyword transformation and for interfacing with the database command and control processor and structure memory. An LSI bit-slice microprocessor may be sufficient for the arithmetic capabilities required in the KXU control processor.

7.2 The Structure Memory Information Processor

The structure memory ifnormation processor (SMIP) performs intersection on the sets of index terms delivered by the structure memory. For an understanding of the operation of the SMIP, let us consider a query conjunction Q,

$$Q = P_1 \wedge P_2 \wedge \ldots \wedge P_n,$$

where each P_i is a predicate. The database command and control processor (DBCCP) makes use of the structure memory and the SMIP to determine the set of index terms to be sent to the mass memory. After the SMIP memory is cleared, the first set of index terms for keywords satisfying P_1, called the argument set of P_1, is provided by the structure memory and then stored in the SMIP memory. Each of the stored index terms is initially associated with a count of one, indicating the number of predicates it has satisfied.

Next, the argument set of P_2 is provided by the structure memory and sent to the SMIP. The associated count of an existing index term in the SMIP memory is incremented by one if the index term matches an index term of the argument set of P_2. The process for P_2 is repeated for each of the other predicates. At the end of this entire process, the stored index terms, those whose counts are n, represent a refined list applicable to the evaluation of Q. This list of index terms is then retrieved by the SMIP and forwarded to the database command and control processor (DBCCP). Subsequently, the list is checked by the DBCCP for security clearance, before being transmitted to the mass memory.

The most important part of the above procedure is the determination of whether an index term already exists in the SMIP memory. To perform this task rapidly, the SMIP is implemented as a set of MU-PE pairs, where MU is a memory unit and PE is a processing element. Since the total number of index terms stored in the SMIP memory is small (in fact, this number is never more than the largest number of index terms of a single attribute), the memory units (MUs) forming the SMIP memory can be made from fast random access memory. A 'double hashing' method may now be applied for the set intersection operation. An index term (f,s), may be treated as a single key and hashed into a number between 1 and m, where m is the number of MU-PE pairs. The index term is thus assigned to a MU-PE pair. Having received the first argument set (that of P_1), the SMIP controller hashes each index term of this set and thereby assigns it to an MU-PE pair. After receiving an index term of the argument set of P_1, each PE uses a second hashing algorithm to determine the address in its MU where the index term is to be stored together with an associated count of one. Thus, the first argument set is distributed among the m memory units. Index terms that hash to the same address in an MU are chained together within the MU itself. In case an MU runs out of space, then a chain can be extended into a less-filled MU.

The ith argument set (namely, that of P_i, for $i > 1$) is treated as follows. Each index term of this set is hashed (using the first algorithm) by the SMIP controller and given to the PE to which the term is hashed. All the PEs can be working in parallel, yet searching for different index terms (in contrast to the structure memory, where all the processors search for the same keyword). After receiving an index term, each PE applies on it the second hashing algorithm to determine the address in its MU which starts a chain of stored index terms. If the given index term is found in this chain and its associated count is (i - 1), then the count is incremented by one; otherwise, no action is taken.

When all the argument sets have been processed in this fashion, the stored index terms, having an associated count of n, are output for further processing. The hardware organization of the SMIP is shown in Figure 7. Each memory unit is a single

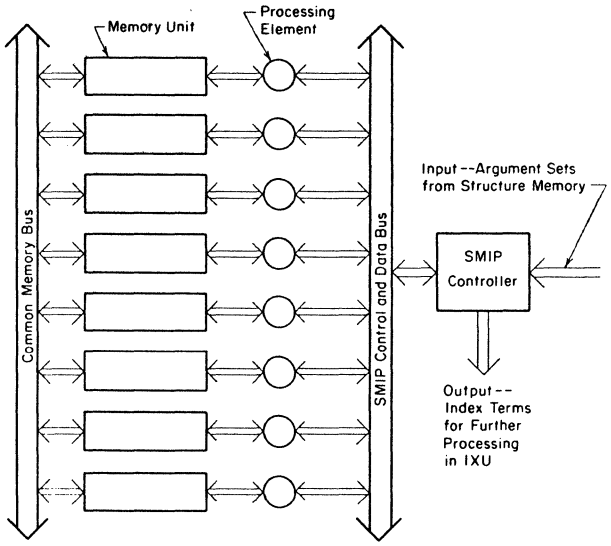


FIGURE 7. Organization of the Structure Memory Information Processor (SMIP)

module of random access memory. The processing elements are made of microprocessors and are capable of doing comparison-type operations as well as executing the second hashing algorithm. The SMIP controller must be quite fast since it executes the first hashing algorithm on all the index terms of the argument sets. However, a very simple but effective algorithm [10] may be used for this purpose, so that the SMIP controller can process index terms at the same rate as it receives them. The common memory bus is used for data transfer when an MU overflows and requires space within another MU.

7.3 The Index Translation Unit

The index terms stored in the structure memory (SM) and manipulated by the structure memory information processor (SMIP) are actually represented in an intermediate form. The purpose of the index translation unit (IXU) is to translate them into a usable form for the mass memory (MM). The other function of the IXU is the assignment and release of cluster identifiers and security atom names, on demand from the database command and control processor (DBCCP).

The DBC allows different users to create files of the database. A user may create one or more files. The creator of a file determines the attributes of the file, the clustering needs and the access rights of the users of the file. The use of files with different primary clustering attributes allows the database computer (DBC) to support different types of data structures such as hierarchical, relational and network data models. Furthermore, it may allow different security provisions to be assigned at the file-level of the database.

There can be a large saving of storage in the structure memory (SM) and in the structure memory information processor (SMIP) if the index terms are reduced in size. This is possible since files are allowed to occupy only disjoint sets of mass memory cylinders. In this case, for an index term of a keyword of some file, instead of storing the absolute cylinder number, only a relative number is stored with respect to other cylinders occupied by the same file. However, since these relative numbers have to be converted into absolute cylinder numbers before being passed on to the mass memory (MM), a cylinder address table is maintained by the IXU for every file of the database.

For an estimate of the type of storage savings that may be achieved, consider a large database with 40,000 cylinders. An absolute cylinder number then, is 16 bits long. If a file is limited to at most 256 cylinders, then only 8 bits are sufficient for a relative cylinder number. Therefore, a 50% saving can be achieved in storing cylinder numbers in the structure memory (SM) and structure memory information processor (SMIP).

In addition to cylinder address tables, the IXU also maintains a cluster identifier bit map and a security atom name bit map. These bit maps are used to keep track of the allocation and release of cluster identifiers and security atom names.

Index terms from the structure memory information processor (SMIP) are received in a burst mode and stored in a buffer made of sequential access memory. These index terms are expanded by the IXU control processor, one at a time, by making use of the cylinder address table. The expanded index terms are sent to the database command and control processor (DBCCP). The IXU also receives requests from the DBCCP for allocation and release of cluster identifiers and security atom names. The bit maps are used for answering such requests. The size of the bit maps and cylinder address table of each file is estimated to be less than 1K bytes. Hence, a small random access memory is used for storing these information about the 'current' file. However, because there may be hundreds of files in the database, the information about the aggregate of all files is stored in a bulk memory.

7.4 The Security Filter Processor

The major function of the security filter processor (SFP) is to enforce the field-level (i.e., type B) security of the database. After the records have been retrieved from the database by the mass memory (MM) in response to a user query conjunction, they are individually checked for security clearance. The SFP is capable of extracting (removing) specified attribute-value pairs from the retrieved records and sends only (none of) these keywords to the database command and control processor (DBCCP).

It might appear that, unlike record retrieval requests, record update and record deletion requests may cause difficult problems if they are to be checked for type B security. This misconception is based on the notion that, once in a while, original copies of deleted or modified

records whould have to be restored, if they have violated the type B security. However, such a problem never appears in the DBC. We recall that record update and record deletion take place in two distinct steps. Both of these operations require that records be first selected on the basis of a given criterion (query conjunction). This is the selection phase (or read phase). The retrieved records are post-checked for type B clearance, and only those that are cleared may now be modified or deleted from the database. This completes the write phase and signals the end of the update or deletion operation. In other words, no deletion or modification of the original database takes place prior to the post-checking for the type B clearance. If there is an overwhelmingly large number of records to be updated by the SFP due to field-level security control and processing, the mass memory (MM) may neither send the next 'batch' of records, if any, to the SFP, nor write newly modified records back to MM, since the SFP is still busy. In this case, the MM misses a disk revolution and attempts to either send the retrieved records or write the modified records in the next revolution. There is no outstanding problem. The lesson to learn is that large amounts of updates due to the type B control will take longer time. However, typical updates follow the 90-10 rule (see Section 3.1), i.e., only 5-10% of data requires to be written back to the MM. Therefore, contention for the MM is typically not present.

The organization of the SFP is shown in Figure 8. Input to the SFP consists of records

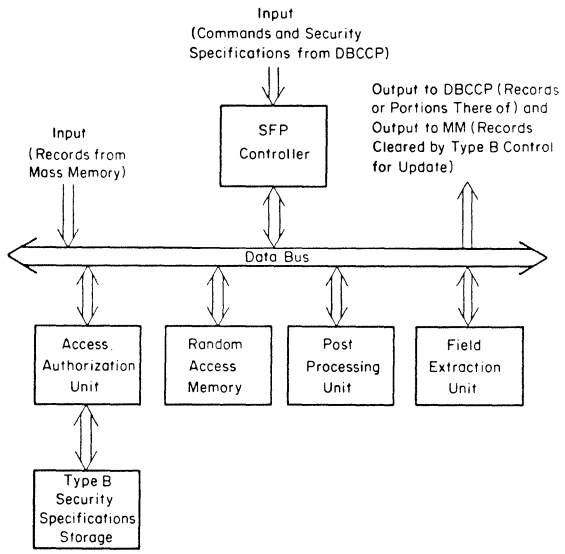


FIGURE 8. Organization of the Security Filter Processor (SFP)

retrieved from the mass memory (MM), and commands the type B security specification from the database command and control processor (DBCCP). Input records that form the response set of a query conjunction are stored by the SFP in a random access memory and, thus, are accessed by all the processing units of the SFP. The type B security specifications are stored in a quasi-random access storage. Whenever needed, the specifications are stored in a quasi-random access storage. Whenever needed, the specifications for a user are loaded by the access-authorization unit for the type B security checking. Records that do not qualify for access are deleted from the random access memory. The post-processing unit performs set function (such as maximum and average) on the response set of a query conjunction. The records in their entirety or certain portions of the records, extracted by the field-extraction unit, are sent back to the database command and control processor (DBCCP). Each of the three processing units is implemented as pairs of circulating memory and processing element. Thus, each of these units can carry out fast comparison-type operations simultaneously on a number of records, thereby providing rapid response to the user request.

7.5 The Database Command and Control Processor

The database command and control processor (DBCCP) provides the control of the entire system as discussed in Section 2 in referring to Figures 1 and 2. In addition, the DBCCP performs clustering. Records to be inserted in the database are physically clustered by the DBCCP according to their primary and secondary clustering attributes. In doing the clustering, the DBCCP maintains a cylinder space table, indicating the space available in each mass memory cylinder, and a cluster information table, showing, first, the definition of each cluster in terms of the keywords with primary and secondary clustering attributes and, secondly, the numbers of the cylinders currently occupied by the cluster. These tables, together with the PES-supplied estimates on the space requirement of the files, support the clustering mechanism of the DBCCP.

Whenever a record is to be inserted in the database, its cluster number is first determined by reference to the cluster information table. The corresponding cylinder numbers found in this table represent candidate cylinders in which the new record may be inserted. The space vacancy of the candidate cylinders is reflected in the cylinder space table. Once a cylinder is determined, it is accessed by the mass memory. The detailed space availability data of each track is found in the header information of the track. The DBCCP then selects a track with the maximum amount of available space. The header for that track is updated and the new record is stored in the track. A detailed algorithm for cylinder selection is presented in [9].

For the type A access control, the DBCCP performs the following: For each query conjunction in an access command, a set of index terms are received from the structure memory via the structrure memory information processor (SMIP) and the index translation unit (IXU) in a pipelined fashion. These index terms carry information on the security atoms to which the records satisfying the query conjunction may belong. Accordingly, only those idnex terms are sent to the mass memory whose atoms are authorized for access. The DBCCP checks the access authorization by using

atomic access privilege lists which show, for every user, the access rights on each atom of a file. Such a list is prepared by the DBCCP on a one-time basis for every user of a file. Finally, the mass memory does its share in security checking by accessing the records that not only satisfy the given query conjunction, but are also tagged with the numbers of the atoms authorized for access. For the type B security, checks on any access are done solely by the security filter processor (SFP). In performing this operation, the SFP makes use of the security specifications supplied on a one-time basis by the DBCCP.

In Figure 2, we have sketched the path in the DBC data loop through which commands and data flow. Access commands are security-checked in the DBCCP unless they have the type B security requirement. Insert commands result in activating the record clustering mechanism of the DBCCP.

The DBCCP can be implemented on a moderately powerful minicomputer with sufficient random access memory to store the information on the characteristics of only the active files and active users. Other information may be stored in a conventional disk. The minicomputer should preferably be microprogrammable, so that the various functions of the DBCCP may be directly implemented in firmware.

Although it is in charge of a number of different tasks, the DBCCP performs only a limited number of tasks during the processing of a single command. If a command, on the average, requires access to one or two content-addressable cylinders in the mass memory, then the DBCCP should be able to handle a command within the time it takes for one or two disk revoluations (i.e., 20-40 msec). By using a minicomputer and implementing the various tasks in firmware, it is anticipated that the DBCCP will be able to cope with the above performance requirement.

8. CONCLUDING REMARKS

Since a large number of common database management functions are implemented in hardware, the DBC is expected to perform appreciably better than the computers that provide these functions by software means. High cost of and long delay in software security enforcement may also be absorbed by the hardware. In addition, it should be performance-and-cost-effective to support very large databases in an on-line and interactive mode, since the DBC's database is stored in relatively low-cost and simply modified moving-head disks. The mass memory information processor (MMIP), if need arises, may be expanded to simultaneously handle disk cylinders each of which is from a separate disk drive. In this expansion, it is only necessary that the number of track information processors (TIPs) in the MMIP be increased accordingly, i.e., one set of TIPs for each drive. Although the mass memory is expanded into even larger content-addressable blocks (each block being made up of several cylinders), the need for a structure memory is still there, since no two blocks may be accessed concurrently. However, as the size of these blocks grows, the need for clustering and the amount of indexing decrease. Thus, the structure memory may decrease in size. Another benefit may occur if there are a multiplicity of MMIPs where each MMIP handles a separate query conjunction, thereby allowing user queries to be multiprocessed.

8.1 A Raw Estimate of the Hardware Performance*

A rather gross first-order analysis of the DBC hardware may proceed as follows: The mass memory logic is designed to process an entire cylinder in one revolution. Because a cylinder generally consists of between 20 and 40 tracks, and because conventional disk systems process one track at a time, we can expect a performance improvement factor of between 20 and 40 over conventional disk systems. Furthermore, since the structure loop can be processing a current request while the mass memory is processing a previous one, a performance improvement factor of 2 can be expected over conventional systems which process or store both the indices and database at the same time or on the same storage medium. In addition, the high degree of pipelining of the DBC components and the clear delineation of front-end general-purpose processing from back-end special-purpose database management may allow a performance improvement factor of 2. Thus, the DBC is likely to have a hardware processing power which is (20, or 40, x 2 x 2 =) 80 to 160 times that of conventional software-based systems.

8.2 Hardware Performance and Limitations

Several simulation experiments [21] have been carried out to determine the response times to query conjunctions, and possible bottlenecks in the DBC hardware. In the simulation study, record retrieval requests to the DBC were assumed to represent 50% of all requests. Since the DBC is designed primarily to respond to the retrieval requests rapdily and the update requests adequately, this low retrieval percentage was expected to be a worst-case performance measure. Retrieval requests as well as update requests may require the use of query conjunctions. A job in the simulation model consists of a single query conjunction and its associated access operation.

A request is processed, first, in the structure loop of the DBC, and then in the data loop. When a job, i.e., a query conjunction, is scheduled for processing by the structure loop, its predicates are first translated by the keyword transformation unit (KXU), index terms for keywords satisfying the predicates are then retrieved from the structure memory (SM) and intersected in the structure memory information processor (SMIP), and finally, the resulting index terms are translated by the index translation unit (IXU). In the data loop, a job is associated with a cylinder number.

The results of the simulation are as follows. Assuming that the SMIP and the IXU can match the processing speed of the structure memory (SM) and that the KXU provides a fixed processing delay,

---

*This estimate was suggested to us by Gordon Bell during a presentation of DBC architecture at DEC by one of the authors.

the response time to requests in the structure loop increases rather rapidly as the access time of the structure memory increases. For instance, for a KXU processing delay of 1 msec, the response time is about 35 msec when the structure memory access time is 1 msec. The response time increases to about 120 msec when the structure memory access time is 2 msec and KXU delay is 1 msec. The structure memory reaches 90% or greater utilization with a 2 msec access time. The response times given above are measured for requests that are composed of 50% retrieval requests with query conjunctions being made up of an average of 4 predicates of indexed attributes. The response time is improved by 10 to 20% when a look-aside buffer is used.

The data loop is slightly slower because of the assumption that the disk revolution time is 20 msec and a processing time of 15 msec is required by the security filter processor (SFP). Jobs arriving at the mass memory may be placed in one of several queues based on the cylinder to be accessed. Good performance can be achieved by executing in sequence all those jobs that are queued up to the same cylinder. In general, the wait time of jobs improves rapidly until the number of queues reaches 4 or 5, and there is very little improvement beyond that point. Due to a limited buffer space in the track information processors (TIPs) and a limited capacity of the bus carrying information from the TIPs to the mass memory controller and beyond, it is not always possible to execute a job in one disk revolution time even if it refers to a single cylinder. However, jobs requiring the read-out of complete cylinders are very rare. Therefore, the average number of disk revolutions per job (i.e., query conjunction) remains very close to 1.

8.3 Performance Evaluation of the DBC in Supporting the Existing Applications

We have also investigated the manner in which the DBC supports hierarchical [12], CODASYL [13], and relational [5] databases. An existing database may be supported on the DBC by converting the database to conform to the DBC representation of data. This one-time conversion is known as database transformation. We do not require the user to reprogram his database management applications. Instead, we provide an interface which in real-time translates the database management calls issued by the application programs into DBC commands. Because DBC commands constitute a high-level data language which closely resembles many high-level data languages and calls of contemporary systems, the translation is straightforward and the interface requires minimal software. Such a process is known as query translation. Both the tasks of database transformation and query translation are charged to a software package called the DBC interface which resides in the front-end computer system. Thus, the interface, together with the database computer, replaces a full-scale software database management system and its conventional disk storage. However, it does not replace the application programs written for the database and run in the general-purpose front-end computers.

It has been estimated [14,15,16] that in supporting these applications on the DBC, the database transformation may result in a database storage requirement as much as 1.5 or 2 times that in a conventional system. This excess storage requirement, however, is adequately offset by one or more orders of magnitude improvement in the execution time or user transactions. Furthermore, the storage requirement for the indices decreases by one or more orders of magnitude. Finally, the size of the software (i.e., the DBC interface) is expected to be several orders of magnitude smaller than conventional database management software.

8.4 Future Work

Certain important problems such as recovery from failure, concurrency control and integrity validation are currently being delegated to software in the front-end system. Future research is anticipated, therefore, in improving the DBC to provide some hardware solutions to the aforementioned problems and relieve the front-end system further from much of database software. We would also like to investigate more thoroughly the performance bottlenecks of the DBC, in particular, the mass memory, the database command and control processor and the security filter processor due to their complexity in design and elaborate usage. The anticipated security cost in utilizing both types A and B will be studied. Preliminary analysis of DBC performance and capability, however, tends to indicate that the DBC may indeed perform very well in realizing the conventional database management applications. This leads us to believe that database machines in general and the DBC in particular may become viable special-purpose computers for very large database management.

ACKNOWLEDGEMENT

The work reproted here is the result of a research initiated by David K. Hsiao, contributed first by Richard I. Baum, expanded by Krishnamurthi Kannan, and continued by Jayanta Banerjee under the supervision of David K. Hsiao. The entire work is conducted at The Ohio State University and supported by the Office of Naval Research through contract N00014-75-C0573.

The authors would like to thank Richard I. Baum for his contributions to the database computer project. Portions of this paper are derived from project reports available either through NTIS under AD-A034154, AD-A035178 and AD-A036217, or from The Ohio State University under OSU-CISRC-TR-76-1, OSU-CISRC-TR-76-2 and OSU-CISRC-TR-76-3. These reports were issued in September, October, and December of 1976, respectively, and co-authored by either Richard Baum, David K. Hsiao and Krishnamurthi Kannan, or David K. Hsiao and Krishnamurthi Kannan.

Thanks are due to the referees' for comments and suggestions which have helped in shortening of the original draft. Thanks are also due to Glen Langdon who, as the Guest Editor of this issue, has handled the manuscript with care and thoughtfulness, and David WanHua Hsiao who typed the manuscript over the weekend. Authors of this paper are listed alphabetically.

REFERENCES

[1] Hsiao, D. K. and Madnick, S. E., "Database Machine Architecture in the Context of Information Technology Evalution," Proceedings of the Third International Conference on Very Large Data Bases, ACM, New York, 1977, pp. 63-84.

[2] Baum, R. I. and Hsiao, D. K., "Database Computers -- A Step Toward Data Utilities," IEEE Transactions on Computers, Vol. C-25, No. 12, December 1976, pp. 1254-1259.

[3] Su, S. Y. W. and Lipovski, G. J., "CASSM: A Cellular System for Very Large Data Bases," Proceedings of the First International Conference on Very Large Data Bases, ACM, New York, September 1975, pp. 456-472.

[4] Lin, C. S., Smith, D. C. P. and Smith, J. M., "The Design of a Rotating Associative Memory for Relational Database Applications," ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976, pp. 53-65.

[5] Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

[6] Ozkarahan, E. A., Schuster, S. A. and Smith, K. C., "RAP -- Associative Processor for Data Base Management," AFIPS Conference Proceedings, Vol. 44, 1975, pp. 379-388.

[7] Ozkarahan, E. A. and Sevcik, K. C., "Analysis of Architectural Features for Enhancing the Performance of a Database Machine," ACM Transactions on Database Systems, Vol. 2, No. 4, December 1977, pp. 297-316.

[8] Moulder, R., "An Implementation of a Data Management System on an Assoicative Processor," Proceedings of the AFIPS National Computer Conference, Vol. 42, 1973, pp. 171-176.

[9] Banerjee, J., Baum, R. I. and Hsiao, D. K., "Concepts and Capabilities of a Database Computer" to appear in ACM Transactions on Database Systems. Also available in, Baum, R. I., Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer -- Part I: Concepts and Capabilities," Technical Report OSU-CISRC-TR-76-1, The Ohio State University, Columbus, Ohio, September 1976.

[10] Kannan, K., Hsiao, D. K. and Kerr, D. S., "A Microprogrammed Keyword Transformation Unit for a Database Computer," Proceedings of the Tenth Annual Workshop on Microprogramming, October 1977, Niagara Falls, New York; and Hsiao, D. K., Kannan, K., Kerr, D. S., "Structure Memory Designs for a Database Computer," Proceedings of ACM 77 Conference, October 1977, Seattle, Washington; also available in Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer -- Part II: The Design of the Structure Memory and its Related Processors," Technical Report OSU-CISRC-TR-76-2, The Ohio State University, Columbus, Ohio, October 1976.

[11] Kannan, K., "The Design of a Mass Memory for a Database Computer," Proceedings of the Fifth Annual Symposium on Computer Architecture, April 1978, Pal Alto, California; also available in Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer -- Part III; The Design of the Mass Memory and its Related Processors," Technical Report OSU-CISRC-TR-76-3, The Ohio State University, Columbus, Ohio, December 1976.

[12] IBM, Information Management System/ Virtual Storage (IMS/VS) Version 1, General Information Manual, GH20-1260-4.

[13] CODASYL Data Base Task Group Report, ACM, New York, April 1971.

[14] Banerjee, J., Hsiao, D. K., and Ng, F. K., "Data Network - A Computer Network of General-Purpose Front-end Computers and Special-Purpose Back-end Database Machines," Proceedings of International Symposium on Computer Network Protocols, (Danthine, A., Editor), Liege, Belgium, February 1978, pp. D6-1 to D6-12; also available in Hsiao, D. K., Kerr, D. S., and Ng, F. K., "DBC Software Requirements for Supporting Hierarchical Databases," Technical Report OSU-CISRC-TR-77-1, The Ohio State University, Columbus, Ohio, April 1977.

[15] Banerjee, J., Hsiao, D. K., and Kerr, D. S., "DBC Software Requirements for Supporting Network Databases," Technical Report OSU-CISRC-TR-77-4, The Ohio State University, Columbus, Ohio, June 1977.

[16] Banerjee, J. and Hsiao, D. K., "Performance Evaluation of a Database Computer in Supporting Relational Databases," Fourth International Conference on Very Large Data Bases, Berlin, Federal Republic of Germany, September 13-15, 1978; and Banerjee, J. and Hsiao, D. K., "The Use of a 'Non-Relational' Database Machine for Supporting Relational Databases," Fourth Workshop on Computer Architecture for Non-numeric Processing, Syracuse, New York, August 1-3, 1978; also available in Banerjee, J. and Hsiao, D. K., "DBC Software Requirements for Supporting Relational Databases," Technical Report OSU-CISRC-TR-77-7, The Ohio State University, Columbus, Ohio, November 1977.

[17] PTD-9300 Parallel Transfer Disk Drive, Ampex Corporation, Redwood City, California. (A product announcement communicated to the authors in May 1978.)

[18] Hoagland, A. S., "Magnetic Recording Storage," IEEE Transactions on Computers, Vol. C-25, No. 12, December 1976, pp. 1283-1289.

[19] McCauley, E. J. III, "Highly Secure Attribute-Based File Organization," Proceedings of the Second USA-Japan

Computer Conference, August 1975, pp. 497-501.

[20] Altman, L., "New Arrival in the Bulk Storage Inventory," Electronics, Vol. 51, No. 8, April 13, 1978, pp. 106-113.

[21] Hsiao, D. K. and Kannan, K., "Simulation Studies of the Database Computer (DBC)," Technical Report OSU-CISRC-TR-78-1, The Ohio State University, Columbus, Ohio, February 1978.

[22] Coulouris, G. F., et al., "Towards Content-Addressing in Data Bases," Computer Journal, Vol. 15, No. 2, February 1972, pp. 95-98.

DATA NETWORK - A COMPUTER NETWORK OF GENERAL-PURPOSE FRONT-END COMPUTERS AND SPECIAL-PURPOSE BACK-END DATABASE MACHINES.*

Jayanta Banerjee and David K. Hsiao
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210
USA

Fred K. Ng
Bell Telephone Laboratories
Naperville, Illinois 60540
USA

Very large databases on intelligent database stores are resources which may be shared among a large number of on-line users. Through a network of computers, they constitute a data utility providing shared, concurrent access with security and integrity for multiple users. The intelligent database stores are back-end database machines capable of performing a number of data management functions such as storage, search, retrieval, update, access control, clustering. Every database store is associated with one or more host computers in the computer network and is accessed via these front-end computers. In this paper we first motivate the notion of data networks consisting of geographically distributed front-end computers and back-end database machines. We next present the protocol necessary for communication between conventional front-end computers and the specialized database machines. We then proceed to show how hardware support of this protocol can greatly improve typical database management activities such as IBM's IMS found in the front-end computers. Finally, we point out that for high-performance and high-volume database management systems, data networks using such intelligent database machines may be necessary.

1. MOTIVATION AND REQUIREMENTS

A data network is to provide high-volume and high-performance on-line database management activities in a computer network environment. By high-volume we mean that there are one or more large databases in the network which are on the order of, say, 10^{10} bytes. High performance for on-line activities requires that the database stores be capable of content search, rapid retrieval, and efficient update. Such a computer network can support database management applications that cannot be found in conventional stand-alone database management systems. In addition to its limited capacity and fixed locality, a stand-alone database management system is always designed and implemented for a particular data model. On the other hand, in a data network of many database stores, multiple views can be accommodated by having some database stores to support one view and other database stores to support another view. With this support, the data network not only enables users to utilize distributed databases with their individual views, but also opens the opportunity for diverse groups of users to access a very large and centralized database where each group may view the database at a level of abstraction most suitable to the group. Consequently, the novice users may interact with the data network, for example, through a relational view; the application programmers may develop applications on the data network with a hierarchical view; and the transaction analysts may navigate the data network for transaction processing via a CODASYL view. Such use of the data networks is a step toward the goal of a data utility where a database, like public utilities, can be used by a wide spectrum of users for a large number of applications.

To meet this goal, the database stores of the data network must strive for facilities to provide multiple views, great capacity, high performance, low cost, good security, and high reliability. For example, the designs of database machines such as CASSM (1), DBC (2, 3, 4, 5), RAP (6), and RARES (7) are aimed to achieve some of the aforementioned requirements of the database stores. They may also be used to alleviate certain other problems associated with distributed databases. Maintaining consistency in a redundant distributed database (8,9) may be simplified with the use of database machines. In addition, since database machines can directly interpret high-level protocols, protocol standardization is greatly simplified, thus contributing to the ease of use of a distributed database. In this paper, we shall concentrate on the requirements of a database machine to support multiple views in a data network. We shall propose a database management protocol which is sufficiently primitive to be implemented in a database machine for high performance and reliability, and which is rich enough to support multiple data models with little software and security overhead. To study the cost and performance of the protocol, we compare the utilization of a popular type of database on a conventional stand-alone computer system with the utilization of the same database on a data network via the protocol. The result of the comparison is gratifying -- indicating that the direction towards data utility perhaps lies in the use of data networks.

*The work reported herein was conducted at The Ohio State University and supported by contract N00014-75-C-0573 from the Office of Naval Research.

2. THE NETWORK ENVIRONMENT

A large on-line database, intended to be shared by a network of computers, is supported on one or more intelligent database stores, depending on whether the database is centralized or distributed. A database store consists of a physical database and a back-end database machine (DBM) endowed with the capabilities of multiple views (or data models), a very large on-line storage capacity (of the order of 10^{10} bytes), high performance attained through intelligent search, efficient update and automatic sorting mechanisms, low cost, extensive security provisions and high reliability. A user sitting on a terminal or running his program in batch will access the database via a conventional front-end computer (FEC) such as a UNIVAC 1108, which communicates the data management needs to a DBM. The latter does the actual operations on the database. A minimal environment in which a DBM will exist is depicted in Figure 1.

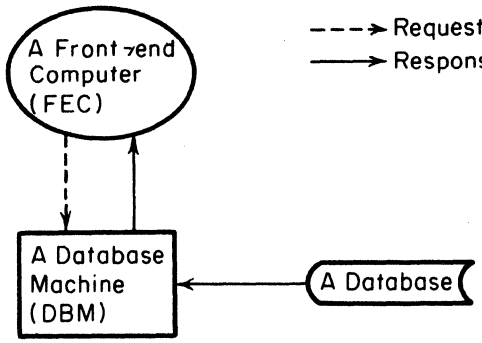


Figure 1. Minimal Network Environment for a DBM

The FEC compiles and executes user programs with the help of the DBM. The FEC directs all data management commands in the form of requests to the DBM. The DBM responds to such a request by conducting the activity (such as inserting a record, retrieving records satisfying a query) required by the request and sending a response back to the FEC. The response may consist of either control information, (such as a completion signal), an error message, or any data collected from the database (such as a group of records satisfying the query, a set operation on these records). The dialogue between an FEC and a DBM, then, is carried out with a data management protocol of requests and responses.

2.1 Centralized Data Networks

The minimal data network may be a simple network of two computers, a DBM and an FEC. A more elaborate network may consist of a network of FECs using a centralized database via a DBM. One such configuration called a centralized data network is shown in Figure 2. The centralized database is managed by a single DBM. A user of the database may log into one of the FECs, which interfaces with the DBM via data management protocols. The sharing of other computer facilities is done via inter-FEC protocols (such as those in (10)) which we shall not address in this paper.

2.2 Distributed Data Networks

Very often a database is distributed geographically over a number of regions. To every computer in a network, only a portion of the database

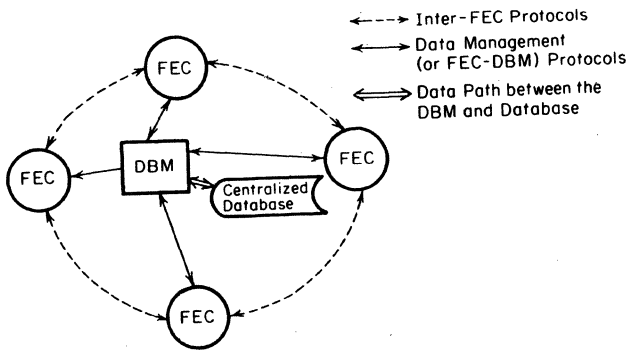


Figure 2. A Centralized Data Network

is locally available. The rest of the distributed database will be accessible to such a computer through its participation in the network. The software design of one such system is demonstrated in (11).

It is possible to extend the application of DBMs further by placing them in charge of various portions of the database in a distributed data network. There are various reasons for using DBMs in distributed data networks:

(1) faster response time due to quicker search and update capabilities of a DBM.
(2) reduced software in the FECs since all data management activities are handled by the DBMs, and
(3) standardized high-level data management protocols for any FEC-DBM communication.

In a distributed data network, the entire database is usually organized to support a single logical data model such as CODASYL (12), relational (13) or hierarchical (14). The logical view of the database (in the form of a schema and one or more subschemas) and a physical map of the database may be stored either in a central store accessible to all FECs or in a local store in each FEC. An example of a distributed data network is shown in Figure 3.

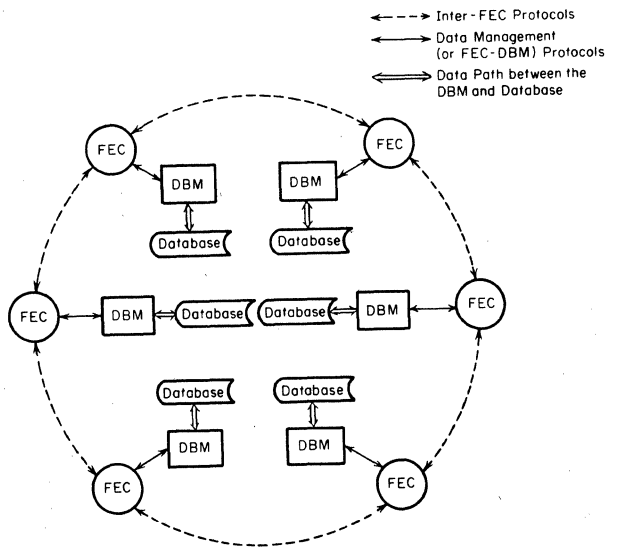


Figure 3. A Distributed Data Network

We now categorize distributed data networks as follows:

(1) networks with identical DBMs supporting identical data models,
(2) networks with identical DBMs supporting dissimilar data models,
(3) networks with dissimilar DBMs supporting identical data models, and
(4) networks with dissimilar DBMs supporting dissimilar data models.

The greatest generality occurs when a network consists of DBMs of different types and the databases are based on dissimilar data models. The problem of dissimilar DBMs can be easily eliminated if we can design a general protocol that is acceptable by every DBM. This will be our aim in the next section where we use the features required in all database machines to come up with a standard protocol design.

The problem of dissimilar data models is, however, more complex. Even if a specific program is to be limited to accessing only those databases that are represented by the same data model, it will be necessary to provide each FEC with a variety of schemas and subschemas as well as translators for the data languages of all the different models. The problem is further compounded when a single user program wants to access databases associated with a variety of data models. There is no known attempt as yet to address this problem. The problem is generally evaded by creating only homogeneous distributed databases, i.e., those that are based on a single data model. The protocol suggested in the next section is not a complete solution to the problems arising from dissimilar data models. However, it is rich enough to allow the expression of data management needs for any existing type of databases. This is substantiated in Section 4, where a hierarchical database management system is "simulated" with the proposed data management protocol.

3. DATA MANAGEMENT PROTOCOLS

In an attempt to provide a standardized and efficient means of communication between the DBMs and FECs in a data network, we advance a complete data management protocol that is implemntable by a database machine. Let us first extract a few salient features common to all machines with large on-line database stores. The secondary storage is made content-addressable in order to provide for rapid search and update. However, in spite of decreasing processing costs, it is doubtful whether it will ever become cost-effective to build a large secondary storage with monolithic associative memory. On the other hand, it is viable to partition the secondary storage into blocks where each block is individually content-addressable and where access is limited to a few blocks at a time. Consequently, it is desirable to cluster similar data into as few blocks as possible. Clustering information as well as some other structure information about the database is stored in a separate memory called the directory memory. The performance of the directory memory must be sufficiently high to minimize the number of accesses to the database store and to keep pace with the database store accesses. In addition, a database machine must provide content-based security mechanisms, set operations on retrieved data, and intelligent update capability.

Relying on the provision of the above features in a database machine, we shall now introduce a general attribute-based data model which will be the basis of the design of a data management protocol for communication between front-end computers and back-end database machines.

3.1 An Attribute-Based Data Model

The data model represents a logical view of the data as seen from a front-end computer (FEC) which intends to interact with a database via a database machine (DBM). This model may also represent the physical organization of the data stored by the DBM. There are four aspects associated with the data model: data structure, query, clustering, and security. The data structure is the way that information is represented so that there is a uniform way to access and manipulate the information. Access and manipulation of information is done with the use of queries. Clustering affects the way the data is physically grouped into partitions; security controls the way the data structure is protected from unauthorized access.

The database is composed of records where a record R is a set of ordered pairs of the form:

<an attribute, a value>.

The database may be partitioned into subsets called files, each with a unique file name.

A keyword K is an attribute-value pair which characterizes the record. A keyword predicate is a triple of the form <attribute, relational operator, value>. A relational operator is an element of the set $\{=,\neq,<,\leq,\geq,>\}$. A keyword $<A,V>$ is said to satisfy a keyword predicate $<A_p, O_p, V_p>$ if and only if $A=A_p$ and $V\ O_p\ V_p$, i.e., V and V_p are related by the relational operator O_p. A query conjunct is a conjunction of keyword predicates. Finally, a query is a Boolean expression consisting of disjuncts of query conjuncts. An example of a query is

([DEPT='TOY']∧[SALARY<10000])∨
([DEPT='BOOK']∧[SALARY>50000]).

If the above query refers to a file of employees of a department store, then it will be satisfied by records of the employees working either in the toy department and earning less than 10,000 or working in the book department and making more than 50,000. We note that this query consists of two query conjuncts each of which consists of two keyword predicates.

Queries are used not only to retrieve a set of records from the database, but also to specify security requirements and clustering conditions. Security specifications are based on the actual contents of the database. A database access or, simply, an access is the name of an operation which transfers information to or from the database. Examples of accesses are retrieve, insert, and delete. For every user of the database, there exists a database capability, which is simply a list of file sanctions whose entries are of the form:

$(F, [Q_1, A_1], [Q_2,A_2],\ldots,[Q_n,A_n])$

where F is a file name, each Q_i is a query and each A_i is a set of accesses. The database capability of a user determines the records he can access. For example, for a user to be allowed to perform an access operation a on a record R of file F, the following condition must hold for every (Q_i,A_i) in this file sanction for F:

If R satisfies Q_i then $a \in A_i$

Keywords appearing in the queries of a file sanction are designated as security keywords. This type

of security specification is powerful and elegant. With this specification, not only can security be enforced in terms of record types or entire files, but security can also be facilitated at a much more detailed level based on the actual content of the records in the database.

Since the secondary storage of a DBM is partitioned into blocks and only one block is accessible at a time, it may be convenient for the user to have some control over the placement of records in the physical storage. Such control is achieved through clustering keywords. For each file, there are certain keywords which are designated for record placement purposes and are called clustering keywords. The occurrence of some of these keywords within a record defines a cluster. When a record is to be inserted into the database, the user can specify clustering conditions which enable the DBM to determine the identity of the cluster in which the record should belong. These conditions are queries consisting of clustering keywords.

The power of the clustering (security) mechanism is derived from the fact that it utilizes the same query facility for clustering (protecting) as is used for retrieving records. Security and clustering keywords are instances of a general class of keywords called Type-D keywords or directory keywords. For every Type-D keyword specified by the user, the DBM maintains an entry in its directory memory associating the keyword with all the content-addressable blocks in which records containing this keyword appear.

3.2 Protocol Primitives

DBM-FEC communication is achieved through a data management protocol used for making requests from an FEC and responses from a DBM. Each of the FECs in the network has a unique identification which we shall refer to as the FEC-ID. Similarly, every DBM in the network has a DBM-ID. Within a single DBM, all the files have unique identification and the FECs are aware of the file identification that they might have to refer to. Each request sent from a source (FEC) to a destination (DBM) is distinguished from all other requests from the same source by means of a request-id, R-ID. Thus, the response data retrieved by the destination can be identified by the source if the corresponding R-ID is included in it. All requests received by a DBM at any instant are queued and executed in an order of importance that depends on the priority associated with a request with respect to the priority of other requests from the same source, and the content-addressable blocks to be accessed in executing the request. Since an FEC has neither the control nor the knowledge of the location of data in the secondary storage, it can only ensure the correct order of execution of two requests to the same DBM by doing either of the following:

(1) Sending the second request only after receiving the response to the first, or
(2) Assigning a higher priority to the first request and then sending it before the second. In an ARPA-like network, this ensures the right ordering.

When the delay involved in the first approach is intolerable, the second solution presents a very good alternative.

The request-response (or request-reply) protocol consists of the following two messages:

(1) Request: <FEC-ID, R-ID, R-CODE, priority, user ID, argument set>
(2) Response: <FEC-ID, R-ID, E, result set>

The FEC-ID is the identification number of the source of the request. R-ID, together with the FEC-ID, uniquely identifies a request in the entire network. The R-CODE is a coded version of the request name. The priority of a service can be one of several possible levels. The argument set includes, besides other arguments, a file-ID indicating the file upon which the request is to be carried out. In the response message, E is an error bit indicating whether the request has been satisfactorily (correctly) carried out or not.

Before we describe the requests recognized by a DBM, let us first describe some elementary items that may be used to build up a request. A query, as we have seen earlier, consists of a disjunction of query conjuncts. A query conjunct, in turn, is made up of a conjunction of keyword predicates. A keyword predicate is a triple of the form <attribute, relational operator, value>. If each attribute in a file is given a unique attribute number, then the format of a keyword predicate is as shown in Figure 4. The format of a keyword within records is the same as that of a keyword predicate, except for the fact that bits representing the relational operator (bits 3 through 7) are set to zero. The format of a query conjunct is shown in Figure 5 and that of a record in Figure 6. The format of a query in disjunctive normal form is depicted in figure 7. A clustering condition consists of a query that consists entirely of clustering keyword predicates. The format of such a condition is shown in Figure 8. The format of a file sanction as shown in Figure 9 is made up of a query conjunct and an access descriptor indicating the type of access allowed on the records satisfying the query conjunct. The format of an access descriptor is as depicted in Figure 10. Finally, security keywords are identified by security descriptors of the format shown in Figure 11. When a range of values for a particular attribute must be secured, the lower and upper bounds for that attribute must be specified in its security descriptor.

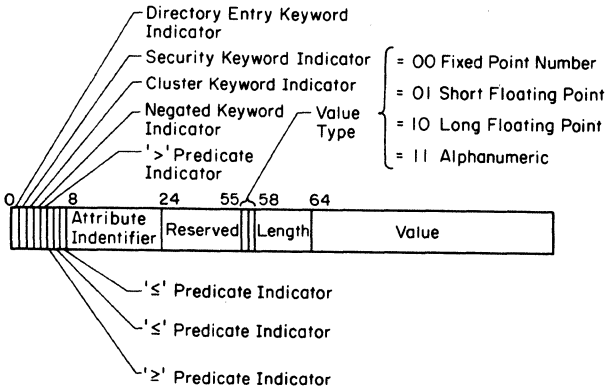


Figure 4. Format of a Keyword Predicate (T)

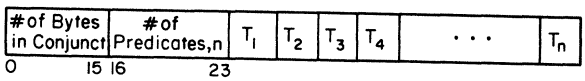


Figure 5. Format of a Query Conjunct ($T_1 \wedge T_2 \wedge .. \wedge T_n$)

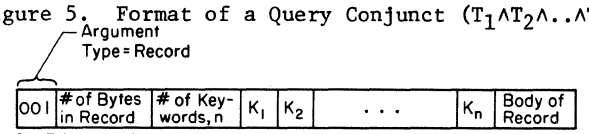


Figure 6. Format of a Record with Keywords $K_1, K_2 \ldots K_n$

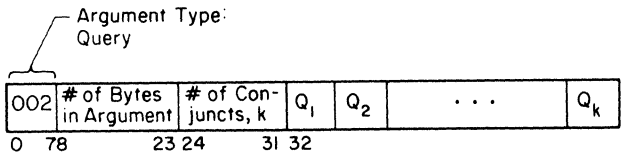


Figure 7. Format of a Query

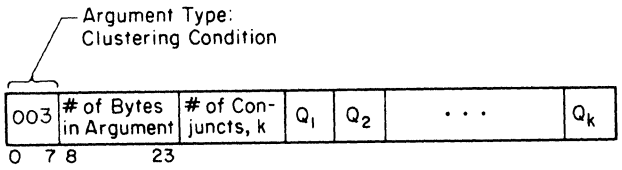


Figure 8. Format of a Clustering Condition

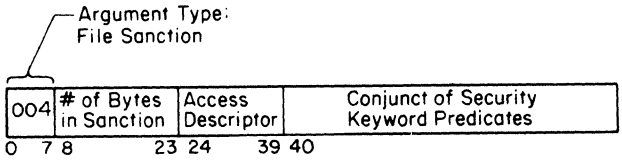
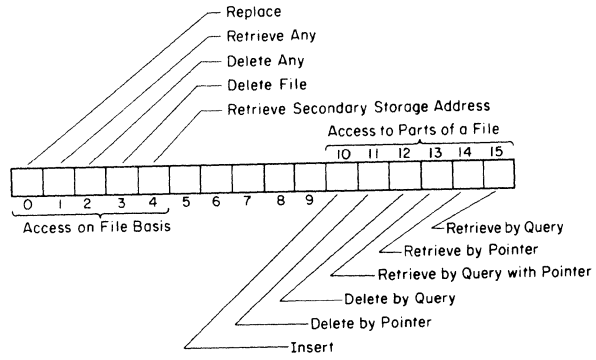


Figure 9. Format of a File Sanction Consisting of a Conjunct of Security Keyword Predicates and an Access Privilege.



Note: A '1' in a bit position indicates a right to perform the access, while a '0' indicates a denial of the right.

Figure 10. Format of an Access Descriptor

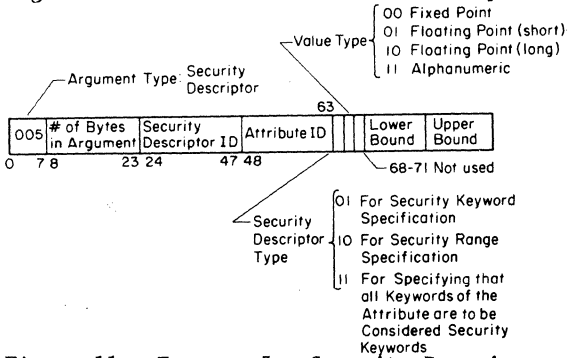


Figure 11. Format of a Security Descriptor

3.3 Preparatory Requests

Requests recognized by a DBM may be categorized as preparatory, retrieval and data manipulation (update) requests. The preparatory requests are used for file creation, security specification, etc. They are described below:

(1) Open-database-file-for-creation (Figure 12): This request is sent to the DBM before records of the file are loaded into the database. It provides information on the number of attributes the file is to have, the number of secondary storage blocks that need to be allocated initially, and the number of blocks that may be allocated if the initial allocation is insufficient.

(2) Load-attribute-information (Figure 13): This request is used to provide a DBM with information on the attributes used for the file.

(3) Load-security-descriptor (Figure 14): The security descriptors of a file are loaded by this request.

(4) Close-database-file (Figure 15): This request indicates that the file may be deactivated i.e., to indicate that there will be no more requests from the user on the file.

(5) Open-database-file-for-access (Figure 16): Since the processing for creation of a database file is different from that for accessing a file, this separate request is provided.

(6) Load-creation-capability-list (Figure 17): This request is used to indicate to the DBM the identity of the users who may issue the open-database-file-for-creation request.

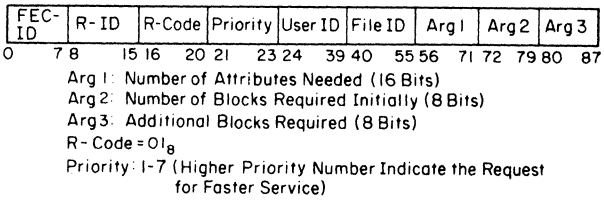


Figure 12. Format of the Open-Database-File-for-Creation Request

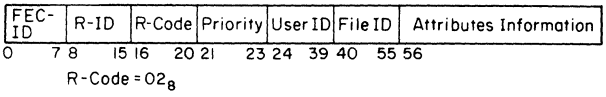


Figure 13. Format of the Load-Attribute-Information Request

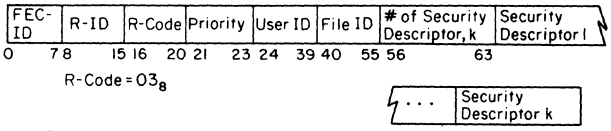


Figure 14. Format of the Load-Security-Descriptor Request

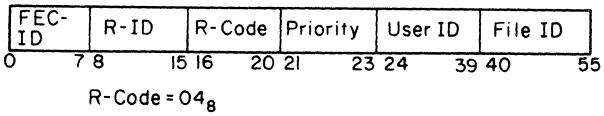


Figure 15. Format of the Close-Database-File Request

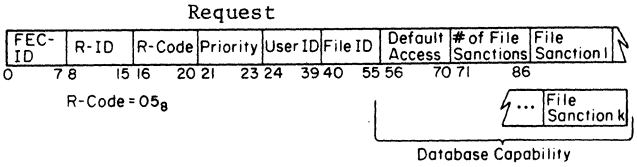


Figure 16. Format of the Open-Database-File-for-Access Request

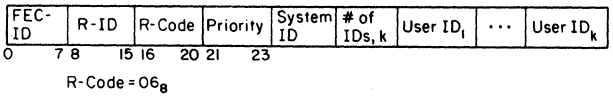


Figure 17. Format of the Load-Creation-Capability-List Request

3.4 Retrieval Requests

There are four requests that may be used to retrieve records from the database. In some of these, it is possible to specify whether the records satisfying a query need be sorted or not, whether a set operation should be performed on the records, etc. Another request is reserved for the join operation.

(1) Retrieve-by-query (Figure 18): In this, a query made of keyword predicates in the disjunctive normal form is used to identify records desired by the user. Sorting of the records can be specified, if necessary. If a set operation is desired on a particular field, then the appropriate function (MAX, MIN, AVG, SUM, COUNT, NULL) may be specified in bits 73-75. NULL is not actually a set function. It only indicates that set function is not desired, and in that case no attribute should be included for set function (bits 76-91). Bit 92 indicates whether only the set function need be returned or the records as well. If records are to be returned then either each reocrd may be returned in its entirety or only certain fields of it. This is specified in bit 93.

(2) Retrieve-by-query-with-pointer (Figure 18): This request is similar to the retrieve-by-query request except that the positions (in the database) of the records satisfying the query are also returned in the form of pointers.

(3) Retrieve-by-pointer (Figure 19): This request is used to retrieve a single record when the location of the record is known.

(4) Retrieve-and-connect (Figure 20): With this request, two groups of records are retrieved using two different queries. These records are then joined on two attributes using the same property (i.e., the attribute names may be different but their values are taken from the same domain). For example, let three records retrieved by query 1 be
(<A, VA1> <B, VB1> <C, VC1> <D, VD1>)
(<A, VA2> <V, VB2> <C, VC2>)
(<A, VA3> <B, VB3> <C, VC3> <D, VD3>)
and two records retrieved by query 2 be
(<E, VE1> <F, VF1>)
(<E, VE2> <F, VF2>).

If we now join on attributes C and E, and extract fields B and D for query 1 and field F for query 2, then the two new records returned are formed with the following pairs:
(<B, VB2>,<F, VF1>) and (<B, VB3> <D, VD3>, <F, VF2>)
where we have assumed that the values VC2 and VE1 are equal and the values VC3 and VE2 are also equal. In Figure 20, all information related to query 1 is numbered 1, and all information associated with query 2 is numbered 2.

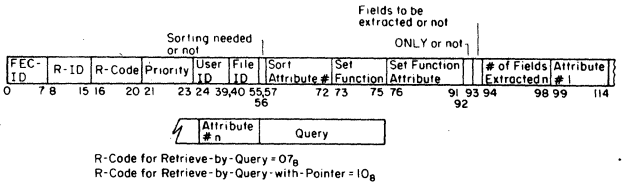


Figure 18. Format of a Retrieve-by-Query Request and Retrieve-by-Query-with-Pointer Request

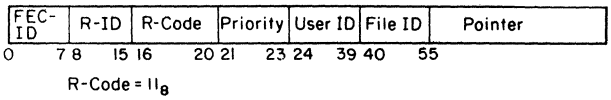


Figure 19. Format of a Retrieve-by-Pointer Request

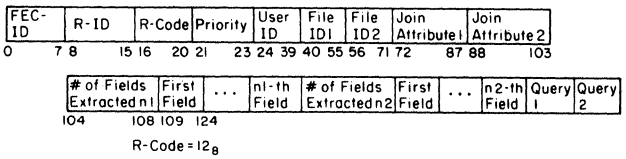


Figure 20. Format of a Retrieve-and-Connect Request

3.5 Data Manipulation and Update Requests

There are six different requests reserved for database loading (creation), record insertion, record deletion, and record modification. We describe these requests in turn:

(1) Load-record (Figure 21): During the creation of a file, the user may not perform any access operation except for loading records. The clustering conditions included in this request are used for physical clustering of the records. No security check is made since the load-record request follows the open-database-file-for-creation request which causes security checking.

(2) Insert-record (Figure 21): This request is used to add records to an existing file. It undergoes a security check before the operation is carried out. In the load-record request, in contrast, the time required for security checking is avoided, thus accelerating the database loading operation.

(3) Delete-by-query (Figure 22): This request is very similar (in processing) to the retrieve-by-query request. It uses a query to identify those records that have to be removed from the file.

(4) Delete-by-pointer (Figure 23): This is similar to the retrieve-by-pointer request and is used to delete a specific record.

(5) Delete-file (Figure 24): Often a user may wish to destroy an entire file. This action is provided by the delete-file request. It not only releases the database areas or blocks occupied by the file, but also the directory memory space occupied by the keyword directory entries and auxiliary information kept by the DBM.

(6) Replace-record (Figure 25): In database operations, it is frequently desired to update certain fields of a record and retain only the updated version of the record. Such a facility is provided for by the

replace-record request. There are two arguments to this request--a pointer to the old record that is to be replaced and the new record that is to replace the old record. Internally, this request is divided into two parts--a delete-by-pointer for the old record, and an insert-record protocol for the new record.

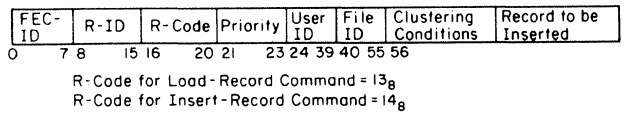


Figure 21. Format of a Load-Record Request and Insert-Record Request

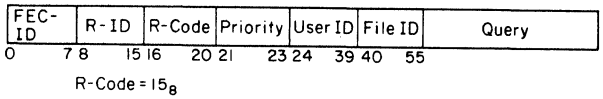


Figure 22. Format of a Delete-by-Query Request

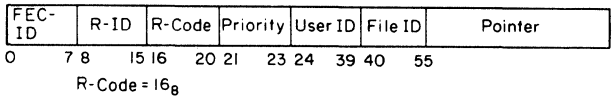


Figure 23. Format of a Delete-by-Pointer Request

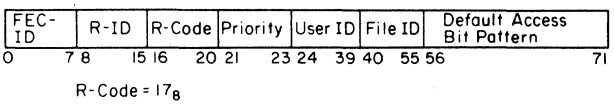


Figure 24. Format of a Delete-File Request

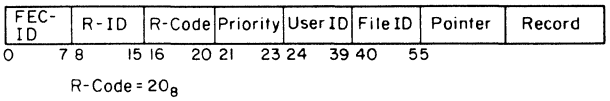


Figure 25. Format of a Replace-Record Request

3.6 DBM Response Messages

The data returned to the source FEC in response to a request will first identify the request and then include the results of the request. The format of a DBM response to preparatory and update requests is shown in Figure 26.

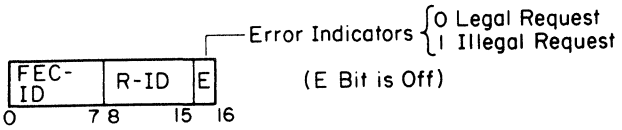


Figure 26. Response to Preparatory and Update Requests

The response only acknowledges to the source that the request has been carried out. A retrieve-by-query request may cause a DBM to return a set function or a number of records or both (Figure 27).

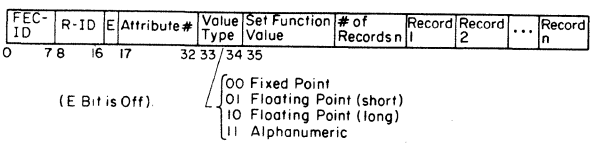


Figure 27. Response to a Retrieve-by-Query Request

If a set function is to be returned then its attribute number and value are first included. The records returned may only be partial records since only certain keywords will be extracted, corresponding to the fields specified in the request. The response to a retrieve-by-query-with-pointer request (Figure 28) will be similar to that of a retrieve-by-query request except that record pointers will also be a part of the response. The retrieve-by-pointer request causes the return of only a single record (Figure 29). The response to a retrieve-and-connect request includes several pairs of records (Figure 30). The first part of a pair is a record satisfying the first query of the request, and the second part of the pair is a record satisfying the second query. The records are only partial records since only those fields that are specified in the request are extracted and included. In case a request cannot be carried out due to an illegal structure, an error message is returned, the format of which is shown in Figure 31. The E-bit is turned on.

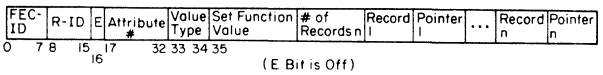


Figure 28. Response to a Retrieve-by-Query-with-Pointer Request

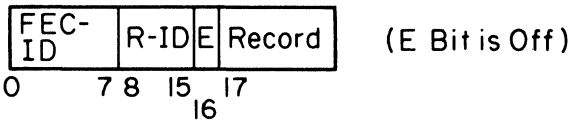


Figure 29. Response to a Retrieve-by-Pointer Request

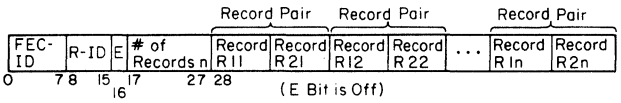


Figure 30. Response to a Retrieve-and-Connect Request

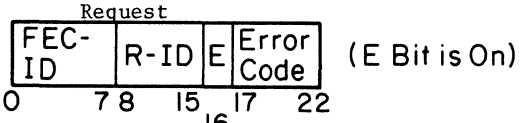


Figure 31. Format of an error message

4. A CASE FOR PERFORMANCE

In this section we first demonstrate how an existing database management system software, such as IBM's Information Management System (IMS) (14,15,16) can be replaced with database machines that communicate with FECs using the suggested protocol. The entire database is assumed to be represented by a single data model, and the DBMs are assumed to be capable of interpreting the data management protocol of Section 3. Whenever an FEC requires the use of the database in a data network, it first determines the identity of the DBM with whom it should communicate. It then uses the data management protocol to send a request to that DBM.

We conclude this section by considering a specific database machine, the DBC, which directly supports the above protocol. Additionally, we analyze the performance of the DBC in the data network.

4.1 The Information Management System (IMS)

IMS is the one most important example of a hierarchical database management system. An IMS database consists of a number of hierarchically related segments, each of which belongs to a segment type. An example of an IMS database is shown in Figure 32. The segment type A has three occurrences. It is called the root segment type Some relationships among the various segments in our example are:

A1 is the parent of V1 and G1.

H1, H2, and I1 are children of G1.

E1 and E2 as well as F2 and F3 are twins.

IMS application programs must traverse the segments of the database in order to make retrievals. The convention of traversing is from top to bottom (parent to child), front to back (among twins), and left to right (among children). The database in Figure 32 would be traversed in the order of A1, B1, C1, D1, D2, D3, E1, F1, E2, F2, F3, G1, H1, H2, I1, J1, J2, A2, A3. Notice that the traversal order defines a next segment with respect to a given segment. Finally, a hierarchical path is a sequence of segment occurrences, one per level. proceeding directly from a segment at one level to a particular segment at a lower level. For example, A1, G1, I1, J2 is a hierarchical path.

An IMS user processes an IMS database with

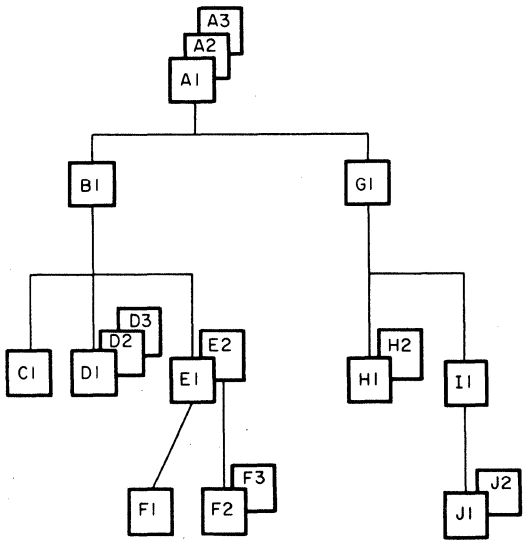


Figure 32. Schematic Representation of an IMS Database

application programs using Data Language/1 (DL/1). A DL/1 call has the following format:

FUNCTION SEARCH-LIST

where FUNCTION is one of insert (ISRT), delete (DLET), replace (REPL), or a form of get (GET), and where SEARCH-LIST is a sequence of segment search arguments (SSA), possibly one per level, which are used to select a hierarchical path. The basic function of the SEARCH-LIST is to narrow the field of search. It has the form

SSA_1 SSA_2 ..... SSA_n

where each segment search argument (SSA) is of the form

<segment-type> <Boolean expression>

with Boolean expression relating values of fields of the given segment type. The Boolean expression need not appear, in which case we say that the SSA is unqualified; otherwise it is qualified.

4.2 Representing an IMS Database in the Attribute-Based Model

An FEC can represent an IMS database by viewing every IMS segment as a record composed of keywords. A record is as defined in the attribute-based model of Section 3. A DBM may be unrestricted in its way of representing data, but it is equipped with a mechanism to interpret the data management protocols (of Section 3) expressed in the attribute-based model. Thus, an FECs view of a database can be supported by a DBM.

Address-dependent pointers are totally discarded from records of the attribute-based model (or simply, records) and enough information is stored in the form of keywords to show the relationship among various segments. An IMS segment includes a sequence field whenever it is necessary to indicate the ordering among twin segments. Since each segment becomes a record and no address-dependent pointers are included, we assign a symbolic identifier to each segment, identifying it uniquely from all other segments in the database. The symbolic identifier of a segment S is a group of fields consisting of (1) the symbolic identifier of the parent of S, and (2) the sequence field of S. The only ambiguity that may occur is when the sequence fields of different types of segments have the same field name. This can easily be resolved by qualifying the field name with the segment type.

The creation of a record from an IMS segment is done by forming keywords as follows (17):

(1) For each field in the segment, form a keyword using the field name as the attribute and the field value as the value.

(2) Form a keyword of the form <TYPE, segtype> where TYPE is a literal and segtype is the type of the segment in consideration.

(3) For each sequence field in the symbolic identifier of the segment, form a keyword using the field name as the attribute and the field value as the value.

For example, for the logical data structure of an IMS database shown in Figure 33, the attribute templates of the five types of records corresponding to the five segment types are shown in Figure 34. Qualified field names such as PREREQ.COURSE# are used to resolve ambiguity among field names.

The pattern of access of the records should determine the clustering policy. Clustering is desirable because the secondary storage blocks

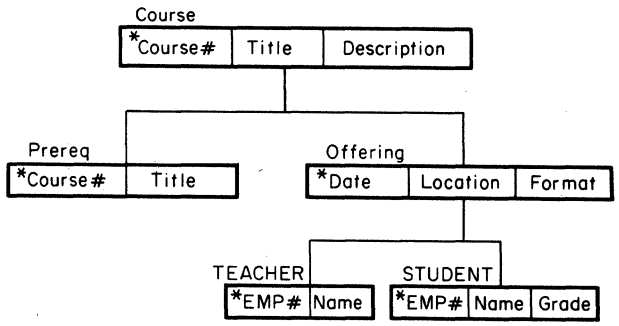


Figure 33. The Logical Data Structure of an IMS Database

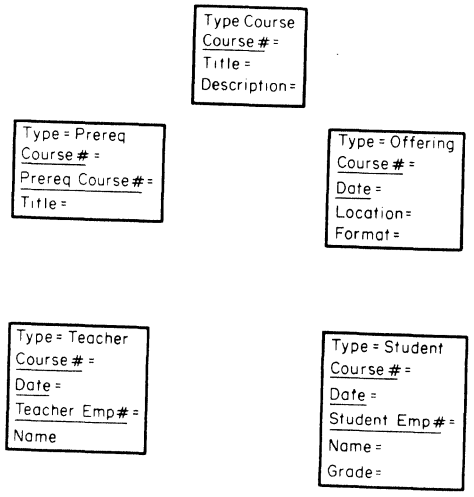


Figure 34. The Attribute Templates of DBC Records for the Segments of Figure 33

are individually content-addressable. Since the traversal of an IMS database is usually along a hierarchical path, the clustering policy is to first cluster the records which represent all the IMS root segments and then cluster the records which represent all dependent segments.

4.3 Translation of DL/1 Calls into Data Management Protocols

The FECs interface with the users by executing their application programs that use DL/1 calls. These calls are first translated into DBM requests using the suggested protocol, and sent to the appropriate DBM via the communication lines. For each user, an FEC maintains a user area called the interface system buffer (ISB) since this area provides the interface between the user and the database. The information obtained in the course of executing a DL/1 call is maintained in the ISB and consists of records which are retrieved from the database in the execution of the DL/1 call.

We shall briefly illustrate the translation of a get-unique (GU) call. Consider the IMS database of Figure 33 to which the following call is addressed:

```
GU    COURSE     (TITLE='MATH')
      OFFERING   (LOCATION ='STOCKHOLM')
      STUDENT    (GRADE='A')
```

This DL/1 call is intended to determine the first student in the database (along the hierarchical path) who has obtained an A-grade in the Math course offered at Stockholm.

Now we shall illustrate how the content of the ISB is established during the execution of the above GU call.

(1) Starting with the first SSA in the call, i.e., COURSE (TITLE='MATH'), the COURSE segments which satisfy the qualification TITLE='MATH' are retrieved from the DBM and put into the ISB. These segments are retrieved by a retrieve-by-query request with the query being (TYPE=COURSE ∧ TITLE='MATH'). The segments are sorted by the DBM according to the values of their sequence fields.

(2) The first COURSE segment in the ISB is taken as the current COURSE segment.

(3) The OFFERING segments, children of the current COURSE segment, are then retrieved with the qualification (LOCATION ='STOCKHOLM') and stored in the ISB in the order defined by their sequence field. If the symbolic identifier of the current COURSE segment were (COURSE#=M5), then the query created for this retrieval would be (TYPE =OFFERING ∧ COURSE#=M5 ∧ LOCATION='STOCKHOLM').

(4) If no segment can be retrieved in step 3, then the next COURSE segment in the ISB is established as the current COURSE segment and step 3 is repeated.

(5) Supposing some OFFERING segments are retrieved and stored in the ISB, the first OFFERING segment in the ISB is taken as the current OFFERING segment.

(6) A process similar to step 3 and step 4 is performed to retrieve the first segment with qualification (GRADE='A')

(7) The first of the retrieved STUDENT segments is sent to the user.

In creating the IMS database it is necessary to execute the following data management requests: open-database-file-for-creation, load-attribute-information, load-security-descriptor, load-creation-capability-list and load-record. Before and after the files are accessed the open-database-file-for-access and close-database-file requests are used. The execution of the GET calls require the use of retrieve-by-query and retrieve-by-query-with-pointer requests. The ISRT call requires the use of insert-record request, and the DLET call requires the use of delete-by-query request. The REPL call can use either the replace-record request or a sequence of three requests retrieve-by-query, delete-by-query, and insert-record. Although the retreive-and-connect request and the set functions are not required in supporting hierarchical databases, they can fruitfully be employed in simulating other data models such as relational.

We shall now provide a brief description of a specific database machine, the DBC, and show how hardware support of the data management protocol greatly enhances the performance of data management activities in a network environment.

4.4 DBC Architecture

The Database Computer (DBC) [2,3,4,5] has an architecture that satisfies most of the requirements of a database machine. The DBC provides for the entire database an on-line secondary storage known as the mass memory (MM) which is partitioned into blocks and each block is called a minimal access unit (MAU). The MAUs are individually content-addressable and only one MAU is accessed at a time. The MM is built on modified disk units, each cylinder being an MAU. Since a typical disk cylinder can store up to 10^6 bytes, a 10^{10} byte database can be accommodated in 10,000 content-addressable MAUs.

Another major component of the DBC is a processor called the database command and control processor (DBCCP). When a request from an FEC is sent to the DBC, the DBCCP decodes it, determines the MAUs to be searched in order to satisfy the request, issues appropriate orders to the MM and transfers the response data back to the FEC.

The directory memory of the DBC is known as the structure memory (SM) which stores an entry for every Type-D keyword identifying the MAUs in which records containing this keyword appear. Specialized information is maintained for those Type-D keywords that also serve as security or clustering keywords. The SM, like the MM, is also content-addressable with lower capacity and higher processing speed. Typically, directories are of the order of 1% to 10% of the database. Therefore, the SM has a capacity of 10^7 to 10^9 bytes. The relationship among SM, MM, and DBCCP is depicted in Figure 35.

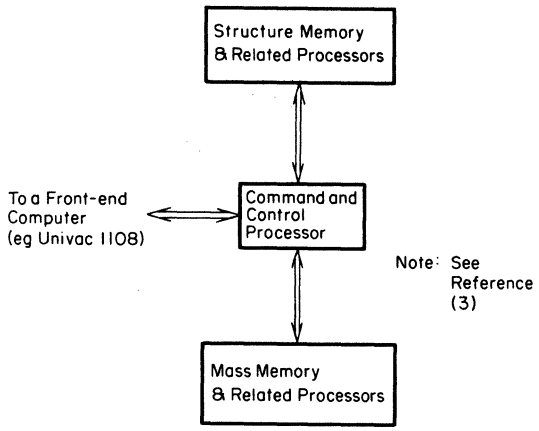


Figure 35. Basic Architecture of the DBC

The processors associated with the MM have the capability of returning a group of records (satisfying a query) in a sorted order, say, sorted by a given attribute. The processors can also carry out certain set operations. In particular, they can take a group of records and determine the minimum (MIN), maximum (MAX), sum (SUM), and average (AVG) of the various values of a given attribute considering all the records in the group. The number of records satisfying a query (COUNT) can be counted by hardware. Furthermore, any specific combination of fields of a record may be returned (on request) to the user, rather than the record in its entirety.

4.5 DBC Performance in a Network Environment

Hardware support of the data management protocol greatly increases the speed of database activities. The DBC is an example of a database machine that interprets and executes FEC requests in a direct manner. We , therefore, find a considerable improvement in performance when IMS is supported on a DBC-FEC environment rather than a regular IMS environment (conventional computers and software). A simple but moderately large database will be considered as a sample database to demonstrate this improved performance in terms of the number of physical block transfers required to process user transactions.

The IMS database used in our example is shown in Figure 36. We assume that it has 1,000 root segments and the length of each segment is 200 bytes (including pointers, etc.). Each root segment has 30 children (i.e., OFFERING segment occurrences) of length 100 bytes each. Each OFFERING segment has 50 children (i.e., STUDENT segment occurrences) of length 100 bytes each. The entire IMS database is linearized according to its traversal sequence and segments are loaded in this order

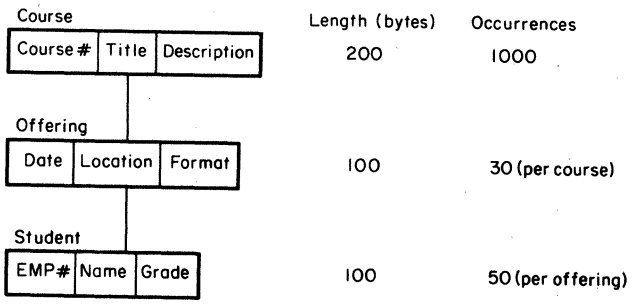


Figure 36. A Sample Database

into the blocks of the data file. For instance, the first m segments in the sequence are stored in the first block of the data file. The next m segment will be stored in the second block, and so on. A COURSE segment and all of its dependent segments consists of 153,200 (=200 + 30 x 100 + 50 x 30 x 100) bytes. Assuming a block size of 4K bytes, which is a favorable page size, this course information will spread across 39 (+153,200/4000) blocks.

The case studies used in the comparison of DBC and IMS environments involve a variety of processing situations including retrieval of a specific segment, retrieval of a number of segments, sequential traversal of the entire database, and addition of segments to and deletion of segments from the IMS database. The comparison is based on the number of disk accesses required by IMS and by the DBC.

Case 1: To retrieve a specific STUDENT segment

```
GU      COURSE      (COURSE#=CIS211)
        OFFERING    (DATE=730105)
        STUDENT     (EMP#=1684)
```

In the IMS environment: We will consider the best case and the worst case. If the STUDENT segment which satisfies the call is the first STUDENT segment in the IMS database record, then the number of disk accesses can be calculated as follows:

(1) One disk access to the index for the root segment (assuming that the index is small) to locate the block containing the root (i.e., COURSE) segment.
(2) One disk access to retrieve the block containing the root segment.

Since the required STUDENT segment is stored in the same block as the root segment, no more database accesses are required. Hence the total number of disk accesses is two.

If the STUDENT segment which satisfies the call is the last STUDENT segment in the IMS database record, then the number of disk accesses can be calculated as follows:

(1) One disk access to the index to locate the block containing the root segment.
(2) 30 more disk accesses to traverse from the first OFFERING segment to the last using the twin pointers (since there are 30 OFFERING segments and since we may assume that each of them is located in a different block, there will be 30 disk accesses. The justification of assuming different blocks is as follows. On the average, there are 50 students per offering, each STUDENT segment requiring 100 bytes. Thus, the average physical distance in bytes between two adjacent OFFERING segments is 5Kbytes, which is larger than a page. We may, therefore, expect at least one disk access per OFFERING segment).

(3) One disk access to traverse from the last OFFERING segment to the last STUDENT segment (since the OFFERING segment and its last STUDENT segment are located in different blocks).

Hence, the total is 32 disk accesses.

In the DBC environment: The number of disk accesses is calculated as follows:

(1) One disk access to retrieve the root segment.
(2) One disk access to retrieve the OFFERING segment.
(3) One disk access to retrieve the STUDENT segment.

The total is 3 disk accesses.

The remaining four are analyzed (17) in a manner similar to Case 1. The result of the case studies is tabulated in Figure 37.

Case 2: To retrieve a number of STUDENT segments

```
        GU   COURSE     (COURSE#=CIS211)
             OFFERING   (LOCATION=LONDON)
             STUDENT    (GRADE='B')
LOOP    GN   COURSE     (COURSE#=CIS211)
             OFFERING   (LOCATION=LONDON)
             STUDENT    (GRADE='B')
         GO TO LOOP
```

Case 3: To sequentially traverse the entire IMS database

```
        GU   COURSE
LOOP:   GN
        GO TO LOOP
```

Case 4: To insert a new STUDENT segment

```
        ISRT COURSE     (COURSE#=CIS211)
             OFFERING   (DATE=730105)
             STUDENT
```

Case 5: To delete a COURSE segment

```
        GHU  COURSE     (COURSE#=CIS211)
       DLET
```

Case Study		Environments: IMS	DBC-FEC
1. To retrieve a specific segment	min	2	3
	max	32	3
2. To retrieve a number of segments	min	31	2
	max	40	32
	"med"	36	17
3. To traverse the entire database		39000	31000
4. To insert a segment	min	3	1
	max	33	2
	"med"	18	3
5. To delete a segment		40	2

Note: "med" = (min + max)/2

Figure 37. Summary of Performance Comparisons

In the above analysis, the number of database accesses has been considered for performance measurement. It is a reliable measure since it is anticipated that for all large databases secondary storage will be one or more orders of magnitude slower than primary memory. With such a measure, the use of a DBC in a network environment considerably enhances database performance in handling hierarchical databases. In a similar study, it has been demonstrated (18) that the same data management protocol can support database management systems of the CODASYL type (12). It has been shown that in supporting CODASYL databases, a performance improvement of at least an order of magnitude can be expected when using DBCs in a network environment in lieu of software systems running on conventional general-purpose computers.

5. CONCLUDING REMARKS

We have proposed in this paper the use of data networks, consisting of conventional computers and specialized database machines, to manage large on-line centralized and distributed databases. To this end, we have advanced a high-level data management protocol. The protocol is not tied to any particular database machine. It, in fact, reflects a goal that should be pursued in the hardware design of any database machine. This was the case when a specific database machine, the DBC, was designed to function in a network environment. Furthermore, the protocol is powerful enough to support a variety of user views (data models) of the database.

In this paper, we have also studied through an example the performance of database machines that are capable of interpreting the proposed protocol by direct hardware means. The result of the performance study is encouraging. It points at a fruitful pursuit for hardware and network solutions as alternatives for better database management.

REFERENCES

[1] Su, S.Y.W. and G. J. Lipovski, "CASSM: A Cellular System for Very Large Databases," Proc. International Conference on Very Large Data Bases, Sept. 1975, pp. 456-472.

[2] Hsiao, D.K. and K. Kannan, "The Architecture of a Database Computer-A Summary," Proc. of the 3rd ACM Workshop on Computer Architecture for Non-Numeric Computation, Syracuse, May 16-17, 1977.

[3] Baum, R.I., D.K. Hsiao, and K. Kannan, "The Architecture of a Database Computer-Part I: Concepts and Capabilities," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-1, Sept. 1976,
Hsiao, D. K. and K. Kannan, "The Architecture of a Database Computer-Part II: The Design of Structure Memory and its Related Processors," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-2, Oct. 1976, and
Hsiao, D.K. and K. Kannan,"The Architecture of a Database Computer-Part III: The Design of the Mass Memory and its Related Processors," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-3, Dec. 1976.

[4] Hsiao, D.K. and S.E. Madnick, "Data Base Machine Architectures in the Context of Information Technology Evaluation," Proc. of the 3rd International Conference on Very Large Data Bases, ACM, New York, 1977.

[5] Hsiao, D.K., K. Kannan, and D.S. Kerr, "Structural Memory Designs for a Database Computer," Proc. National ACM Conference, ACM, New York 1977.

[6] Ozkarahan, E.A., S.A. Schuster, and K.C. Smith, "RAP-An Associative Processor for Database Management," Proc. National Computer Conference, AFIPS, 1975, pp. 379-387.

[7] Lin, C.S., D.C.P. Smith, and J.M. Smith, "The Design of a Rotating Associative Memory for Relational Database Application," ACM Transactions on Database Systems, 1, 1, March, 1976, pp. 53-65.

[8] Rothnie, J.B. and N. Goodman, "A Study of Updating in a Redundant Distributed Database Environment," Tech. Rep. No. CCA-77-01, Computer Corporation of America, Cambridge, Mass., February 15, 1977.

[9] Rothnie, J.B. and N. Goodman, "An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases," Proc. of the 2nd Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, Calif., May 25-27, 1977, pp. 39-57.

[10] Crocker, S.D., J.F. Heafner, R.M. Metcalfe and J.B. Postel, "Function-oriented Protocols for the ARPA Computer Network," Proc. of the 1972 SJCC, AFIPS, 40, 1972, pp. 271-279.

[11] Chang, E., "A Distributed Medical Data Base, Network Software Design," Computer Networks, 1, 1, June, 1976, pp. 33-38.

[12] CODASYL Data Base Task Group Report, April, 1971, ACM, New York.

[13] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," Comm. ACM, 13, 6, June, 1970, pp. 377-387.

[14] IBM, Information Management System/Virtual Storage (IMS/VS), Version 1, General Information Manual, GH 20-1260-4.

[15] IBM, Information Management System/Virtual Storage (IMS/VS), Version 1, Application Programming Reference Manual, SH 20-9026-4.

[16] IBM, Information Management System/Virtual Storage (IMS/VS), Version 1, System Programming Reference Manual, SH 20-9027-4.

[17] Hsiao, D.K., D.S. Kerr, and F.K. Ng, "DBC Software Requirements for Supporting Hierarchical Databases," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-77-1, April, 1977.

[18] Banerjee, J., D.K. Hsiao, and D.S. Kerr, "DBC Software Requirements for Supporting Network Database," The Ohio State University,Tech. Rep. No. OSU-CISRC-TR-77-4, June, 1977.

PART II

DBC's CAPABILITY IN SUPPORTING EXISTING DATABASE MANAGEMENT APPLICATION

DATABASE TRANSFORMATION, QUERY TRANSLATION AND PERFORMANCE ANALYSIS OF A NEW DATABASE COMPUTER IN SUPPORTING HIERARCHICAL DATABASE MANAGEMENT

JAYANTA BANERJEE AND DAVID K. HSIAO
THE OHIO STATE UNIVERSITY

FRED K. NG
BELL LABORATORIES

Database computers are special-purpose storage and processing devices which are intended to relieve the database management (software) systems running on the general-purpose computers and provide improved storage and processing capabilities (via hardware) for the existing and new database application. However, to support existing database applications, two steps must be followed. First, the existing database must be transformed into the storage format of the new database computer. This one-time transformation, known as database transformation, is required to preserve the semantics of the database and to take advantage of the advanced hardware features of the new computer. Second, the database sublanguage used in the existing application programs must be supported in real-time by the new database computer so that application programs may be executed in the new environment without the need of program conversion. Such real-time translation of sub-language calls to the instructions of the new database computer, known as query translation, must be straightforward with minimal software support.

In this paper, we present a methodology to carry out database transformation and query translation for a prevailing type of database management, known as hierarchical database applications. In addition, we present an analysis of the storage requirement of the transformed database and transaction execution time of the translated sub-language calls. Although the new database computer's storage requirement is comparable to that of a conventional implementation, the execution time of typical transactions in the new database computer is considerably better -- a factor of one or two magnitudes of improvement is possible.

It is hoped that both the methodology and the analysis can be employed not only for a comparative performance study of hierarchical database management on conventional general-purpose computers vs. new special-purpose devices, but also, with some extension, for study of other types of database management, such as CODASYL and relational, on the new database computers.

1. MACHINE ELEMENTS

Although this paper deals with the capability of a database computer, known as DBC [1,2], in supporting existing hierarchical database management application, there are some basic features of DBC which can also be found in many proposed and experimental database machines [3]. These features play an important role in the study of the storage requirement and performance gain of the new computer. Let us introduce them briefly.

1.1 Hardware Capabilities of DBC

For the on-line database store, DBC utilizes modified moving-head disks. This modification includes the capability for tracks-in-parallel readout and content-addressability. By tracks-in-parallel readout [4] it is meant that all the tracks (usually 20 of them) of a cylinder can be read in one revolution, once the access mechanism is positioned at the cylinder. Content-addressability is achieved by local processing units (usually 20 of them) which accept database management instruction (say, a search-retrieve instruction), compare the instruction parameters (in the form of predicates) with the incoming data streams from the tracks and output the answer. The key concepts of the content-addressability of the database store are that the processing is done locally at the disk controller and that search, retrieval and update of records are based on predicates (a more complete discussion is given in Section 1.2). A simple expression of three predicates, namely, an equality predicate, an inequality predicate, and a less-than predicate is depicted below:

(Name = HSIAO) AND (Location ≠ MICHIGAN) AND (Salary < 50000)

The above expression includes the logical AND of attribute values where values are either capitalized or numerals, and relational operators precede the values and follow the attributes.

More complex instruction parameters involving both logical AND (i.e., conjunction of attribute values) and logical OR (disjunction of attribute values) are possible.

The database store, known as the mass memory, is of a single conjunction and multiple data stream (SCMD) architecture. Records from the tracks of a cylinder form separate data streams which are content-addressed by the corresponding processing units in a bit-by-bit serial fashion against the same conjunction in one disk revolution time. This is depicted in Figure 1.

To content-address the entire cylinder space of the database is not only unnecessary but also undesirable, since the disk access mechanism moves slowly. In order to confine the content-addressing into those few cylinders which may have records satisfying the conjunction, the database store is aided with separate components of DBC known as the structure memory and structure memory information processor. For an attribute value

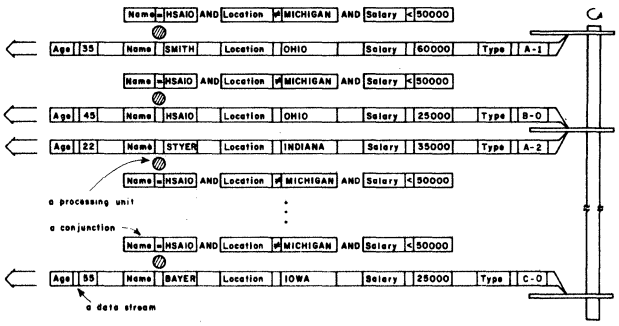


FIGURE 1. A Single Conjunction and Multiple Data Stream Architecture (SCMD)

(or attribute value range) the structure memory maintains the cylinder numbers of the cylinders whose records contain the attribute value. The structure memory entries are termed indices. When several attribute values are involved in a conjunction, the indices found in the structure memory for the conjunction must be intersected by the structure memory information processor. Thus, only cylinder numbers of the cylinders whose records contain all the attribute values of the conjunction will be given to the database store for content-addressing. In this way, the number of cylinders involved in content-addressing will be drastically reduced. The structure memory is made of either charge-coupled devices (CCDs) or bubble-domain memories (BDMs). The structure memory, like the mass memory, is content-addressable, but has much smaller capacity (1% of mass memory size) and higher speed (20 times faster block access rate).

The DBC control processor is in charge of interfacing with the front-end host computer. Given an instruction from the front-end, the DBC control processor parses the instruction parameter, determines (by activating the structure memory and structure memory information processor) the cylinders to be searched in order to satisfy the parameter, issues appropriate orders to the mass memory, and transfers response data back to the front-end system. The control processor also coordinates other activities of DBC, including data clustering. An overview of these components of DBC is illustrated in Figure 2. We note that the application programs are still resident in a front-end general-purpose host computer. The operating system of the host computer is relieved of data management tasks, since the database is handled by the back-end DBC. Data management calls are now directed by the operating system to the database interface (DBI), which is a small software package keeping account of these calls, translating them into DBC instructions and forwarding the instructions to DBC for execution. DBC also performs some post processing of relevant data for security and sequencing purposes.

1.2 Data Structures

Data is stored and manipulated in DBC as collections of records. Each record can represent an entity of the physical world and a number of

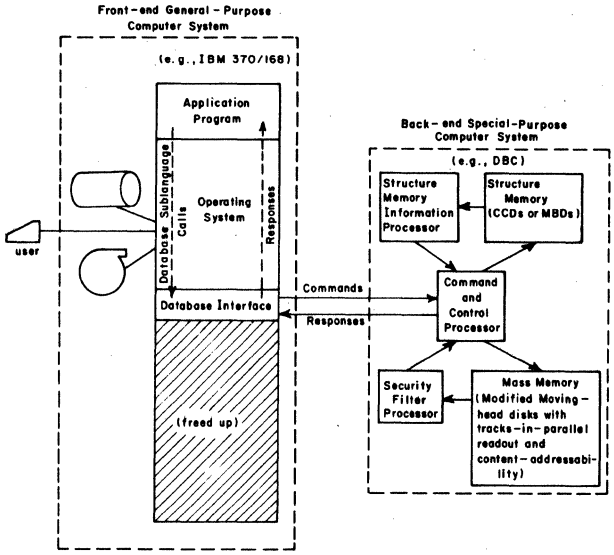


FIGURE 2. An Overview of the New Database Environment

records can be grouped into files on the basis of ownership, security and proximity. A record consists of a record body and a set of variable-length attribute values, where the attribute may represent the type, quality or characteristic of the value. The record body is composed of a (possibly empty) string of characters which are not used for content-addressing purposes by the mass memory. For logical reasons, all the attributes in a record are required to be distinct. An example of a record is shown below:

(<Type, EMP>, <Job, MGR>, <Dept, TOY>).

The record consists of three attribute values. The value of the attribute Job, for instance, is MGR. Attribute values are also called keywords which characterize records and may be used as 'keys' in content-addressing. Keywords for which directory entries are maintained in the structure memory are called type-D keywords.

DBC interfaces with front-end host systems by accepting a large repertoire of high-level database management instructions, by delivering collections of records as response sets, and by indicating successful or unsuccessful execution of the instructions. Some of the instructions, called record access commands, may be used for specifying a collection of records in the database and for carrying out an intended operation on these records, such as retrieval, deletion and modification. Other instructions may be used for database loading, record insertion, initialization, etc.

An important feature of record access commands is that they allow natural expressions for specifying a record collection. A record collection may be specified in terms of a keyword predicate, or simply, a predicate, which is a triple consisting of an attribute, a relational operator (such as, =, ≠, >, ≥, <, ≤) and a value. For instance, (Salary > 10000) is a predicate. A

record collection may also be specified in terms of a conjunction of predicates, called a query conjunction. Finally, a record collection may be specified in terms of a disjunction of query conjunctions, called a query.

Certain attributes of a file may be designated by the file creator as clustering attributes. Correspondingly, keywords having clustering attributes are called clustering keywords. If a query makes use of clustering attributes, the query can be satisfied in as few disk revolutions as there are query conjunctions within the query.

2. A HIERARCHICAL DATABASE MANAGEMENT SYSTEM (IMS)

In order to study our approach more specifically, we shall concentrate on a typical hierarchical database management system, the Information Management System (IMS) [5-8], which is perhaps one of the most widely used. In this section, we shall provide a very brief summary of the basic features of IMS.

An IMS database consists of a number of hierarchically related segment occurrences (or simply, segments), each of which belongs to a segment type. In the example of Figure 3, segment type A, the root segment type, has three occurrences. All others are dependent segment types, each having a unique parent segment type and zero or more child segment types. Some relationships among the various segments in our example are:

A1 is the parent of B1 and G1.
H1, H2, I1 are children of G1.
J1 and J2 are twins.
H1, H2, I1, J1, J2 are descendants or dependents of G1.

Successive levels are numbered such that a root segment is at level 1. All segment occurrences are made of one or more fields.

An IMS database is traversed in the order parent to child, front to back smong twins and left to right among children. The traversal sequence for the database of Figure 3 is (A1, B1, C1, D1, D2, D3, E1, F1, E2, F2, F3, G1, H1, H2, I1, J1, J2, A2, A3). Notice that the traversal sequence defines a next segment with respect to a given segment. A hierarchical path is a sequence of segments, one per level, starting at the root, e.g., (A1, G1, I1, J2).

An IMS user processes an IMS database with application programs using Data Language/1 (DL/1). A DL/1 call has the following format:

FUNCTION search-list

where FUNCTION is one of insert (ISRT), delete (DLET), replace (REPL) and get (GET) calls, and where search-list is a sequence of segment search arguments (SSAs), at most one per level, which are used to select a hierarchical path. Each segment search argument is of the form

<segment-type><Boolean expression>

with Boolean expression relating values of fields of given segment type.

After each retrieval or insertion operation, a segment is "established" in the traversal sequence of the IMS database. For a retrieval operation, this segment refers to the segment just retrieved; for an insertion operation, this segment refers to the segment just inserted. Such a segment in the traversal sequence is termed the current position in the database. The hierarchical path leading from the root segment to the current position in the database consists of many segments. Each of these segments is called the segment on which position is established at that level.

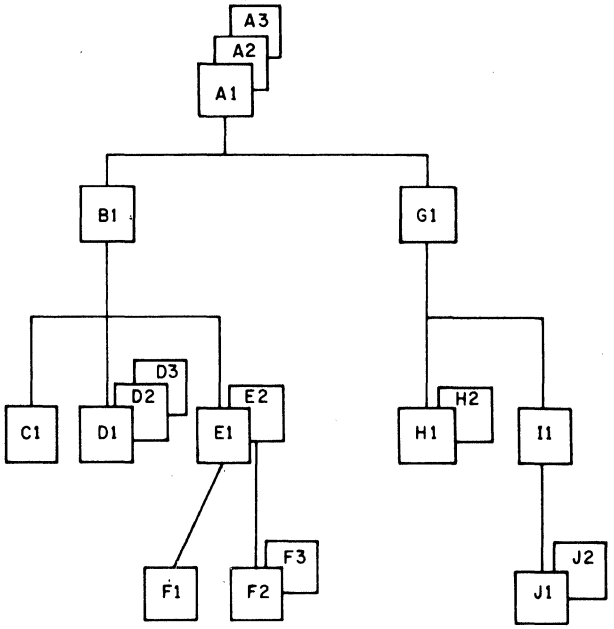


FIGURE 3. Schematic Representation of an IMS Database

There are several forms of the get call, each of which returns a single segment. A get-unique (GU) call retrieves a specific segment at level n by starting at the root segment type, finding the first segment at each level i satisfying SSA_i, and finally retrieving the segment satisfying the last SSA. A get-next (GN) call starts the search at the current position in the database and proceeds along the traversal sequence satisfying the SSAs and retrieving the segment satisfying the last SSA. A get-next-within-parent (GNP) call restricts the search to descendants of a specific parent segment. Thus IMS also maintains a parent position which is set at the last segment that was retrieved by a GU or GN call. The parent position remains constant for successive GNP calls.

3. DATABASE TRANSFORMATION

An existing database may be supported on DBC by converting the database to conform to the DBC data structure. This one-time conversion is known as database transformation. Existing database management applications need not be reprogrammed. Instead, the database interface (DBI) software residing in the front-end host computer will translate in real-time the data management calls into DBC instructions. This process is known as query translation and will be the subject of our discussion in the next section.

An IMS database can be structured by considering every IMS segment as a DBC record (or, simply, a record) composed of keywords. Address-dependent pointers of the segments are replaced in the records by keywords that are not dependent on physical location.

3.1 The Notion of Symbolic Identifier

An IMS segment includes a sequence field whenever it is necessary to indicate the order among the twin segments. Since each segment becomes a record and no address-dependent pointers are allowed, we assign a symbolic indentifier to each segment, identifying it uniquely from all other segments in the database. The symbolic identifier of a segment S is a group of fields consisting of (1) the symbolic identifier of the parent of S, and (2) the sequence field of S. Since the sequence fields of different segment types may use the same field name, we may qualify the field name with the segment type.

3.2 The Conversion of IMS Segments

The creation of a record from an IMS segment can now be accomplished by forming keywords as follows:

(1) For each field in the segment, form a keyword using the field name as the attribute and the field value as the value.
(2) Form a keyword of the form <Type, SEGTYPE> where Type is a literal and SEGTYPE is the segment type in consideration.
(3) For each sequence field in the symbolic identifier of the segment, form a keyword using the field name as the attribute and the field value as the value.

For example, for an IMS database shown in Figure 4, the attribute templates of the five collections of records corresponding to the five segment types are shown in Figure 5. Qualified field names such as Prereq.Course# are used to distinguish the same field names, i.e., Course#, among different segment types.

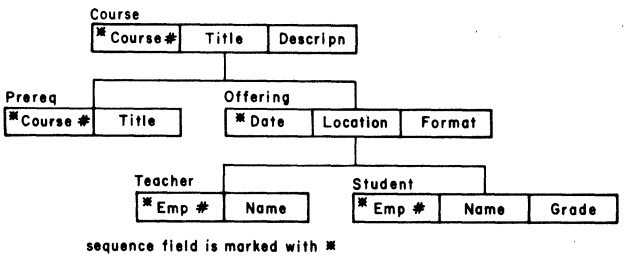


FIGURE 4. The Logical Data Structure of an IMS Database

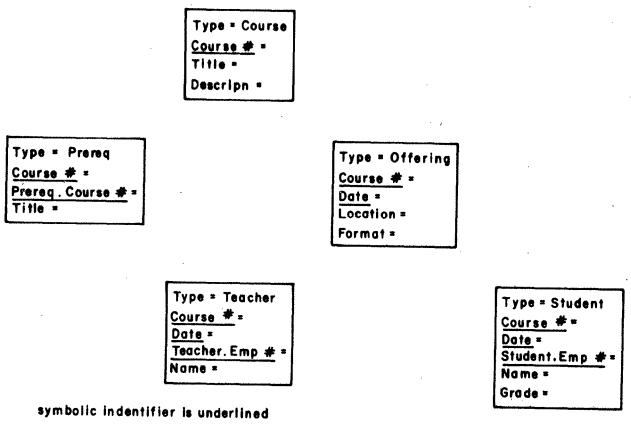


FIGURE 5. The Attribute Templates of DBC Records for the Segments of Figure 4

3.3 The Clustering of the New Database

The access pattern of the segments should be used to determine the clustering policy in DBC. Since the traversal of an IMS database is usually along a hierarchical path, one clustering policy is to first cluster the records which represent all the IMS root segments and then cluster the records which represent all dependent segments. An application of this policy is illustrated in Figure 6. An advantage of using this policy is that if several root segments are to be accessed collectively, a single cylinder access will retrieve them all. Furthermore, since for a given root segment the average size of all the segments in a hierarchical path is usually smaller than the size of a cylinder, it is possible to cluster all the dependent segments of the root segment in the same cylinder.

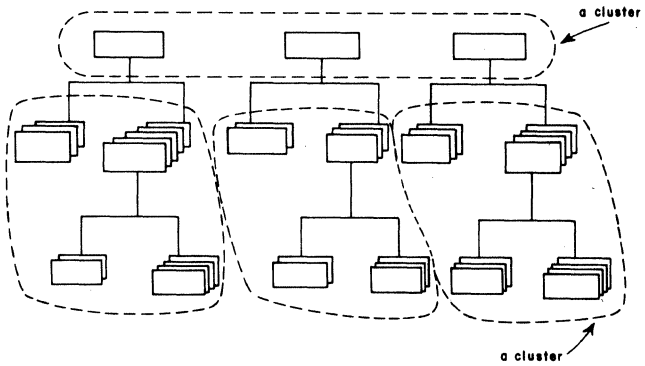


FIGURE 6. An Application of the Clustering Policy

For the clustering policy, an IMS database is therefore created on DBC with two kinds of clusters, one cluster containing only the converted root segments and the other kind of cluster containing the rest of the converted segments. The only clustering attribute for the first cluster is Type. This assures that all the converted root segments

form a single cluster, since they all have the same clustering keyword. If the sequence field of the root segment is called Seq, then the only clustering attribute for the second kind of cluster is Seq. Thus, there are as many clusters as there are unique sequence field values in the root segments (see Figure 6 again). The clustering keywords also constitute the only type-D keywords.

Since there is a one-to-one correspondence between an IMS segment and its converted form, i.e., an equivalent DBC record, we shall refer to them in the sequel without confusion with either terminology.

4. QUERY TRANSLATION

The database interface (DBI), residing in the front-end host computer, translates DL/1 calls into DBC instructions and maintains a buffer, called the interface system buffer (ISB) (or, simply, buffer), for storing information retrieved from the database. Unlike the IMS where DL/1 is allowed to specify only one segment at a time for retrieval and update and to select the segment by traversing many intermediate segments which precede it, DBC can content-address a group of records that satisfy a given condition and retrieve the records at the same time. At any moment, the buffer may contain the segments which form the hierarchical path to the current position of the database.

4.1 Illustrating the DBC Execution of DL/1 Calls

We shall illustrate how the information in the buffer is maintained and used by showing the manner in which get-unique (GU) and get-next(GN) calls are executed by the database interface (DBI). Referring back to the IMS database of Figure 3, suppose the DL/1 call to be processed is:

```
GU   Course     (Title = 'MATH')
     Offering   (Location = 'STOCKHOLM')
     Student    (Grade = 'A')
```

This call asks for the first Student segment of the database which satisfies the predicate Grade = 'A', and has a parent segment Offering (with Location = 'STOCKHOLM'), whose parent, in turn, is a Course segment (with Title = 'MATH'). The call is processed by DBI and executed by DBC as follows:

(1) Starting with the first segment search argument (SSA_1) in the call, i.e., Course (Title = 'MATH'), the Course segments which satisfy the predicate Title = 'MATH' are retrieved from DBC and placed in the buffer. These segments are retrieved by DBC using the query formulated by DBI ((Type = COURSE) ∧ (Title = MATH)) and are sorted by DBC according to the values of their sequence field, i.e., by the attribute Course#.

(2) If no Course segment exists in the buffer, then the DL/1 call is unsuccessful. Otherwise, the first Course segment in the buffer is designated as the current Course segment.

(3) The Offering segments are then retrieved with the predicate Location = 'STOCKHOLM' and stored in the buffer, sorted by their sequence field, i.e., by the attribute Date. If the sequence field of the current Course segment is (Course#, C), then the DBC query used for this retrieval is ((Type = OFFERING) ∧ (Course# = C) ∧ (Location = STOCKHOLM)).

(4) If no Offering segment exists in the buffer, then the current Course segment is removed from the buffer and control is transferred to Step 2. Otherwise, the first Offering segment in the buffer is designated as the current Offering segment.

(5) The Student segments are then retrieved with the predicate Grade = 'A' and stored in the buffer, sorted by their sequence field, i.e., by the attribute Emp#. If the sequence field of the current Course segment is (Course#, C) and that of the current Offering segment is (Date, D), then the DBC query used for this retrieval is ((Type = STUDENT) ∧ (Course# = C) ∧ (Date = D) ∧ (Grade = A)).

(6) If no Student segment exists in the buffer, then the current Offering segment is removed from the buffer and control is transferred to Step 4. Otherwise, the first Student segment in the buffer is designated as the current Student segment.

(7) The DL/1 call is successful and the current Student segment is returned.

The content of the buffer at the end of execution of the DL/1 call may look like the one shown in Figure 7. It should be noted at this point that the content of the buffer established by the above GU call may be used to process the

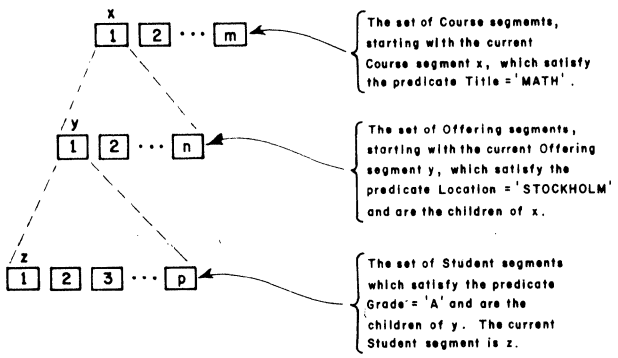


FIGURE 7. The Content of the Interface System Buffer (ISB)

next DL/1 call, for example, to retrieve the next student who has an A grade in a mathematics course offered in Stockholm. This is illustrated by the following get-next (GN) call:

```
GN   Course     (Title = 'MATH')
     Offering   (Location = 'STOCKHOLM')
     Student    (Grade = 'A')
```

In this case, the relevant segment may already be present in the buffer. The current Student segment is removed from the buffer and control is transferred to Step 6 of the procedure given for the GU call.

On the other hand, if the GN call is

```
GN   Course     (Title = 'MATH')
     Offering   (Location = 'STOCKHOLM')
     Student    (Grade = 'B')
```

then only a portion of the buffer information may be used, namely, the existing Course and Offering segments only. However, it is necessary that the

next Student segment returned should not precede the current Student segment in the traversal sequence. Hence, if the sequence field of the current Student segment is (Emp#, E), that of the current Offering segment is (Date, D) and that of the current Course segment is (Course#, C), then the following DBC query is used for retrieval of the next set of Student segments:

((Type = STUDENT) ∧ (Course# = C) ∧ (Date = D) ∧ (Emp# ≥ E) ∧ (Grade = B))

The previously existing Student segments of the buffer are removed, and control is transferred to Step 6 of the procedure given for the GU call.

Finally, if the GN call is

GN Course (Title = 'HISTORY')
Offering
Student

then no currently existing segments of the buffer are useful. Hence, new sets of segments must be retrieved, one set for each level.

4.2 Data Structures Used for the Execution of DL/1 Calls

The information in the interface system buffer (ISB) is managed by making use of two tables: the status information table (SIT) and the hierarchy table (HT). The two tables also maintain all 'currency' information about the database, so that DL/1 calls may be properly executed.

The status information table (SIT) has as many entries at any moment as there are levels in the hierarchical path from the root segment to the current position of the database. The first entry corresponds to the first level (i.e., root segment level), the second entry corresponds to the second level, and so on. Each entry in SIT consists of four fields: Seg, Count, Addr, and Qual. The meaning of the i-th entry of SIT is as follows:

SIT.Seg(i)	(coded) name of the segment type of the i-th level of the hierarchical path (including the current position of the database);
SIT.Count(i)	number of segments in buffer of the above segment type;
SIT.Addr(i)	address in buffer of the first of the above segments;
SIT.Qual(i)	the segment search argument (SSA_i) corresponding to the above segment type.

The hierarchy table (HT) has the same number of entries as there are in SIT. Each entry in HT consists of two fields: F and V. The meaning of the i-th entry of HT is as follows:

HT.F(i)	sequence field name of the current position (segment) in level i.
HT.V(i)	sequence field value of the current position (segment) in level i.

Even though the names and values of the above sequence fields are also available in the segments occupying the interface system buffer, it is convenient to have them together in the form of a single table.

4.3 Algorithms for the Execution of DL/1 Calls

A DL/1 call, as we have noted earlier, has the following format:

FUNCTION s_1 q_1
s_2 q_2
⋮
s_n q_n

where each s_i is a (coded) segment-type name. All segment types are assumed to be coded such that the code for a segment type A is less than the code for a segment type B, if A precedes B in the IMS traversal sequence. Each q_i is a Boolean expression of predicates, possibly null. To demonstrate how DL/1 functions are handled in DBC, we shall provide the algorithms used for executing the two most important functions, namely, get-unique (GU) and get-next (GN).

First, let us consider Algorithm GU, which translates GU calls and causes DBC to execute the equivalent instructions. In referring to Figure 8, we note that root segments satisfying q_1 are retrieved in the first step. Segments in all other

Algorithm GU

This algorithm executes the following DL/1 call:

GU s_1 q_1
s_2 q_2
⋮
s_n q_n

where each s_i is a segment type in level i, each q_i is a qualification (possibly null) and $n \geq 1$. We assume that the sequence field name of segment type s_i is f_i.

Step 1. (Retrieve root segments into buffer and update SIT, HT)
i ← 1
retrieve(((Type = s_1) ∧ q_1), sort attribute f_1, buffer address a, count c)
if c = o then return ('failure', -)
SIT(1) ← (s_1, c, a, q_1)
let (f_1, v_1) be the sequence field of the segment in address a
HT(1) ← (f_1, v_1)

Step 2. (All segments retrieved?)
i ← i + 1
if i > n then go to Step 6

Step 3. (Retrieve segments at i-th level)
retrieve(((Type = s_i) ∧ (f_1 = v_1) ∧....∧ (f_{i-1} = v_{i-1}) ∧ q_i), sort attribute f_i, buffer address a, count c)
if c ≠ o then go to Step 5

Step 4. (Retract one level and try again)
i ← i - 1
if i = o then return ('failure', -)
(s_i, c, a, q_i) ← SIT(i)
c ← c - 1
if c = o then go to Step 4
reset(buffer address a)

Step 5. (update SIT, HT)
SIT(i) ← (s_i, c, a, q_i)
let (f_i, v_i) be the sequence field of the segment in address a
HT(i) ← (f_i, v_i)
go to Step 2

Step 6. (Operation successful)
number of entries in SIT or HT ← n
current position of database ← n
parent position ← n
return ('success', buffer address a)

FIGURE 8. The Algorithm GU

levels are retrieved in Step 3. The preferred treatment accorded to the root segments is due to the fact that they belong to a cluster different from all other segment types. If no segment is retrieved in some level i, then another segment in the previous level (i-1) has to be considered. This is done in Step 4.

In general, Algorithm GU returns (to the application program) a status of 'failure' if no segment is found that satisfies the GU call. If the call is successfully executed, then the buffer address of the required segment is returned. The status information table (SIT) and the hierarchy table (HT) are used in the execution of the DL/1 call.

In Algorithm GU, two procedures are used: retrieve and reset. The retrieve procedure has four parameters: a DBC query Q, a sort attribute f, a buffer address and a DBC record count. The procedure sends a search-retrieve (sr) command to DBC for the purpose of content-addressing all segments (i.e., DBC records) that satisfy the query Q. DBC is also asked to sort these segments by attribute f before transmitting them to the data-base interface (DBI). This is accomplished by the Security Filter Processor (see Figure 2 again). The sorted segments are stored in the buffer. Finally, the retrieve procedure returns the address of the first of the sorted segments and the count of these segments to the calling algorithm.

The reset procedure, given a buffer address, removes from the buffer the segment stored in that address, thereby releasing the space occupied by the segment. The procedure also assigns to the address variable the address of the segment (if any) that is next to the removed segment. Thus, the address variable now refers to the first of the remaining segments (in the buffer) of the same type.

Algorithm GN translates GN calls and causes DBC to retrieve segments only if necessary. Most of the time, the required segment may already be available in the buffer, since DBC tends to retrieve segments in bulk. Referring to Steps 1 through 3 in Figure 9, a number t is determined such that s_i and q_i are the same as those of the previous DL/1 call, for $1 \leq i \leq t$. However, s_{t+1} or q_{t+1} are not the same as those of the previous call. Thus, all segments in the buffer that correspond to level (t+1) or beyond (up to level m, which is the number of entries in SIT or HT) have no further use. Space occupied by those segments (if any) are released by the procedure clear-buffer. In case t = 0, then the entire search-list is different from the previous one. Other conditions are checked in Steps 5 through 9. Steps 10 through 16 are almost identical to the steps in Algorithm GU.

4.4 A Case of Optimization

It may be noted that in executing certain types of transactions, a number of unnecessary accesses may be avoided, if we further optimize Algorithms GU and GN. Consider, for example, the following transaction which retrieves all student records obtaining grade A in history:

Algorithm GN

This algorithm executes the following DL/1 call:

GN s_1 q_1
s_2 q_2
⋮
s_n q_n

where each s_i is a (coded) segment type in level i, each q_i is a qualification (possibly null) and $n \geq 1$. We assume that the code for a segment name A is less than the code for a segment name B if A precedes B in the traversal sequence. m is the number of entries in SIT or HT.

Step 1. (Find t such that the condition ((s_i = SIT.Seg(i)) ∧ (q_i = SIT.Qual(i))) is satisfied for $1 \leq i \leq t$ but not for i = t + 1)
t ← 0

Step 2. t ← t + 1
if t > n or t > m then go to Step 3
(f_t, v_t) ← HT(t)
if s_t = SIT.Seg(t) and q_t = SIT.Qual(t) then go to Step 2

Step 3. t ← t - 1
clear-buffer (t + 1, m)

Step 4. (No buffer information is useful?)
if t = 0 then go to Step 10

Step 5. (Perhaps the necessary segment is in the buffer? $1 \leq t$, but is $t = n \leq m$?)
if t = n then set i ← t + 1 and go to Step 14

Step 6. (Entire buffer information is useful? $1 \leq t$ and $m < n$, but is t = m?)
if t = m then set i ← t + 1 and go to Step 12

Step 7. ($1 \leq t < m < n$)
if s_{t+1} < SIT.Seg(t+1) then return ('failure', -)

Step 8. if s_{t+1} > SIT.Seg(t+1) then set i ← t + 1 and go to Step 12

Step 9. ($1 \leq t < m < n$, s_{t+1} = SIT.Seg(t+1), q_{t+1} ≠ SIT.Qual(t+1))
i ← t + 1
retrieve (((Type = s_i) ∧ (f_1 = v_1) ∧....∧ (f_{i-1} = v_{i-1}) ∧ ($f_i \geq v_i$) ∧ q_i), sort attribute f_i, buffer address a, count c)
go to Step 13

Step 10. (Retrieve root segments into buffer and update SIT, HT)
retrieve (((Type = s_1) ∧ ($f_1 \geq v_1$) ∧ q_1), sort attribute f_1, buffer address a, count c)
if c = o then return ('failure', -)
SIT(1) ← (s_1, c, a, q_1)
let (f_1, v_1) be the sequence field of the segment in address a
HT(1) ← (f_1, v_1)

Step 11. (All segments retrieved?)
i ← i + 1
if i > n then go to Step 16

Step 12. (Retrieve segments at i-th level)
retrieve (((Type = s_i) ∧ (f_1 = v_1) ∧....∧ (f_{i-1} = v_{i-1}) ∧ q_i), sort attribute f_i, buffer address a, count c)

Step 13. if c ≠ o then go to Step 15

Step 14. (Retract one level and try again)
i ← i - 1
if i = o then return ('failure', -)
(s_i, c, a, q_i) ← SIT(i)
c ← c - 1
if c = o then go to Step 14
reset (buffer address a)

Step 15. (Update SIT, HT)
SIT(i) ← (s_i, c, a, q_i)
let (f_i, v_i) be the sequence field of the segment in address a
HT(i) ← (f_i, v_i)
go to Step 11

Step 16. (Operation successful)
number of entries in SIT or HT ← n
current position of database ← n
parent position ← n
return ('success', buffer address a)

FIGURE 9. The Algorithm GN

```
GU    Course     (Title = 'HISTORY')
      Offering
      Student    (Grade = 'A')

Loop  GN Course  (Title = 'HISTORY')
      Offering
      Student    (Grade = 'A')
      Go to Loop
```

In the method previously outlined, one database access is necessary to retrieve all root segments satisfying Title = 'HISTORY' (assuming that the

cluster of root segments occupies a single cylinder). Let us assume that n_1 root segments are retrieved. For each root segment, an access is made to retrieve all its children of type Offering (assuming that the cluster of the dependent Offering segments of a root segment is no larger than a cylinder). If there are n_2 Offering segments per root segment, then a total of n_1 n_2 accesses are required for the Student segments. Thus, the total number of accesses will be $(1 + n_1 + n_1 n_2)$. On the other hand, if we retrieve all root segments satisfying Title = 'HISTORY', and then retrieve all Student segments that are descendants of these root segments by using DBC queries in the form:

((Type = STUDENT) ∧ (Course# = x) ∧ (Grade = A)

where x is a sequence field value of a root segment, then the total number of accesses will be only $(1 + n_1)$. This is a large saving in accesses, due to the fact that one or more intermediate segment types (in this case, Offering) have no predicate and, therefore, need not be accessed.

Thus, in case there are no expressions associated with intermediate segment types (i.e., excluding the root segment type and the target segment type), these segment types need not be accessed. Modified versions of Algorithms GU and GN may be written to achieve the above optimization.

The algorithms for the execution of all other DL/1 functions have the same underlying philosophy as that of Algorithms GU and GN. Algorithms for the GNP, ISRT, DLET, REPL and the various get-hold calls are, therefore, not discussed in this paper.

5. PERFORMANCE ANALYSIS

An IMS database supported on DBC (in the manner indicated in Sections 3 and 4) can be shown to perform appreciably better than the same database supported on a conventional general-purpose computer. The performance may be measured in terms of the storage needed to accommodate the database and the time required to execute typical transactions. The enhancement in performance is mainly due to the content-addressability and tracks-in-parallel read-out mechanism of the DBC database store, i.e., the mass memory (MM). For example, while a conventional computer may require many database accesses to locate, via address-dependent pointers, the segments satisfying a given DL/1 transaction, MM has the capability of content-searching the database, thereby locating the segments and transferring them simultaneously to the front-end computer. Other gains are possible. Due to the very large size of the content-addressable blocks of the mass memory, the indices kept on the structure memory (SM) are fewer than the ones kept in the directory of a conventional system, since the former contains only cylinder numbers and the latter must include also track, page and record numbers. In addition, index processing in DBC is done at the structure memory information processor (SMIP) which does not involve the host computer. In the conventional environment, the indices must be brought from the secondary storage to the main computer for processing. The concurrency achieved in DBC, where index processing is overlapped with database processing as depicted in Figure 2, is difficult to realize in a conventional environment.

In this section, we shall present an analysis of the performance of DBC as compared to that of a conventional computer in supporting hierarchical (e.g., IMS) databases. We shall call the environment consisting of DBC and a front-end host computer the DBC environment. The environment consisting of a general-purpose computer acting alone with a database management system will be called the GPC environment.

Because of the enormous number of factors involved in the analysis of a database management system, it is necessary to reduce the number of parameters to a manageable few. The reduction, however, must not distort the characteristics of the real systems. In this study, the following set of parameters stands for the average values of the variables in a real system:

- N = total number of IMS root segments in an IMS database;
- L = depth of an IMS database, i.e., the average number of levels;
- m = average fanout or number of child segment types of a segment type;
- y = average number of twin occurrences of a child segment type;
- p = length of an address pointer in the GPC environment;
- k = average length of a DBC keyword or a field (including sequence field);
- d = average number of fields (including the sequence field) in each segment.

In our analysis we shall also use the following variables and notations:

- r = ratio between the keyword length and address pointer length, i.e., k/p;
- n = number of segments satisfying certain DL/1 transactions;
- M_d = DBC database storage requirement;
- M_g = GPC database storage requirement;
- $R_m = M_g/M_d$, called the database storage ratio;
- T_d = transaction execution time in DBC environment;
- T_g = transaction execution time in GPC environment;
- $R_t = T_g/T_d$, called the transaction execution time ratio.

5.1 Storage Analysis

Storage is required for the indices, the database definition, the buffers, the database management system software and the database store. If we ignore all secondary indices that may be maintained in the GPC environment, then the index storage requirements are about the same in the two environments. In fact, the primary index maintained in the GPC environment has almost the same number of entries as there are type-D keywords in the DBC environment. There is only one type-D keyword, namely, (Type, ROOT-SEGTYPE), for the root segments. If there are t unique sequence field values in the set of all root segments, then there are only t type-D keywords for the rest of

the segments. Correspondingly, in the GPC environment, there are t entries in the primary index.

The database definition is independent of the computer on which the database management is provided. The storage requirements for the database definition, in the two cases, therefore, are identical.

In the GPC environment, there is an input/output buffer for the storage of a few pages of segments. In the DBC environment, on the other hand, the interface system buffer (ISB) is used for accommodating segments whose contents are known and, therefore, can be used for future DL/1 calls. There should be enough buffer space in the ISB to accommodate a set of twin segments from each level of the database. This can occasionally amount to a few hundred segments. However, the main memory requirement of ISB for such a large number of segments is compensated by the freed-up memory due to the removal of the conventional database management system software (e.g., IMS).

Furthermore, the main memory requirements for the database management system software are not difficult to estimate. It is possible to find out the size of the software for any given IMS implementation in a GPC environment. On the other hand, the database interface (DBI), whose algorithms are depicted in Figures 8 and 9, is expected to have a considerably smaller size than the conventional IMS software.

By far the largest investment in storage is the storage for the database. We shall, therefore, make a relatively thorough analysis of the database storage requirement.

A. Database Storage Requirement in DBC Environment

The use of symbolic identifiers increases the storage required to represent an IMS segment as a DBC record. At each level of a hierarchical data structure, the number of additional keywords stored in a DBC record equals the number of keywords in the symbolic identifier of the parent, i.e., zero at the root level, one at the second level, and (i-1) at the i-th level. A DBC record will require some more space for each of the d fields. (We are using the notation given in the beginning of Section 5.) Thus, a record at the i-th level will require ((i-1)k + dk) units of space, where k is the average length of a keyword or field. The total number of dependent segments at level i of a parent segment is $m^{i-1}y^{i-1}$, where m is the average fanout and y is the average number of twins. If in all there are L levels and N root segments, then the DBC database storage requirement M_d in the mass memory is shown to be:

$$M_d = N \sum_{i=1}^{L} m^{i-1}y^{i-1}\,((i-1)k+dk)$$

$$= \frac{Nk}{my} \sum_{i=1}^{L} i(my)^i + \frac{Nk}{my}(d-1) \sum_{i=1}^{L} (my)^i$$

$$= \frac{Nk}{my}\left(\frac{my-(my)^{L+1}}{(1-my)^2} - \frac{L(my)^{L+1}}{1-my}\right) + \frac{Nk}{my}(d-1)\frac{my-(my)^{L+1}}{1-my}$$

$$= \frac{Nk}{1-my}\left(\frac{1-(my)^L}{1-my} - L(my)^L + (d-1)(1-(my)^L)\right).$$

B. Database Storage Requirement in GPC Environment

To estimate the database storage requirement in the GPC environment, consider an IMS heirarchical database implementation, called the child/twin pointer representation. The child/twin pointer provides minimal traversal paths to an IMS database. Each segment has the following pointers to its "relatives":

(1) A pointer to the first child segment of each type.
(2) A pointer the last child segment of each type.
(3) A forward pointer the next twin.
(4) A backward pointer to the previous twin.
(5) A pointer to the parent segment.

If there are m child segment types of this segment, then there are (2m + 3) pointers. The space requirement for a single segment is, therefore, ((2m + 3)p + dk). Thus, the GPC database storage requirement M_g is as follows:

$$M_g = N \sum_{i=1}^{L} m^{i-1}y^{i-1}((2m + 3)p + dk)$$

$$= \frac{N}{my}(2mp + 3p + dk) \sum_{i=1}^{L} (my)^i$$

$$= \frac{N}{my}(2mp + 3p + dk)\frac{my-(my)^{L+1}}{1-my}$$

$$= \frac{N}{1-my}(2mp + 3p + dk)(1-(my)^L)$$

C. Database Storage Ratio

Let us first consider the ratio M_d/M_g,

$$\frac{M_d}{M_g} = \frac{k}{2mp + 3p + dk}\left(\frac{1}{1-my} - \frac{L(my)^L}{1-(my)^L} + (d-1)\right)$$

Since (my) >> 1 in most cases (i.e., the multiplicity of fanouts and twins), we approximate the above ratio as

$$\frac{M_d}{M_g} = \frac{k}{2mp + 3p + dk}(L + d - 1)$$

Finally, we have the database ratio R_m which is the inverse of the above approximation.

$$R_m = \frac{M_g}{M_d} = \frac{2mp + 3p + dk}{k(L + d - 1)}$$

We note incidentally that, in this ratio, the numerator is the length of a segment in GPC environment and the denominator is the length of a DBC record corresponding to an external (terminal or leaf) segment. This points to the fact that there are as many DBC records as there are IMS segments, and that the external segments being numerous play a more important role in storage

estimation than the internal segments.

Substituting r for the ratio k/p, we may relate the database storage ratio to the ratio between the storage requirements for keywords and their replacement, i.e., the pointers.

$$R_m = \frac{2m + 3 + dr}{r(L + d - 1)}$$

In Figure 10, we have tabulated the values of R_m for typical values of L, m, r and d. Most IMS databases have few levels, hence L is varied from 2 to 6.

level L=2

m \ d	r=2					r=4					r=8				
	2	4	8	16	32	2	4	8	16	32	2	4	8	16	32
1	1.50	1.30	1.17	1.09	1.05	1.08	1.05	1.03	1.01	1.01	0.88	0.92	0.96	0.98	0.99
2	1.83	1.50	1.28	1.15	1.08	1.25	1.15	1.08	1.04	1.02	0.96	0.98	0.99	0.99	1.00
3	2.17	1.70	1.39	1.21	1.11	1.42	1.25	1.14	1.07	1.04	1.04	1.03	1.01	1.01	1.00
4	2.50	1.90	1.50	1.26	1.14	1.58	1.35	1.19	1.10	1.05	1.13	1.08	1.04	1.02	1.01
5	2.83	2.10	1.61	1.32	1.17	1.75	1.45	1.25	1.13	1.07	1.21	1.13	1.07	1.04	1.02

level L=3

m \ d	r=2					r=4					r=8				
	2	4	8	16	32	2	4	8	16	32	2	4	8	16	32
1	1.13	1.08	1.05	1.03	1.01	0.81	0.88	0.92	0.96	0.98	0.66	0.77	0.86	0.92	0.96
2	1.38	1.25	1.15	1.08	1.04	0.94	0.96	0.98	0.99	0.99	0.72	0.81	0.89	0.94	0.97
3	1.63	1.42	1.25	1.14	1.07	1.06	1.04	1.03	1.01	1.01	0.78	0.85	0.91	0.95	0.97
4	1.88	1.58	1.35	1.19	1.10	1.19	1.13	1.08	1.04	1.02	0.84	0.90	0.94	0.97	0.98
5	2.13	1.75	1.45	1.25	1.13	1.31	1.21	1.13	1.07	1.04	0.91	0.94	0.96	0.98	0.99

level L=4

m \ d	r=2					r=4					r=8				
	2	4	8	16	32	2	4	8	16	32	2	4	8	16	32
1	0.90	0.93	0.95	0.97	0.99	0.65	0.75	0.84	0.91	0.95	0.52	0.66	0.78	0.88	0.93
2	1.10	1.07	1.05	1.03	1.10	0.75	0.82	0.89	0.93	0.96	0.58	0.70	0.81	0.89	0.94
3	1.30	1.21	1.14	1.08	1.04	0.85	0.89	0.93	0.96	0.98	0.63	0.73	0.83	0.90	0.95
4	1.50	1.36	1.23	1.13	1.07	0.95	0.96	0.98	0.99	0.99	0.67	0.77	0.85	0.91	0.95
5	1.70	1.50	1.32	1.18	1.10	1.05	1.04	1.02	1.01	1.01	0.73	0.80	0.88	0.93	0.96

level L=5

m \ d	r=2					r=4					r=8				
	2	4	8	16	32	2	4	8	16	32	2	4	8	16	32
1	0.75	0.81	0.88	0.92	0.96	0.54	0.66	0.77	0.86	0.92	0.44	0.58	0.72	0.83	0.91
2	0.92	0.94	0.96	0.98	0.99	0.63	0.72	0.81	0.89	0.94	0.48	0.61	0.74	0.84	0.91
3	1.08	1.06	1.04	1.03	1.01	0.71	0.78	0.85	0.91	0.95	0.52	0.64	0.76	0.86	0.92
4	1.25	1.19	1.13	1.08	1.04	0.79	0.84	0.90	0.94	0.97	0.56	0.67	0.78	0.87	0.93
5	1.42	1.31	1.21	1.13	1.07	0.88	0.91	0.94	0.96	0.98	0.60	0.70	0.80	0.88	0.93

level L=6

m \ d	r=2					r=4					r=8				
	2	4	8	16	32	2	4	8	16	32	2	4	8	16	32
1	0.64	0.72	0.81	0.88	0.93	0.46	0.58	0.71	0.82	0.90	0.38	0.51	0.66	0.79	0.88
2	0.79	0.83	0.88	0.93	0.96	0.54	0.64	0.75	0.85	0.91	0.41	0.54	0.68	0.80	0.89
3	0.93	0.94	0.96	0.98	0.99	0.61	0.69	0.79	0.87	0.93	0.45	0.57	0.70	0.82	0.90
4	1.07	1.06	1.04	1.02	1.01	0.68	0.75	0.83	0.89	0.94	0.48	0.60	0.72	0.83	0.90
5	1.21	1.17	1.12	1.07	1.04	0.75	0.81	0.87	0.92	0.95	0.52	0.63	0.74	0.84	0.91

$$R_m = \frac{\text{GPC database storage requirement}}{\text{DBC database storage requirement}}$$

FIGURE 10. Database Storage Ratio (R_m)

The fanout m is varied from 1 to 5, the number d of fields per segment is varied from 2 to 32 in multiples of 2, and the ratio r of keyword (field) length to pointer length is varied from 2 to 8 in multiples of 2. It may be noted from Figure 10 that the storage ratio varies from 0.38 to 2.83. The larger the number of levels, the larger is the storage requirement in the DBC environment while the larger the fanout, the larger is the storage requirement in the GPC environment. In general, however, the storage ratio is reasonably close to 1, so that we may assume that, for a typical database, the storage requirements in the two environments are almost identical.

5.2 Time Analysis of Transaction Execution

DBC performance may be measured in terms of the execution time of tyipcal database transactions. The time analysis of transactions is more complex. It involves, first, the database structure and, then, the transaction itself. We shall, therefore, provide a reasonable, but not necessarily exhaustive, classification of transactions and use the parameterized logical database structures to complete our analysis.

A. Unit of Measure

In the GPC environment, transaction execution time consists of the processing time in CPU and time to access the index and database storage. Processing time in CPU is usually very small compared to either index storage or database storage access time, and therefore, may be ignored. Indices may be relatively smaller than the database and may be stored on a device (say, fixed-head disk) that is faster than the database store (moving-head disks). Furthermore, for most transactions, there may be fewer accesses to the index device than the accesses to the database. Hence, we shall ignore the accesses to index storage as well. This simplification can only exaggerate the performance of the conventional system and minimize the performance of DBC.

In the DBC environment, transaction execution time consists of the processing time in the front-end host computer, time for accessing the structure memory, structure memory information processor and mass memory, and time for sorting the twin segments. The front-end processing time is again overlooked. Because the accesses to the structure memory and structure memory information processor are overlapped with the accesses to the mass memory for different transactions and sorting is done by the fast security filter processor which also overlaps its operations with those of other components, the only time of significance is the access time to the mass memory for the database.

The transaction execution time, in either environment, therefore, may be measured in terms of the number of accesses to the database. It should be emphasized that one access to the DBC database is sufficient to content-search an entire disk cylinder, while one access to the GPC database is required to retrieve or store a single page. However, in order to be considerably partial to the GPC environment, we have proposed a more powerful page access in the final analysis.

B. Physical Data Organization

In our analysis, we assume that an IMS database is implemented in the GPC environment with the HIDAM (hierarchical indexed direct access method) or HDAM (hierarchical direct access method). They allow direct, rather than sequential access of data, thereby offering rapid access to any root segments. The DBC performance can thus be compared with, perhaps, the best possible GPC performance.

In either the HIDAM or the HDAM, the logical adjacency of segments that are descendants of a root segment is often reflected by physical adjacency as well. In fact, whenever possible, a root segment and all its descendants are stored

contiguously within a set of adjacent pages. Updates to the database may sometimes prevent this contiguity, thereby forcing an increased number of database accesses to be made for some retrieval requests. We shall, however, preclude such degradation of IMS.

In the DBC environment, as we may recall, all root segments are clustered into as few cylinders as possible. All descendant segments of each root segment are also clustered, so that they may occupy as few cylinders as possible.

C. Estimating Tree Breadth and Cylinder Capacity

Because of the way that segments are physically stored via HIDAM or HDAM, it is very often the case that all twin segments at the lowest (leaf or terminal) level may be retrieved with a single page access (e.g., all the relevant pages are from the same track). But the twins at higher levels (including the root level) are generally scattered so that one page access is required for every twin. We shall, therefore, assume for the sake of the GPC environment that all the twins at the lowest level (i.e., all leaf segments having the same parent) occupy a single page on the average. But this implies that, depending on the tree breadth (namely, m·y, where m is the fanout and y is the average number of twins), the segment size will vary, since all m·y segments occupy a page. Accordingly, the cylinder capacity (in terms of segments) will vary as well. By varying the average size of a root segment and all its descendants from a fraction of a cylinder to, at most, a few cylinders, we can therefore estimate various tree breadths in this study.

The average number of levels in most practical databases is low, perhaps, two, three, or four; hence it is reasonable to assume a depth of three for the database. However, similar analyses can be carried out for databases with different depths.

If all the m·y children of a segment are to occupy a single page, then

$$\text{segment size} = \frac{\text{page size}}{my}$$

If all the descendant segments of a root segment, excluding the root, are to occupy a cylinder, then

$$\text{segment size} = \frac{\text{cylinder size}}{m^2y^2 + my}$$

since there are m·y segments in level 2 and m^2y^2 segments in level 3. Equating the two segment sizes and estimating a ratio of at least 50 for cylinder size to page size, we find that the tree breadth is 49, i.e., m·y = 49. (A disk cylinder has a capacity of about 500,000 bytes, so the ratio of cylinder size to page size is 50, even if the page size is as high as 10,000 bytes.) We thus find a way to estimate the tree breadth; in our final results, we shall vary m·y from 20 to 320.

Given a specific value for tree breadth (m·y), we compute the number of segments per cylinder as

$$\text{cylinder capacity} = \frac{\text{cylinder size}}{\text{segment size}}$$

$$= \frac{\text{cylinder size}}{\text{page size}} \cdot my = 50\ my.$$

D. Classification and Analysis of Transactions

We classify transactions involving retrieval requests into seven important categories and analyze the times T_g and T_d required to execute these transactions in the GPC and DBC environments, respectively. No secondary indices are used, but their presence will not distort the analysis by any appreciable amount. As is ordinarily the case, root segments have unique sequence field values. The retrieval requests are made for the lowest level segments (of a three-level database), but the analyses for higher levels are simple extensions of the one provided.

Each transaction type is first specified in terms of a requirement. An example of such a transaction is then provided, using the database of Figure 4. Finally, a general format is given of the transaction type, and the execution time is analyzed. In the general format, s_1, s_2, and s_3 are segment types in the first, second and third levels respectively. q_2 and q_3 are qualifications for the second and third level segments, respectively. q_1 and q_f are qualifications for root (first level) segments, but while q_f includes a predicate involving the sequence field, q_1 does not.

Transaction Type 1

Requirement: (1) Find a single segment satisfying a given condition
(2) An expression involving the sequence field is given at root level

Example: Find the student with employee number 50, taking a CIS 211 course in Columbus. We note that course numbers are sequenced.

```
GU  Course      Course#  = CIS 211
    Offering    Location = COLUMBUS
    Student     Emp#     = 50
```

General Format: GU s_1 q_f
s_2 q_2
s_3 q_3

GPC time analysis: One page access is needed to find the address of the required root segment, either via index (in HIDAM) or by hashing (in HDAM). At the second level, it may be expected that half the s_2-type segments (that are twins, and children of the above root segment) have to be searched before getting the one satisfying q_2. There are y twins, hence y/2 accesses are required, since the second level twins are physically scattered. At the third level, once again, y/2 twins may have to be searched before getting the right one. But they are stored contiguously, hence only one access is required. Thus,

$$T_g = 1 + \frac{y}{2} + 1.$$

DBC time analysis: Since there are N root segments (clustered together) and because there are (50 my) segments per cylinder,

$$\frac{N}{50my}$$

cylinder accesses are required to fetch the required root segment by content-searching the database. Since there are $(m^2 y^2 + my)$ descendant segments per root segment and they are clustered together, they occupy $\frac{m^2 y^2 + my}{50my} = \frac{my + 1}{50}$ cylinders. Hence, $\frac{my + 1}{50}$ accesses are required to find each of the necessary second and third level segments. However, cylinder sizes are large, so that these numbers may represent small fractions even though at least one access is always required at each level. Therefore, we take the ceiling (smallest integer greater than a given real number) of these numbers. Thus,

$$T_d = \left\lceil \frac{N}{50my} \right\rceil + \left\lceil \frac{my+1}{50} \right\rceil + \left\lceil \frac{my+1}{50} \right\rceil$$

Transaction Type 2

Requirement: (1) Find a single segment satisfying a given condition
(2) Any expression at the root level does not involve the sequence field

Example: Find the student with employee number 50, taking a mathematics course in Columbus.

```
GU   Course     Title      = MATH
     Offering   Location   = COLUMBUS
     Student    Emp#       = 50
```

General Format:

```
GU   s1   q1
     s2   q2
     s3   q3
```

GPC time analysis: Since q_1 does not involve an indexed (or hashed) field, an average of half the root segments must be searched before finding the right one. y/2 accesses are needed at the second level and one access at the third level as in transaction type 1. Thus,

$$T_g = \frac{N}{2} + \frac{y}{2} + 1$$

DBC time analysis: Same as in transaction type 1.

$$T_d = \left\lceil \frac{N}{50my} \right\rceil + \left\lceil \frac{my+1}{50} \right\rceil + \left\lceil \frac{my+1}{50} \right\rceil$$

Transaction Type 3

Requirement: (1) Find all segments satisfying a given condition
(2) An expression involving the sequence field is given at the root level

Example: Find all students who failed in some graduate course in CIS (i.e., CIS 600 and beyond) offered in Columbus.

```
GU   Course     Course#   ≥ CIS 600
     Offering   Location  = COLUMBUS
     Student    Grade     = F
Loop GN   Course     Course#   ≥ CIS 600
     Offering   Location  = COLUMBUS
     Student    Grade     = F
     GO TO Loop
```

General Format:

```
GU   s1   qf
     s2   q2
     s3   q3
Loop GN   s1   qf
     s2   q2
     s3   q3
     GO TO Loop
```

GPC time analysis: Assuming that the given condition is satisfied by n third level segments scattered evenly among the root segments and their descendants, then n root segments will have to be accessed. For each of these root segments, all y of its children (twins) need to be searched until one is found that satisfies the second level qualification. All children of this second level segment may be retrieved with one access to the database, since these twins are contiguously located. One of these third level twins satisfies the given condition.

$$T_g = n(1 + y + 1)$$

DBC time analysis: The necessary n root segments are retrieved in

$$\frac{N}{50\ my}$$

accesses. $\frac{my+1}{50}$ accesses are required to retreive the second level segments that are children of each of the root segments. Since there is one s_2-type segment that is a child of one of the retrieved root segments and satisfies q_2, a total of n $\frac{my+1}{50}$ accesses are required for the n second level segments. An identical number of accesses are also required for the third level segments.

$$T_d = \left\lceil \frac{N}{50my} \right\rceil + n \left\lceil \frac{my+1}{50} \right\rceil + n \left\lceil \frac{my+1}{50} \right\rceil$$

Transaction Type 4

Requirement: (1) Find all segments satisfying a given condition
(2) An expression at the root level does not involve the sequence field

Example: Find all students who failed in some mathematics course offered in Columbus.

```
      GU  Course     Title    = MATH
          Offering   Location = COLUMBUS
          Student    Grade    = F
Loop  GN  Course     Title    = MATH
          Offering   Location = COLUMBUS
          Student    Grade    = F
      GO TO Loop
```

General Format:

```
GU      s1   q1
        s2   q2
        s3   q3
Loop GN s1   q1
        s2   q2
        s3   q3
   TO TO Loop
```

GPC time analysis: Assuming that the given condition is satisfied by n third level segments evenly scattered, the analysis is the same as the one for transaction type 3, except that all N of the root segments must be searched anyway, since the qualification q_1 does not involve the sequence field.

$$T_g = N + n(y + 1)$$

DBC time analysis: Same as in transaction type 3.

$$T_d = \frac{N}{50my} + n \frac{my+1}{50} + n \frac{my+1}{50}$$

Transaction Type 5

Requirement:
(1) Find all segments satisfying a given condition
(2) An expression involving the sequence field is given at the root level
(3) No expression is given at any intermediate level

Example: Find all students who failed in some graduate course in CIS.

```
      GU  Course     Course# ≥ CIS 600
          Offering
          Student    Grade   = F
Loop  GN  Course     Course# ≥ CIS 600
          Offering
          Student    Grade   = F
      GO TO Loop
```

General Format:

```
GU      s1   qf
        s2
        s3   q3
Loop GN s1   qf
        s2
        s3   q3
   GO TO Loop
```

GPC time analysis: Assuming that the given condition is satisfied by n third level segments scattered evenly, n root segments will have to be accessed. For each of these root segments, all y of its children are retrieved. For each of these second level segments, all y of its children are retrieved in one access. One out of these y third level twins satisfies the given condition.

$$T_g = n(1 + y + y)$$

DBC time analysis: Since there is no expression at the second level, no access is required for second level segments.

$$T_d = \left\lceil \frac{N}{50my} \right\rceil + n \left\lceil \frac{my+1}{50} \right\rceil$$

Transaction Type 6

Requirement:
(1) Find all segments satisfying a given condition
(2) An expression at the root level does not involve the sequence field
(3) No expression is given at any intermediate level

Example: Find all students who failed a mathematics course.

```
      CU  Course     Title = MATH
          Offering
          Student    Grade = F
Loop  GN  Course     Title = MATH
          Offering
          Student    Grade = F
      GO TO Loop
```

General Format:

```
GU      s1   q1
        s2
        s3   q3
Loop GN s1   q1
        s2
        s3   q3
   GO TO Loop
```

GPC time analysis: Assuming that the given condition is satisfied by n third level segments evenly scattered, the analysis is the same as the one for transaction type 5, except that all N of the root segments must be searched anyway, since the qualification q_1 does not involve the sequence field.

$$T_g = N + n(y + y)$$

DBC time analysis: Same as in transaction type 5.

$$T_g = \left\lceil \frac{N}{50my} \right\rceil + n \left\lceil \frac{my+1}{50} \right\rceil$$

Transaction Type 7

Requirement:
(1) Find all segments satisfying a given condition
(2) No expression is given at any level, except, perhaps, the lowest level

Example: Find all students who obtained an A grade in some course.

```
        GU  Course
            Offering
            Student     Grade = A
Loop    GN  Course
            Offering
            Student     Grade = A
        GO TO Loop
```

General Format:

```
GU  s1
    s2
    s3   q3
Loop GN  s1
         s2
         s3
    GO TO Loop
```

GPC time analysis: All root segments, all s_2-type segments and all s_3-type segments are to be accessed.

$$T_g = N(1 + y + y)$$

DBC time analysis: All root segments are to be accessed, and for each root segment, all its descendant s_3-type segments are to be accessed.

$$T_d = \left\lceil \frac{N}{50my} \right\rceil + N\left\lceil \frac{my+1}{50} \right\rceil$$

E. Performance Gains

The ratio R_t of transaction execution times has been tabulated in Figure 11 for typical values of the parameters m, y, N and n. The average fanout m, representing the number of different types of child segments of any given segment, has been fixed at 2. The average number y of twins has been varied from 10 to 160, in multiples of 2. The number of root segments has been varied from 100 to 10,000. For transaction types 3 through 6, the number n of segments satisfying a given condition is considered between 4 and 64.

For transaction type 1, the ratio $R_t = T_g/T_d$ is normally a small number. For smaller numbers N of root segments, R_t does not appreciably vary with y, since both T_g and T_d tend to increase proportionately with y. This is because the time to access the root segments in DBC environment is small whenever N is small. For larger values of N, T_g increases proportionately with y but T_d does not. Hence R_t increases almost proportionately with y.

Similar reasoning can be applied for all the other transaction types. We shall, however, make some more general statements about the results. For transaction type 1, the GPC and DBC environments are not very different in terms of performance, because only a single segment satisfies the given condition and because GPC has the advantage of using its primary index. For transaction type 2, even though only a single segment is to be retrieved, GPC performs poorly since its primary index cannot be of any help, but DBC performs as well as before. Since GPC has the help of the primary index while executing type 3 and type 5

TRANSACTION TYPE 1

y \ N	100	1000	10000
10	2.33	2.33	0.58
20	1.00	1.00	1.71
40	1.10	1.40	3.14
80	1.47	1.47	1.20
160	5.47	5.47	5.47

TRANSACTION TYPE 2

y \ N	100	1000	10000
10	18.67	168.67	117.17
20	20.33	170.33	711.46
40	14.20	101.20	717.29
80	10.11	60.11	501.10
160	8.73	38.73	338.73

TRANSACTION TYPE 7

y \ N	100	1000	10000
10	20.79	20.94	20.94
20	10.50	10.96	10.94
40	10.30	10.18	10.49
80	10.15	10.21	10.25
160	15.70	15.85	15.86

TRANSACTION TYPE 3

	N=100					N=1000					N=10000				
y \ n	4	8	16	32	64	4	8	16	32	64	4	8	16	32	64
10	5.33	5.65	5.82	5.91	5.95	5.33	5.65	5.82	5.91	5.95	2.67	3.69	4.57	5.19	5.57
20	9.78	10.35	10.67	10.83	10.91	9.78	10.35	10.67	10.83	10.91	6.77	8.38	9.51	10.20	10.59
40	9.88	10.18	10.34	10.42	10.46	9.88	10.18	10.34	10.42	10.46	9.84	9.50	10.03	10.26	10.36
80	9.94	10.09	10.17	10.21	10.23	9.94	10.09	10.17	10.21	10.23	9.65	9.94	10.09	10.17	10.21
160	11.37	11.47	11.52	11.55	11.56	11.37	11.47	11.52	11.55	11.56	11.37	11.47	11.52	11.55	11.56

TRANSACTION TYPE 4

	N=100					N=1000					N=10000				
y \ n	4	8	16	32	64	4	8	16	32	64	4	8	16	32	64
10	16.00	11.06	8.30	6.95	6.23	116.00	64.00	35.64	20.40	13.21	556.00	388.00	242.29	139.89	77.57
20	20.14	15.76	13.21	11.88	11.19	120.14	68.71	40.18	25.72	18.17	775.68	484.19	279.35	154.67	85.29
40	15.53	12.97	11.63	10.95	10.60	68.47	40.24	25.18	17.92	14.10	534.95	293.09	159.64	86.35	48.74
80	12.45	11.51	10.82	10.47	10.30	40.12	25.35	17.80	13.98	12.05	303.65	161.33	86.89	48.81	29.54
160	13.05	12.28	11.89	11.70	11.60	28.84	20.25	15.49	13.70	12.60	186.74	99.89	55.89	33.75	22.64

TRANSACTION TYPE 5

	N=100					N=1000					N=10000				
y \ n	4	8	16	32	64	4	8	16	32	64	4	8	16	32	64
10	18.40	18.67	19.76	20.36	20.68	18.40	18.67	19.76	20.36	20.68	6.00	9.33	12.92	16.00	18.16
20	32.40	36.44	38.59	39.76	40.37	32.40	36.44	38.58	39.76	40.37	14.22	25.23	31.24	35.46	38.03
40	36.00	38.12	39.27	39.88	40.19	36.00	38.12	39.27	39.88	40.19	29.45	34.11	37.03	38.69	39.57
80	37.88	39.02	39.63	39.94	40.09	37.88	39.03	39.63	39.94	40.09	35.78	37.88	39.03	39.63	39.94
160	44.28	45.05	45.45	45.65	45.76	44.28	45.05	45.45	45.65	45.76	44.28	45.05	45.45	45.65	45.76

TRANSACTION TYPE 6

	N=100					N=1000					N=10000				
y \ n	4	8	16	32	64	4	8	16	32	64	4	8	16	32	64
10	36.00	28.89	24.71	22.42	21.23	216.00	128.89	77.05	49.70	35.68	720.00	561.11	396.92	253.33	152.43
20	52.00	41.67	33.53	31.82	30.92	232.00	140.67	90.47	59.00	41.27	1124.93	793.45	506.47	304.86	182.63
40	84.67	65.53	51.82	40.92	40.47	416.67	296.47	189.09	91.77	47.11	930.18	560.00	322.29	187.46	115.42
80	43.53	41.82	40.92	40.47	40.23	96.47	69.09	54.77	47.44	43.71	591.11	331.76	190.30	116.31	78.45
160	47.50	46.47	46.19	45.96	45.81	74.62	62.46	54.16	49.96	47.41	388.97	220.35	133.81	89.96	67.88

R_t = Transaction Execution Time in GIS environment / Transaction Execution Time in DBC environment

FIGURE 11. Transaction Execution Time Ratio (R_t)

transactions, DBC performs only about an order of magnitude better than GPC in such cases. DBC works far better in the case of transaction type 5 as compared to transaction type 3, because segments of intermediate levels need not be accessed in the type 5 transactions. A similar comment applies when comparing transaction types 6 and 4. DBC performs one, two, and even three orders of magnitude better than GPC in the execution of type 4 and type 6 transactions. This is because multiple segments have to be retrieved without the aid of indices. For transactions of type 7, almost the entire database (i.e., all segments of a single type of each level) has to be searched sequentially. Thus, in such a case, the gain of DBC over GPC is proportional to the cylinder size of DBC over the size of GPC. Overall, DBC performance is about one or two orders of magnitudes better than GPC performance.

F. Database Updates

So far, we have not considered the performance of DBC in carrying out database updates such as deletion of a group of related segments from the database or insertion of a new segment into the database. For all updates, IMS requires the position of a target segment to be first determined. The deletion and insertion calls in DL/1, therefore, have similar formats and processing requirements as the get calls. DBC also can treat the deletion and insertion calls in the same manner as it treats the get calls. In general, therefore, DBC inserts a single segment into the database almost as quickly as it can find a single segment. Similarly, DBC can delete a set

of segments satisfying a given condition as quickly as it can find all such segments, using the content-search capability. Consequently, DBC does update operations as well as it does retrieval operations.

6. CONCLUDING REMARKS

We have presented in this paper a methodology for supporting hierarchical database management on a special-purpose computer (DBC) that can content-search one whole disk cylinder in a single disk revolution. Address-dependent pointers are removed from database segments and replaced by symbolic identifiers that not only preserve all relationships determined by the pointers but also facilitate content-addressing. Thus, a basically sequential database can now be stored in DBC with considerable more flexibility. In order to make full use of the power of DBC, such as its data clustering mechanism, segments are clustered in a manner that takes advantage of the way the segments are normally accessed. We have also shown how the more important data sublanguage (DL/1) calls may be handled on DBC by using a single database access to search a set of twin segments instead of only one such segment.

Finally, an analysis of the database storage requirement and transaction execution time is presented. While the storage requirement in the new DBC environment is comparable to that in a conventional environment, there is a very large (one or two orders of magnitude) improvement in transaction execution time. The classification of transactions for this study captures a major portion of all possible transactions, although not exhaustive. The emphasis of this study is that DBC in particular, and database machines in general, can replace an existing hierarchical database management system software and conventional disk storage, and support the existing applications with very good performance.

Similar studies of DBC in supporting database application for relational databases can be found in [9,10] and for CODASYL databases in [11]. These studies are aimed at assessing DBC's capability to support multiple database applications and to develop general methodologies for database transformation and query translation.

ACKNOWLEDGEMENT

The entire work reported herein is conducted at The Ohio State University and supported by the Office of Naval Research through contract N00014-75-C0573. Portions of this paper are derived from a project report available either through NTIS under AD #A039038, or from The Ohio State University under OSU-CISRC-TR-77-1. This report was issued in April, 1977, and co-authored by David K. Hsiao, Douglas S. Kerr and Fred K. Ng.

REFERENCES

[1] Banerjee, J., Baum, R. I. and Hsiao, D. K., "Concepts and Capabilities of a Database Computer," ACM Transactions on Database Systems, Vol. 3, No. 4 (Dec. 1978), pp. 347-384.

[2] Banerjee, J., Hsiao, D. K., and Kannan, K., "DBC - A Database Computer for Very Large Databases," to appear in IEEE Transactions on Computers.

[3] Baum, R. I. and Hsiao, D. K., "Database Computers -- A Step Toward Data Utilities," IEEE Transactions on Computers, Vol. C-25, No. 12 (Dec. 1976), pp. 1254-1259.

[4] Ampex (9-track) Parallel Transfer Disk Drive (DM-PTD9), Product Announcement, November 1977.

[5] IBM, Information Management System/Virtual Storage (IMS/VS) Version 1, General Information Manual, GH20-1260-4.

[6] IBM, Information Management System/Virtual Storage (IMS/VS) Version 1, Application Programming Reference Manual, SH20-9026-4.

[7] IBM, Information Management System/Virtual Storage (IMS/VS) Version 1, System/Application Design Guide, GH20-9025-4.

[8] IBM, Information Management System/Virtual Storage (IMS/VS) Version 1, System Programming Reference Manual, SH20-9027-4.

[9] Banerjee, J. and Hsiao, D. K., "The Use of a Database Machine for Supporting Relational Databases," Proceedings of the Fourth Workshop on Computer Architecture for Non-numeric Processing, Syracuse, New York, August 1-3, 1978.

[10] Banerjee, J. and Hsiao, D. K., "Performance Evaluation of a Database Computer in Supporting Relational Databases," Proceedings of the Fourth International Conference on Very Large Data Bases, Berlin, Federal Republic of Germany, Sept. 13-15, 1978.

[11] Banerjee, J. and Hsiao, D. K., "A Methodology for Supporting Existing CODASYL Databases with New Database Machines," Proceedings of the ACM '78 Conference, Washington, D. C., Dec. 4-6, 1978.

THE USE OF A DATABASE MACHINE FOR SUPPORTING RELATIONAL DATABASES*

Jayanta Banerjee and David K. Hsiao

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

Abstract

One of the goals in the design of database machines of the future is their generality. In addition to being capable of carrying out the common database management functions with high reliability and performance, some of these machines are intended to support more than one data model. A specific database machine, known as the DBC, is intended to support several existing data models. Although the DBC supports many data models, we single out the relational data model for this discussion. In particular, we have tried to concentrate mainly on the subject of database representation and query translation of System R-like database management systems. Some estimates of the storage requirements and performance gains are given in this paper. However, due to limited space, the detailed analysis is shown elsewhere in [22].

Introduction

Advances in technology and database research have prompted considerable attention to the design and implementation of database machines [1,2,3,4]. With the design of a number of database machines either completed or underway [5,6,7,8], there are reasons to believe that the prototype construction of database machines is indeed viable. These machines will perform the basic database management functions with improved reliability and performance as compared to those obtained with software means.

One of the most difficult design decisions that confronts the database machine designer is the type of data structure which should be built into the machines. On the one hand, the designer would like to build into the machine a very elaborate and complex data structure so that it is sophisticated enough to emulate the high level data models such as the relational and CODASYL models, making the need of software support for such models superfluous. On the other hand, the designer would like to build into the machine a very simple and elegant data structure so that its straightforward implementation can lead to a machine with high performance and reliability. Such dichotomy is the main focus of the paper.

*The work reported herein was conducted at The Ohio State University and supported by contract N00014-75-C-0573 from the Office of Naval Research.

In this paper, we shall consider a specific database machine known as the DBC which is capable of supporting multiple data models [5,9,10,11,12]. However, the built-in data structure of the database machine is rather simple and straightforward. We would like to show how the data models, say, the relational data model [13,14], are supported on this machine. We would also mention its performance and its storage requirements for relational databases. It is estimated that the DBC storage requirements may not result in any saving over the conventional computer system. However, the DBC performance may be considerably better than what is achievable on a conventional computer system. In the absence of a large-scale commercial relational database management system, System R [15] has been used for the study. The relational language used, therefore, is SEQUEL 2 [16], which we shall refer to simply as SEQUEL. The study of the database machine in supporting other models such as hierarchical and CODASYL has been documented elsewhere [17,18]. Due to the limited length of this paper, we shall not present these findings here.

The Operating Environment - Front-End Computer and DBC

As a special-purpose computer, the DBC is intended to be used as a back-end machine to a front-end conventional computer. The front-end computer supports all application programs, the operating system and a specialized package called the RDBI (Relational Database Interface). The basic organization of the front-end and back-end computer is depicted in Figure 1.

A user who does not want to make use of the database may simply interact with the operating system of the front-end computer to execute his program. A database user, on the other hand, calls upon the services of the RDBI in order to access the relational database which is stored in the DBC. The user programs that access the database may either be written in the stand-alone version of SEQUEL or in a host programming language which embeds SEQUEL as a data sublanguage. In either case, the SEQUEL statements are identified by the operating system (perhaps, with the aid of a precompiler) and transmitted to the RDBI. The RDBI will then execute the statement by sending appropriate commands to the DBC, collecting the DBC responses in its buffer, and sending the final results back to the operating system.

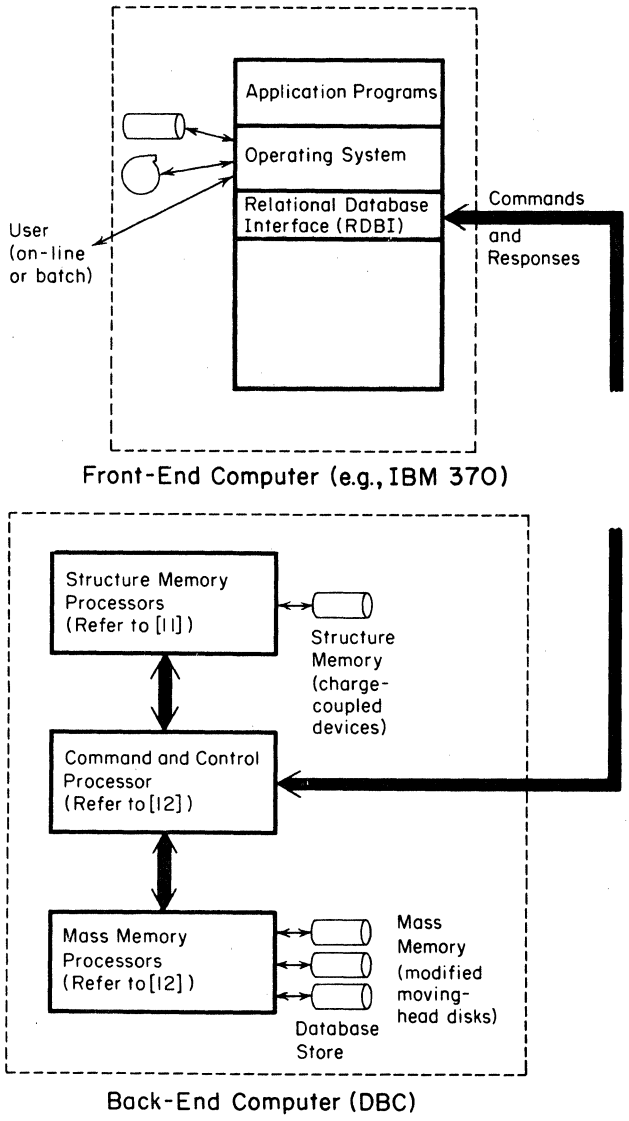


Figure 1. The Basic Relationship of the Front-end and Back-end Computers

The DBC stores the database in an on-line mass memory, which is made of modified moving-head disks. The tracks of a disk cylinder can be accessed and content-addressed simultaneously within a single disk revolution. Access is limited, however, to only one cylinder per drive at a time. Since the number of content-addressable processors is as few as the number of tracks in a given cylinder, the cost reduced in this organization as compared to cost incurred in database machines that associatively address all the cells (e.g., cylinders) may become the single most important factor that makes the DBC possible for very large database stores of the order of 10^{10} bytes. As content-addressable processors, they function simultaneously to search an entire cylinder for records that satisfy a given set of predicates.

Since the on-line mass memory is not a monolithic associative memory, it is important to restrict any database search to as few cylinders as possible. A directory of the database including other structural information of the database (such as security-related information) is, therefore, maintained in a content-addressable memory called the structure memory. The size of this memory is about 1% of the mass memory and is about 20 times faster. Charge-coupled devices (CCDs) are a very cost-effective choice for constructing this memory, as observed in [9,11].

Another major component of the DBC is a controller called the database command and control processor. When a command from the RDBI is sent to the DBC, this controller will decode it, enforce access control by consulting the structure memory, determine the cylinders to be searched with the aid of the structure memory, issue appropriate orders to the mass memory, post-process retrieved data, and transfer the data to/from the RDBI. Although the DBC is well documented elsewhere, this brief outline will be sufficient for our subsequent discussion.

The DBC Data Model

A database in the DBC is a collection of records. A record, in turn, is made up of an ordered set of data items called attribute-value pairs. An attribute-value pair is a member of the product set AT x VA, where AT is a set of "attributes" and VA is a set of "values". Within a record, the attribute part of every attribute-value pair is distinct. The attribute-value pairs that characterize a record (or a group of records) by distinguishing the record (or the group) from all others are called keywords.

A relational operator is an element of the set $\{=,\neq,<,\leq,\geq,>\}$. A triple of the form <attribute, relational operator, value> is called a keyword predicate. A keyword <A,V> is said to satisfy a keyword predicate <Ap,Op,Vp> if and only if A=Ap and V Op Vp, i.e., V and Vp are related by the operator Op. A query is a Boolean expression of keyword predicates in disjunctive normal form. Thus, a query is a disjunction of query conjuncts, which are conjunctions of keyword predicates. A record satisfies a query if it satisfies at least one query conjunct in the query. The set of all records that satisfy a query is called the response set of the query.

As an example of the types of queries that may be recognized by the DBC, consider the following:

([DEPT='TOY']&[SALARY<10000]) ∨
([DEPT='BOOK']&[SALARY>50000]).

If the above query refers to employee records of a department store, then it will be satisfied by records of the employees working either in the toy department and earning less than 10,000, or working in the book department and making more than 50,000.

Queries are used not only to retrieve a set of records among all the records in a database but also to specify protection requirements and clustering conditions.

DBC Commands

While the DBC is provided with a repertoire

of access and preparatory commands, we shall restrict ourselves here only to a simplified description of the retrieve command (and some of its various forms). This is because in this paper we intend only to illustrate how relational query facilities (or retrieval facilities) are handled in a database managed by the DBC. A description of other relational facilities (such as data control and data manipulation facilities) and their implementation on the DBC will be found in [19]. A detailed description of the DBC commands may be found in [12,20].

A retrieve command has the following two simplified forms, where the square brackets are used as metasymbols to indicate zero or one occurrence of the expression inside them:

Form 1

```
RETRIEVE:[set-function([attribute-1]) [ONLY]]
         ([(UNIQUE)] attribute-list) (query)
         [SORT BY attribute-2]
```

The command requires that the database be first searched to find all records that satisfy the given query. Of the response set, the values will be retained if their attributes appear in the attribute-list (which assumes a '*' if all values, i.e., entire records, are desired). In case the UNIQUE option is specified, then all partial records are discarded. These retrieved records are ordered by attribute-2. The DBC can perform by hardware a number of set functions such as AVG (which computes the average value of the elements in a set), MAX, MIN and SUM (which compute the maximum, minimum and sum, respectively, of the elements in a set). In the retrieve command, attribute-1 refers to the attribute-value pair of each retrieved record, whose attribute part is the same as attribute-1. The set function is performed on all these attribute-value pairs. In case the set function is COUNT, then attribute-1 may be null, in which case the number of retrieved records is counted. The ONLY option is used if the response set of the command is to consist of the set function alone instead of records. Attribute-1 and attribute-2 are both required to appear in attribute-list.

Form 2

```
RETRIEVE:(attribute-list-1) (query-1)
         CONNECT ON (attribute-1,attribute-2)
         (attribute-list-2) (query-2)
```

This command specifies that the set A of records that satisfy query-1 and the set B of records that satisfy query-2 be retrieved. The attribute-value pairs corresponding to attributes in attribute-list-1 are extracted from records of set A to form a set A1 of partial records. Similarly, the attribute-value pairs of attribute-list-2 are extracted from records of set B to form the set B1. An equality join is now made of the two sets of records A1 and B1 to create the final response set. The connecting attributes of the join operation are attribute-1 of set A1 and attribute-2 of set B1. Any record of the response set has three parts: attribute-value pairs corresponding to attribute-list-1 (except attribute-1), attribute-value pair corresponding to attribute-1 and attribute-value pairs corresponding to attribute-list-2 (except attribute-2). Note that it is necessary that attribute-1 be one of the attributes in attribute-list-1 and attribute-2 be one of the attributes in attribute-list-2.

The Relational Data Model

We shall provide here a very brief look at the relational model and the data sublanguage SEQUEL. Conceptually, a relation is a table in which each column corresponds to a distinct attribute and each row corresponds to a distinct entity or tuple. Each tuple is distinct in the sense that no two tuples in a relation have identical values for all attributes. A relation or table in a relational database exists in a normalized form, which means that every column of the table represents a simple attribute and is not itself another relation. Other improvements on the normal form are described in [21].

A comprehensive relational database management system which includes provisions such as simple but flexible user views, data definition, data manipulation and query capabilities, as well as convenient access support, system recovery and integrity enforcement can be found in System R [15]. System R provides user interface through a data sublanguage called SEQUEL [16]. Although the complete collection of System R facilities is available through SEQUEL, we shall concentrate in this paper mostly on the query capabilities, which constitute the most basic operations of the SEQUEL language. A discussion on how the other facilities of SEQUEL are handled in the DBC will be found in [19].

A sample database, extracted from [16], is depicted in Figure 2. It consists of four normalized relations. The EMP relation describes a set of employees, giving the employee number, name, department number, job title, manager's employee number, salary and commission for each employee. The DEPT relation gives the department number, name and location of each department. The USAGE relation describes the parts which are used by the various departments. The SUPPLY relation describes the supplier companies from which the various parts may be obtained. We shall make extensive reference to this sample database in all our later examples.

Relation	Attributes
EMP	EMPNO,NAME,DNO,JOB,MGR,SAL,COMM
DEPT	DNO,DNAME,LOC
USAGE	DNO,PART
SUPPLY	SUPPLIER,PART

Figure 2. A Sample Database

We conclude this section with a short example on the use of the query facilities of SEQUEL. For example, to find the names of employees in Dept. 100, one may write

```
SELECT NAME
FROM   EMP
WHERE  DNO=100
```

The SELECT clause lists the attributes to be returned. If the entire tuple is desired, then one may write SELECT *. The WHERE clause may contain any collection of predicates which compare values of attributes of a tuple to constant values (e.g., DNO=100) or compare values of two attributes of a tuple with each other (e.g., SAL<COMM). The predicates may be connected by AND and OR, and parentheses may be used to establish precedence. In our later example queries, extracted from [16], we shall demonstrate the other varieties of query

facilities in SEQUEL and how these queries are supported by the DBC.

Representing A Relational Database

Each tuple of a relation is stored in the DBC as a single DBC record. Not surprisingly, this record format closely resembles the logical structure of the tuple. While a tuple is normally seen as a sequence of values, where the position of each value identifies the underlying column, it is not sufficient in the DBC to store the values alone for each record. To allow for overlap among the individual domains (of the columns) and because the DBC ignores the absolute positions of the keywords in a record, it is necessary to store the column names as well, within the keywords. Therefore, whenever a tuple is to be stored in the database, the RDBI (relational database interface) creates a DBC record which consists of only keywords. One keyword is created, as shown below, for every column of the relation

<column-name, value>

where the attribute part of the keyword is the name of the column (in a coded form).

In order that the DBC may recognize a tuple of one relation from that of another, an extra keyword, as shown below, is added to each DBC record:

<RELATION, relation-name>

where the value of the keyword is the name of the relation to which the tuple belongs.

Thus, any tuple of the DEPT relation of Figure 2 is represented in the DBC by means of a reocrd with the following attribute-value pairs:

<RELATION, DEPT>
<DNO, department-number>
<DNAME, department-name>
<LOC, department-location>

If any one of the columns does not have a corresponding value in some tuple, then it is not necessary to create (or store) an attribute-value pair for that column. Thus, every DBC record representing a DEPT tuple will have an attribute-value pair for DNO (if this column always takes a non-null value), but it may not have such a pair for LOC (if the department is newly planned and is yet to come into existence).

To improve database performance, the DBC records are primarily clustered according to the keyword with attribute RELATION. That is, all those DBC records that correspond to the tuples of a relation are clustered together. Secondary clusters are formed based on database definition, such as clustering links and clustering images [15].

Translation of SEQUEL Queries

Once the database is created on the DBC by appropriate representation of the relational database, all the normal data management functions may then be carried out by the DBC. Every SEQUEL query received by the RDBI is translated into a sequence of DBC commands, some of which may depend on the results of previous commands within the sequence. In each of the following examples, the statement of a problem is first made, then a SEQUEL statement is written to solve the problem and finally this SEQUEL statement is translated into a sequence of one or more DBC commands. The database referenced is the one shown earlier in Figure 2.

Example 1: The following SEQUEL statement and DBC command will find the names of employees in Dept. 50.

```
SEQUEL:
  SELECT NAME
  FROM   EMP
  WHERE  DNO=50
DBC Command:
  RETRIEVE:  (NAME) ((RELATION='EMP')&(DNO=50))
```

Example 2: To list the names of employees in departments 25, 47 and 53, the following statement may be used.

```
SEQUEL:
  SELECT NAME
  FROM   EMP
  WHERE  DNO IN (25,47,53)
DBC Command:
  RETRIEVE:  (NAME) (((RELATION='EMP')&(DNO=25))
                    v((RELATION='EMP')&(DNO=47))
                   v((RELATION='EMP')&(DNO=53)))
```

Example 3: Consider listing the names of employees who work for departments in Evanston. This type of transaction requires access to two different relations and is, therefore, expressed in SEQUEL by means of a nested SELECT statement. The inner part of the nesting returns the collection of DNO values of the departments located in Evanston. The outer part then proceeds as though it were given a set of constants in lieu of the inner SELECT clause.

```
SEQUEL:
  SELECT NAME
  FROM   EMP
  WHERE  DNO IN
         SELECT DNO
         FROM   DEPT
         WHERE  LOC='EVANSTON'
DBC Commands:
  a.  RETRIEVE: (DNO) ((RELATION='DEPT')&(LOC=
  EVANSTON')). For each department number 'di'
  retrieved by (a), the RDBI issues the DBC com-
  mand:
  b.  RETRIEVE:(NAME) (RELATION='EMP')&(DNO=
  'di'))
```

Example 4: An important class of queries is exemplified in the determination of average salary of clerks. The built-in SEQUEL function AVG can be used to accomplish this result. Other built-in functions in the SEQUEL language are SUM, COUNT, MAX and MIN.

```
SEQUEL:
  SELECT AVG(SAL)
  FROM   EMP
  WHERE  JOB='CLERK'
DBC Command:
  RETRIEVE: AVG(SAL) ONLY
            (*) ((RELATION='EMP')&(JOB='CLERK'))
```

Notice that the (*) in the DBC command indicates that entire records must be retrieved before the function AVG is performed. Of course, the same effect could have been achieved by replacing the (*) with (SAL), thereby avoiding the cost of storing entire records in the DBC. The clause ONLY indicates that only the value of the function need be returned to the RDBI.

Example 5: The following statement determines the count of all the different jobs held by employees

in Dept. 50.

```
SEQUEL:
  SELECT COUNT(UNIQUE JOB)
  FROM   EMP
  WHERE  DNO=50
DBC Command:
  RETRIEVE: COUNT( ) ONLY
            ((UNIQUE) JOB) ((RELATION='EMP')&
             (DNO=50))
```

Example 6: Consider listing all the departments and the average salary of each. This is an example of a query in which a relation needs to be partitioned into groups. A built-in function can then be applied to each group.

```
SEQUEL:
  SELECT DNO,AVG(SAL)
  FROM   EMP
  GROUP  BY DNO
DBC Commands:
  a.  RETRIEVE: ((UNIQUE) DNO) (RELATION='EMP')
```

For each department number 'di' retrieved by (a), the RDBI issues a command:

```
  b.  RETRIEVE: AVG(SAL) ONLY
                (*) ((RELATION='EMP')&(DNO='di'))
```

Example 7: Sometimes it may be desired to partition a relation into groups and then to apply a predicate or a set of predicates which chooses only some of the groups and disqualifies other. These group-qualifying predicates are placed in a special HAVING clause. A predicate in a HAVING clause may compare an aggregate property (e.g., AVG(SAL)) of a group to a constant or to another aggregate property of the same group. The followng SEQUEL statement may be used to list all those departments in which the average employee salary is less than 10,000.

```
SEQUEL:
  SELECT DNO
  FROM   EMP
  GROUP  BY DNO
  HAVING AVG(SAL)<10000
DBC Commands:
  a.  RETRIEVE: ((UNIQUE) DNO) (RELATION='EMP')
```

For each department number 'di' retrieved by (a) the RDBI issues a command:

```
  b.  RETRIEVE: AVG(SAL) ONLY
                (SAL) (RELATION='EMP')&(DNO=
                 'di'))
```

Since the DBC does not make comparisons on aggregate properties, the final selection of DNO based on (AVG(SAL)<10000) is done by software (i.e., by the RDBI) in the front-end computer.

Example 8: Set comparison operators like =, ≠, [IS] [NOT] IN, CONTAINS and DOES NOT CONTAIN are allowed in a HAVING clause as illustrated by this example, which lists the departments which have employees with every possible job title.

```
SEQUEL:
  SELECT DNO
  FROM   EMP
  GROUP  BY DNO
  HAVING SET(JOB)=
         SELECT JOB
         FROM   EMP
DBC Commands:
  a. RETRIEVE: ((UNIQUE) DNO) (RELATION='EMP')
  b. RETRIEVE: ((UNIQUE) JOB) (RELATION='EMP')
               SORT BY JOB
```

For every department number 'di' retrieved by (a), issue the command:

```
  c. RETRIEVE: ((UNIQUE) JOB) ((RELATION='EMP')&
                (DNO='di'))
               SORT BY JOB
```

For each department, the comparison of each of the sets in (c) to the set in (b) is done by software (i.e., by the RDBI).

Example 9: A join operation may be required to return values selected from more than one relation. The names of all employees and the locations where they work may be listed by the query:

```
SEQUEL:
  SELECT EMP.NAME,DEPT.LOC
  FROM   EMP,DEPT
  WHERE  EMP.DNO=DEPT.DNO
DBC Command:
  RETRIEVE: (NAME,DNO) (RELATION='EMP')
            CONNECT ON (DNO,DNO)
            (LOC,DNO) (RELATION='DEPT')
```

Here, there are two attribute lists: (NAME,DNO) for the first query and (LOC,DNO) for the second query. The command is to connect (join) on the two DNO attributes and return as response data triples of the form (NAME,DNO,LOC), where NAME is taken from the first attribute list, LOC is taken from the second list, and DNO is common to both. The RDBI now returns to the user only the pairs (NAME,LOC) by deleting DNO from the triples returned by the DBC.

Example 10: In some circumstances, it is necessary to join a relation with itself according to some criterion. The relation name may then have to be listed more than once and labeled, e.g., X and Y may be two labels for a relation EMP. As an example, the following SEQUEL query will list the employee's name and his manager's name for each employee whose salary exceeds his manager's salary.

```
SEQUEL:
  SELECT X.NAME,Y.NAME
  FROM   EMP X,EMP Y
  WHERE  X.MGR=Y.EMPNO
  AND    X.SAL>Y.SAL
DBC Commands:
  a. RETRIEVE: (MGR) (RELATION='EMP')
               CONNECT ON (MGR,EMPNO)
               (EMPNO) (RELATION='EMP')
```

The only difference between this command and the command for Example 9 is that only one attribute is returned, instead of X.NAME and Y.NAME as well. This is because the AND clause has still got to be considered. Notice that since a manager has at least one employee (in general), a modified command (a') would also have the same effect as (a), yet taking less time to execute. However, (a') is not general enough for all situations.

```
  a'. RETRIEVE: ((UNIQUE) MGR) (RELATION='EMP')
```

For each manager number 'mi' returned by (a), do the following: Send a command

```
  b. RETRIEVE: (NAME,SAL) ((RELATION='EMP')&
               (EMPNO='mi'))
```

and for each (nj,sk) pair returned by (b), send a command

```
  c. RETRIEVE: (NAME) ((RELATION='EMP')&(MGR='mi')
               &(SAL>sk))
```

Notice that the name retrieved by (c) is an employee name, and that returned by (b) is the corresponding manager's name.

Steps (b) and (c) have been written in such a way that for every manager, the DBC accesses all

his employees at the same time. These two steps could otherwise have been written such that for every employee, the DBC accesses all his managers at the same time. But, of course, every employee has a single manager. Therefore, the way we have written the commands is better than its alternative, since fewer number of accesses is required in the former case. The decision is made on the basis of the fact that there are fewer unique values of MGR than there are of EMPNO.

Example 11: SEQUEL permits a label to be used to qualify attribute names outside the block in which the label is defined. The following query uses this feature in listing the suppliers who supply all the parts used by Dept. 50.

```
SEQUEL:
  SELECT SUPPLIER
  FROM   SUPPLY X
  WHERE
         (SELECT PART
          FROM   SUPPLY
          WHERE  SUPPLIER=X.SUPPLIER)
  CONTAINS
         (SELECT PART
          FROM   USAGE
          WHERE  DNO=50)
DBC Command:
  a. RETRIEVE: ((UNIQUE) SUPPLIER) (RELATION=
     'SUPPLY')
  b. RETRIEVE: (PART) ((RELATION='USAGE')&(DNO=
     50))
```

Since the block after CONTAINS has a comparison involving a constant, it needs to be executed only once. This is done by command (b) given above. For each supplier 'si' retrieved by (a), a DBC command

```
  c. RETRIEVE: (PART) ((RELATION='SUPPLY')&
     (SUPPLIER='si'))
```

is sent, and the sets retrieved by (b) and (c) are compared by software.

The same query could have been made in SEQUEL by means of GROUP BY and the special function SET, as given below:

```
SELECT SUPPLIER
FROM   SUPPLY
GROUP  BY SUPPLIER
HAVING SET(PART) CONTAINS
       SELECT PART
       FROM   USAGE
       WHERE  DNO=50
```

The DBC commands would be the same as before.

A Brief Look at Performance

Because of the parallelism involved in the operations performed by the DBC, it should be intuitively clear that user transactions will run faster on the DBC than on a conventional computer. The speed is further enhanced by the fact that a sequence of software operations can be replaced completely by a single DBC command. For example, in order to find all the records satisfying a conjunct of predicates, a conventional system will first determine (in some manner, e.g., via an index) the eligible records. It will then retrieve these records and compare each of them against the given predicates. In the DBC, on the other hand, not only are all the eligible records retrieved in parallel, but it is also true that this set of retrieved records is exactly the required response set. The reason is simply that records are compared against the given predicates simultaneously with their retrieval, thereby rendering unnecessary any subsequent software refinement of the retrieved set.

In the rest of this section we shall consider for study the mass storage requirement, directory storage requirement and the execution time of queries. Rather than a complete detailed analysis, what we provide here is more of a motivation for understanding the difference in performance between the DBC and conventional computer systems. A detailed analysis is presented in [19] and published in [22].

For every tuple stored in a conventional system, the DBC stores a record in its mass memory. While a stored tuple consists of pointers (at least one for each link [15]) and the values for each column of a relation, a DBC record consists of keywords. Within a DBC record, each keyword is made up of a coded attribute as well as a value. In addition, there are one or two special keywords, such as <RELATION, relation-name>, but there are no pointers. If the average length of a value is about double (or more) the size of a coded attribute (which is quite normal), then the DBC mass storage requirement is usually no more than double that of a conventional system, even if no links are defined on the relation. On the other hand, if there are many links, then the DBC storage requirement can actually be somewhat less than that of a conventional system.

With regard to directory storage, it must be pointed out that in the DBC implementation of a relational database, directories are maintained for the relation names and for one clustering attribute per relation. Since the DBC records are primarily clustered by relation name, the size of each entry in the directory for relation names will be quite small. The clustering attribute chosen for a relation is not one of the original attributes but a totally new one. Based on the clustering images and the clustering links, this clustering attribute is allowed to take on as many values as the number of cylinders required to store the entire relation. This is because individual cylinders are content-addressable; so there is no need to keep track of records of a relation within individual cylinders. Furthermore, due to the large size of the cylinders only a few of them will be used for accommodating a relation. For every record of a relation, then, a value is computed (based on its original attribute that appears in the clustering image or link) for the clustering attribute. The record is then stored "close" to other records of the same relation that have a matching value for the clustering attribute. Since the possible number of values of a clustering attribute is very small, the corresponding directories will also be small.

On the other hand, in a conventional system, a multi-page index is maintained for every image of a relation in the form of a modified B-tree [15] Since index entries address pages which are much smaller than cylinders in size, there will be many more index entries per image than the value entries for the DBC clustering attribute. It has been estimated in our analysis [19] that even if there is only one image per relation, directory storage

requirement in this system (for usual relations, say, consisting of 1,000 to 100,000 tuples, each of size 50 to 1,000 bytes) is 10 to 100 times the amount required by the DBC.

Query execution time is normally very much (about 10 to 100 times) faster on the DBC. The reasons are the following: (1) In one secondary storage access, the DBC can content-search an entire cylinder instead of scanning only a single page. Since a normal page size is close to 4,000 bytes, while a cylinder can accommodate as many as 400,000 bytes, it is not unreasonable to expect one or two orders of magnitude increase in speed when the DBC is used; (2) The records retrieved by the DBC are normally the records required in the response set of the query. This compares with the fact that in a conventional system, many of the tuples within a retrieved page will not be immediately required and will, therefore, be wasted; (3) The clustering policy used in the DBC implementation, which we have not discussed in detail, tries to optimize the search policy, without incurring an inordinately large storage overhead.

Concluding Remarks

In the limited space available for this paper, it was not possible to discuss a complete DBC implementation of the relational data model. We have tried to describe mainly the database representation problem and the qeury translation aspects. Details on the record clustering problem and its relation to the clustering links and clustering images have not been included. The data control facilities of System R may be implemented on the DBC in a conventional manner. Taking advantage of the hardware security mechanisms of the DBC, however, an extra degree of flexibility can be attained in solving the security problem [19]. Finally, listed below are the results of a performance analysis [19], which we have broadly overviewed in our last section:

(1) For a relational database, the mass memory of the DBC requires typically up to two times more storage than a conventional system.
(2) The storage used within the DBC structure memory is typically one or two orders of magnitude less in size than that required for storing indexes in a conventional system.
(3) The execution time required for the common SEQUEL queries (simple one-relation or two-relation queries) is normally one or two orders of magnitude faster when the DBC is used.

References

[1] Baum, R.I. and Hsiao, D.K., "Database Computers - A Step Towards Data Utilities," IEEE Trans. on Computers, C25, 12, Dec. 1976, pp. 1254-1259.

[2] Hsiao, D.K., "Data Base Computer - Why and How" Data Base Engineering, IEEE Computer Society, 1,2, June 1977, pp. 4-7.

[3] Lowenthal, E.I., "A Survey: The Application of Data Base Management Computers in Distributed Systems," Proc. Third Int. Conf. on Very Large Data Bases, ACM, New York, 1977, pp. 85-92.

[4] Hsiao, D. K. and Madnick, S. E., "Database Machine Architecture in the Context of Information Technology Evolution," Proc. Third Int. Conf. on Very Large Data Bases, ACM, New York, 1977, pp. 63-84.

[5] Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer - A Summary," Proc. Third Workshop on Computer Architecture for Non-Numerical Processing, Syracuse, New York, May 17-18, 1977.

[6] Ozkarahan, E. A., Schuster, S. A. and Sevcik, K. C., "Performance Evaluation of a Relational Associative Processor," ACM Trans. on Database Systems, 2, 2, June 1977, pp. 175-195.

[7] Lin, C. S., Smith D. C. P. and Smith J., "The Design of a Rotating Associative Array Memory for a Relational Database Management Application," ACM Trans. on Database Systems, 1, 1, March 1976, pp. 53-65.

[8] Copeland, G. P., Lipovsky, G. J. and Su, S. Y. W., "The Architecture of CASSM" A Cellular System for Non-Numeric Processing," Proc. First Annual Symp. on Computer Architecture, Dec. 1973, pp. 121-128.

[9] Hsiao, D. K., Kannan, K. and Kerr, D. S., "Structural Memory Designs for a Database Computer," Proc. Nat. ACM Conf., ACM, New York, 1977.

[10] Baum, R. I., Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer - Part I: Concepts and Capabilities," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-1, Sept. 1976.

[11] Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer - Part II: The Design of Structure Memory and Related Processors," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-2, Oct. 1976.

[12] Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer - Part III: The Design of the Mass Memory and Its Related Components," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-3, Dec. 1976.

[13] Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," Comm. of the ACM, 13, 6, June 1970, pp. 377-387.

[14] Codd, E. F., "Further Normalization of the Data Base Relational Model," in Courant Computer Science Symp. 6: Data Base Systems, Prentice-Hall, Englewood Cliffs, N.J., May 1971, pp. 65-98.

[15] Astrahan, M. M., et al., "System R: Relational Approach to Database Management," ACM Trans. on Database Systems, 1, 2, June 1976, pp. 97-137.

[16] Chamberlin, D. D., et al., "SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control," IBM Rep. No. RJ1798 (#26096), IBM Thomas J. Watson Research Center, N. Y., June 1976.

[17] Hsiao, D. K., Kerr, D. S. and Ng, F. K., "DBC Software Requirements for Supporting Hierarchical Databases," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-77-1, April 1977.

[18] Banerjee, J., Hsiao, D. K. and Kerr, D. S., "DBC Software Requirements for Supporting Network Databases," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-77-4, June 1977.

[19] Banerjee, J. and Hsiao, D. K., "DBC Software Requirements for Supporting Relational Databases," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-77-7, Nov. 1977.

[20] Banerjee, J., Hsiao, D. K. and Ng, F. K., "Data Network - A Computer Network of General-Purpose Front-End Computers and Special-Purpose Back-End Database Machines," Symp. on Computer Network Protocols, Liege, Belgium, 13-15, Feb. 1978.

[21] Date, C. J., An Introduction to Database Systems, Second Ed., Addition-Wesley, Reading Massachusetts, 1977.

[22] Banerjee, J. and Hsiao, D. K., "Performance Study of a Database Machine in Supporting Relational Databases," Accepted for publication in the Proceedings of the 4th International Conference on Very Large Databases, Berlin, Federal Republic of Germany, Sept. 1978.

PERFORMANCE STUDY OF A DATABASE MACHINE IN SUPPORTING RELATIONAL DATABASES*

Jayanta Banerjee & David K. Hsiao

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210 U.S.A.

ABSTRACT

Database machines are special-purpose devices that are expected to perform the common data management operations efficiently. In this paper, we attempt to show how a relational database can be supported on a specific database machine, known as the database computer (DBC), with good performance.

The DBC employs modified moving-head disks for database storage. To achieve high-volumed accessing, the read-out mechanisms of the moving-head disks are made into tracks-in-parallel. To provide content-addressable search, the disk controller is incorporated with a set of microprocessors, corresponding to the tracks of a cylinder. In this way, not only can an entire cylinder of data be accessed in one disk revolution, but relevant data which satisfies the user request can also be found and output in the same revolution.

To minimize the number of cylinders involved in a database access, some structural information about the database is maintained in a block-oriented content-addressable memory made of charge-coupled devices (CCDs). Furthermore, clustering and security mechanisms are a part of the hardware features provided by the DBC.

With cylinder-oriented content-addressable database store, block-oriented content-addressable structure memory and several functionally specialized components, the DBC can achieve one or two orders of magnitude of performance improvement over the conventional computer in database management. Also, a possible twofold increase in database storage requirement as compared to a conventional implementation is adequately offset by one or more orders of magnitude reduction in storage for structural information.

The purpose of this paper is to analyze these performance issues. By using the DBC for supporting relational databases, the size of the relational software is considerably reduced. Specifically, the query optimizer of conventional systems is now rendered unnecessary. In comparison with a conventional implementation of a relational system, the DBC has been found to contribute larger performance gains. These gains are tabulated in the paper. All these tend to demonstrate that the DBC in particular and database machines in general can indeed contribute to an appreciable improvement in database management.

*The work herein was conducted at The Ohio State University and supported by contract N00014-75-C-0573 from the Office of Naval Research.

INTRODUCTION

Supported on a database machine, a relational database management system can exhibit a performance that is considerably superior to that which can be achieved on a conventional computer. In this paper, we concentrate on an analysis of the performance gain of such a system when implemented on the database machine instead of the conventional computer. The database machine under consideration is called the database computer (DBC), which has been motivated, designed and documented in [1-6]. With sufficient built-in generality, the DBC can support a number of existing data models and database management systems. The study of the DBC in supporting hierarchical, CODASYL, and relational systems has been conducted and documented in [7], [8], and [9], respectively. In this paper we concern ourselves only with the DBC's performance in supporting the relational system. Thus, the material presented in this paper is extracted from [9].

In the following sections, we begin by taking a brief look at the DBC architecture and capability. We then proceed on to study in brief the implementation of a relational database management system on the DBC. Finally, we analyze the performance of this implementation. For reasons of specificity, we have chosen the relational database management system, System R, and its data sublanguage, SEQUEL, for our study [10], [11]. The analysis is of a comparative nature, since we evaluate the performance of the DBC relative to the implementation of System R. Both the database storage requirement and directory storage requirement are separately investigated. The analysis of query execution time is similar in flavour to that of [10]. The overall analysis is considerably different from the one carried out for RAP [12], since the DBC makes use of indices and data clustering, while RAP does not.

A BRIEF LOOK AT THE DATABASE COMPUTER

A record in the DBC consists of a set of ordered pairs, <attribute, value>, some of which are designated as keywords. The attribute usually names a property whereas the value identifies a specific instance of an attribute. A keyword predicate is a triple of the form:

<attribute, relational operator, value>

where the relational operator is one of the set $\{=, \neq, <, \leq, \geq, >\}$. A keyword $<A, V>$ is said to satisfy a keyword predicate $<A_p, O_p, V_p>$ if and

only if $A = A_p$ and $V\ O_p\ V_p$, i.e., V and V_p are related by the operator O_p. A group of records with similar properties may be specified by means of a query which is a Boolean expression of keyword predicates in disjunctive normal form. Thus, a query is a disjunction of conjuncts known as query conjuncts, which are simply conjunctions of keyword predicates. The set of all records that satisfy a query is the response set of the query. When given a query, the DBC is capable of retrieving all records of the database that satisfy the query, i.e., the response set of the query.

As an example of the types of queries that may be processed by the DBC, consider the following user request for records:

$$([DEPT = 'TOY'] \wedge [SALARY < 10000]) \vee$$
$$([DEPT = 'BOOK'] \wedge [SALARY > 50000]).$$

If the above query refers to a file of employees of a department store, then the DBC will retrieve records of the employees working either in the toy department and earning less than 10,000 or working in the book department and making more than 50,000.

Schematically, the DBC architecture consists of two loops of memories and processors, namely, the structure loop and the data loop as depicted in Figure 1.

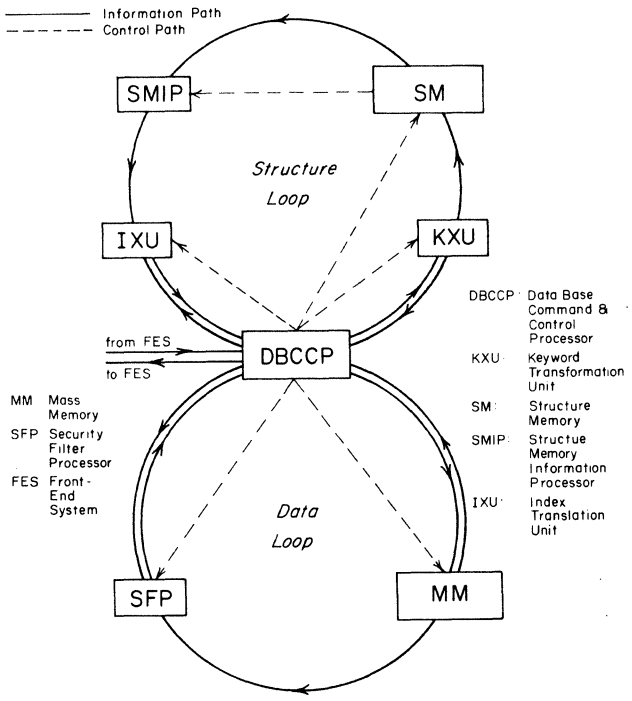


Figure 1. The Architecture of DBC.

The structure loop is composed of four components: the keyword transformation unit, the structure memory, the structure memory information processor and the index translation unit. The keyword transformation unit converts keywords into their internal representations. The primary function of the structure memory is to retrieve and update structural information of the database. This function must be performed at a rate commensurate with that of database operations performed by the components of the data loop. The concept of a partitioned content-addressable memory (PCAM) consisting of a set of processor-memory unit pairs, is used to implement the structure memory with the above properties. Powerful PCAM organizations are possible using emerging technologies. To this end, three design alternatives using three different technologies were examined. They are magnetic bubble memories and charge-coupled devices (CCDs) for the medium capacity (10^7 - 10^8 bytes) and electron beam addressable memories for the large capacity (10^9 bytes).

The structure memory information processor is responsible for performing set intersections on structural information retrieved by the structure memory. The concept of PCAMs is once again utilized to perform rapid intersection. The index translation unit is intended to decode the structural information output by the structure memory information processor.

The four components are designed to operate concurrently. The predicates of the user request are sent to the keyword transformation unit at regular intervals by the database command and control processor. The output of the keyword transformation unit which consists of coded keywords satisfying the predicates is sent to the structure memory which retrieves index terms for the keyword predicates and sends them to the structure memory information processor. The resultant output is interpreted by the index translation unit and sent to the database command and control processor. This organization of processors results in pipelining the processing functions of the structural loop components.

The data loop consists of two components, the mass memory and the security filter processor. The mass memory is the repository of the database and has a capacity of 10^{10} bytes. The design of the mass memory is based on the PCAM concept. In the mass memory, a partition of the PCAM is a cylinder of a moving-head disk unit. The cylinder is made content-addressable by incorporating track information processors (one for each track of a cylinder) for concurrent processing of the tracks of a cylinder. Furthermore, the disk read/write mechanism is modified to allow parallel readout of all the tracks of a cylinder. The cost of this modification is considerably lower than that of a monolithic associative memory implemented with fixed-head disks, CCDs, bubble memory devices or electron beam addressable memories. Such disks may soon be available from the Ampex Corporation [13].

By far the most powerful operation of the mass memory is the search and retrieve operation. The mass memory is capable of searching for and retrieving records which satisfy queries made up of keyword predicates. Because the records in the mass memory are addressable by content and carry no conventional pointers, they need no updating as long as the records exist in the database. This is true even if the security specifications (known as file sanctions) of the database change frequently.

The security filter processor provides the type B security enforcement and sorting/merging. The type B security enforcement mechanism is

provided for thoses users who do not take advantage of the type A security mechanism based on the concept of security atoms. The type A security mechanism incurs less security overhead. However, it needs the user's cooperation. First, the user must understand the security atom concept; then, the user must convey the security requirements in terms of security attributes of his data records. Keywords whose attributes appear in the security requirements are called security keywords. (A security atom is therefore a set of records all of which have the same set of security keywords). On the other hand, the type B security mechanism does not require such user cooperation. Nevertheless, posterior checking of response data against full file sanctions is an expensive undertaking. The sort mechanism enables the response data to be ordered by values of certain attributes and the merging mechanism allows new records to be formed from the response set. These are usually the ways that the user application programs would like to receive the records in the front-end computer system.

The database command and control processor regulates the operations of both the structure and data loops and interfaces with the front-end computer system. It processes all DBC commands received from the front-end computer system, schedules the execution of the commands on the basis of the command type and priority, enforces security on a selective basis, clusters records to be stored in the DBC, and routes the response data to the front-end computer system.

By indicating a property that is common to a group of records, the user may specify the group of records to be physically clustered in the mass memory. Without going into the details of the clustering process, it may suffice to say that this property is provided by the user in the form clustering attributes. Keywords whose attributes are clustering attributes are called clustering keywords. A cluster is therefore a set of records all of which have the same set of clustering keywords.

Each entry in the structure memory is a pair (K, K-list) where K is a keyword and K-list is a sequence of triples (f,c,s) where f is a cylinder number, c a cluster identifier, and s a security atom identifier. Thus, a structure memory entry identifies for all the records containing the keyword, the cylinders in which they reside, the clusters they belong to, and the security atoms with which they classify.

CREATING A RELATIONAL DATABASE

One way to represent a relational database on the DBC is to transform every relational tuple into a DBC record. Let us consider a relation as a table with a number of columns. For a tuple $(v_1, v_2, \ldots, v_n)$ of a relation R $(C_1, C_2, \ldots, C_n)$ where each C_i is a column name and each v_i is a value corresponding to column C_i, a DBC record is created with the following keywords:

<RELATION, R>
<C_1, v_1>
<C_2, v_2>
⋮
<C_n, v_n>

where the first keyword has a special attribute RELATION.

Clustering of tuples in System R requires the specification of access paths, called images and links, to be maintained on the stored relations. An image defines an ordering of the tuples of a relation with respect to one or more column values. It is therefore possible to retrieve tuples of a relation in different orders by keying on different column values. In System R, at most one image of a relation may have the clustering property; i.e., the tuples which are close to each other in the ordering of that image are stored physically near each other in the database. Tuples of one relation are linked to tuples of another relation because they have certain matching column values. Like images, links are used to establish orders among tuples of different relations. However, a link may also be declared to have a clustering property, in which case, the linked tuples will be kept close to each other.

The DBC has no use for images and links that are not designated for clustering purposes. It does not need to maintain either logical or physical ordering of DBC records. Essentially, every DBC record is content-addressable via keywords. Because of the high-volumned readout via tracks-in-parallel and the ability of the security filter processor to perform sort and merge of records, the DBC is concerned only with the number of accesses to cylinders.

Clustering of the DBC records always starts with a relation name. We first attempt to store all those DBC records that belong to the same relation in as few cylinders as possible. The reason for clustering by relation name is simply that all SEQUEL queries involve one or more relations. Therefore, by clustering primarily by relation name, it will always be ensured that the DBC will satisfy any given one-relation query by accessing no other cylinders than those required to store the records of the relation.

Secondary clustering of records is based on the clustering images and links. If there is a clustering image defined on a set of column names of a relation, then the value space of these column names is divided into rN partitions, where N is an estimate of the number of cylinders occupied by the relation and r is a positive integer factor, say, 2. Based on the values of its clustering column names, every record is then allocated a single number, called the cluster number, from the range 1 through rN. A keyword is then formed for the record as shown below:

<CLUSTER, allocated-cluster-number>

This keyword is included as part of the record and is used at the secondary level of clustering. In this way, records whose column values are close to each other will be placed in the same partition, i.e., cluster. Furthermore, it is unlikely that a

cluster will be stored in more than one cylinder because there are more clusters than cylinders.

In case a clustering link is defined on relations, a hashing method is used for determining the cluster number. The details of this process is available in [9] and will not be inlcuded here due to space limitation of this paper.

SUPPORTING A DATA SUBLANGUAGE

The DBC is a back-end machine executing commands given by a front-end computer. User data management requests are translated to DBC commands by the software in the front-end computer. The software package is termed the Relational Database Interface (RDBI). Specifically, the RDBI intercepts user requests written in the SEQUEL data sublanguage [11], translates them into a series of DBC commands, and routes the commands to the DBC. Furthermore, the RDBI handles the response set forwarded from the DBC and passed along to the user application programs. The relationship of the DBC, RDBI and user requests are depicted in Figure 2b.

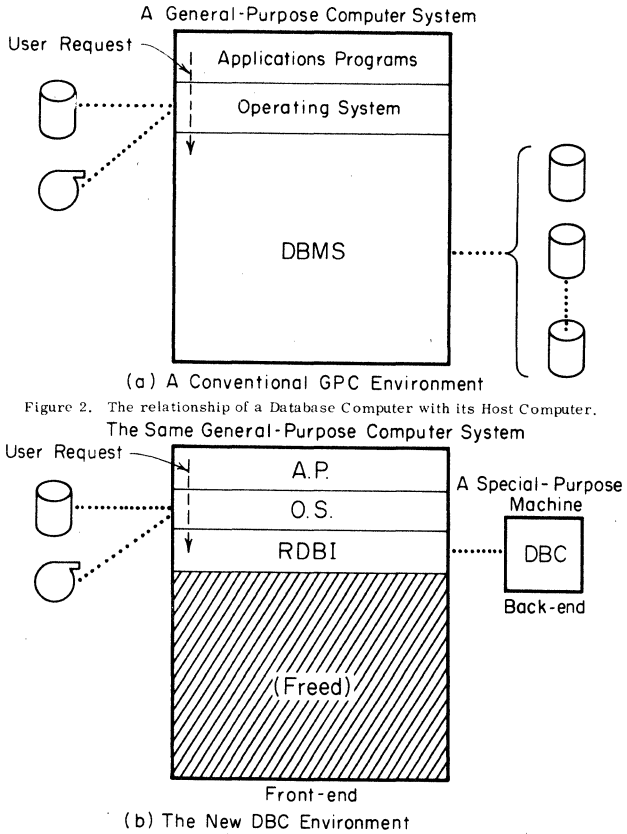


Figure 2. The relationship of a Database Computer with its Host Computer.

As an example of the translation process, we shall consider a sample relation EMP (EMPNO, NAME, DNO, JOB, MGR, SAL), where the parentheses enclose the list of column names of the relation EMP. Each employee record has an employee number, employee name, department number, job designation, manager name and salary. A SEQUEL request to find the names of employees in department 100 is as follows:

```
SELECT    NAME
FROM      EMP
WHERE     DNO = 100
```

This statement is translated by the RDBI into a single DBC command as follows:

Retrieve: (NAME)((RELATION = 'EMP') ∧ (DNO = 100)).

The command causes the keyword with attribute NAME to be output from every DBC record that satisfies the predicate conjunct ((RELATION = 'EMP') ∧ (DNO = 100)). Assuming clustering by relation EMP, the DBC requires one access to the mass memory and completes this command in one disk revolution time.

Consider again that there exists in the database another relation DEPT (DNO, DNAME, LOC). Each department record consists of the department number, department name and location. A SEQUEL request given as follows will then list the names of employees who work for departments in Columbus:

```
SELECT    NAME
FROM      EMP
WHERE     DNO IN
          SELECT DNO
          FROM   DEPT
          WHERE  LOC = 'COLUMBUS'
```

This statement is translated by the RDBI into a series of DBC commands. A command, as follows, is first sent to extract the department numbers from all DBC records containing keywords <RELATION, DEPT> and <LOC, COLUMBUS>.

RETRIEVE: (DNO)((RELATION = 'DEPT')∧(LOC='COLUMBUS'))

For every keyword <DNO, d_i> retrieved in the first step, another command is issued, as shown below, to retrieve the department names from all DBC records containing keywords <RELATION, EMP> and <DNO, d_i>.

RETRIEVE: (NAME)((RELATION = 'EMP') ∧ (DNO = 'd_i'))

If there are n departments in Columbus, then, with clustering by department numbers, the DBC can satisfy the given user request in as few as (n + 1) accesses to the mass memory. A case-by-case study of translating SEQUEL statements to DBC commands can be found in [9] and is not elaborated here.

PERFORMANCE ANALYSIS

We now attempt to conduct an analytical study of the DBC performance and to compare it against that of a conventional computer where a relational database management system (in particular, System R) is being supported. We shall call the environment consisting of the DBC, the Relational Database Interface (RDBI) and the front-end computer as the DBC environment (see Figure 2b again). A conventional system, on the other hand, consists of a general-purpose computer (GPC) which houses the database management system software (System R in this case) and executes user transactions by reading (writing) record from (to) conventional secondary storage devices. We shall call such an environment a GPC environment (see Figure 2a again). The analysis includes a study of the raw database storage requirement, the index storage requirement, and the execution time of simple queries.

The values assumed for the various parameters in this analysis are usually quite realistic. For example, the page size in a conventional system is between 1K bytes and 4K bytes. We have assumed a page size of 4,000 bytes. A disk with 20 surfaces and 30,000 bytes/track will have a

capacity of 500,000 bytes/cylinder. We have assumed a cylinder capacity of 500,000 bytes. Pointers are normally about 4 bytes long. The average length of the value parts of keywords has been assumed to be 4 bytes. This is because most search keys are either numerical (4-byte integers or floating-point numbers) or they are short alpha-numberic strings. Attribute identifiers, cluster identifiers and cylinder numbers are normally less than 4 bytes long, since 4 bytes or 32 bits can represent as many as 2^{32} such numbers in each case. In general, the choice of values for the different parameters have been so made that they may only favour the GPC environment rather than the DBC environment.

Raw Database Storage Requirement

The mass memory of the DBC stores the database records. Correspondingly, the secondary storage of a conventional relational system stores the tuples. Here we estimate the storage requirements. The following definitions are used:

- n = The relation cardinality (the number of tuples in the relation);
- d = The degree of a relation (the number of columns);
- t = The length of a tuple identifier, TID, in number of bytes;
- ℓ = The number of links defined on a relation;
- v_i = The average length in bytes of the value of the i-th column of a relation; and
- a_i = The average length in bytes of the i-th column name of a relation.

A. In the GPC Environment - In a conventional implementation of System R, every physical tuple consists of an ordered list of values (of length $\sum_d v_i$) and a pointer (TID of length t) for every link defined on the relation to which the tuple belongs. Thus, the raw database storage requirement M_g, for a given relation, is

$$M_g = n(\sum_d v_i + t\ell).$$

In case, $v_i = v$, for every i, we have

$$M_g = n(vd + t\ell).$$

B. In the DBC Environment - If the degree of the relation is d, a DBC record is composed of d attribute-value pairs where the length of an attribute is a_i and length of a value is v_i. A record also contains a special keyword with attribute RELATION to identify the relation to which it belongs. In addition, if a clustering link or a clustering image has been defined on the relation, then another special keyword with the attribute CLUSTER is also included in the record. Thus, the mass storage requirement M_d, for any given relation of n records, is

$$M_d = n \sum_{d+2} (v_i + a_i).$$

where the two special keywords are numbered (d+1)-th and (d+2)-th, respectively. The DBC assigns a fixed-length code to each attribute. Therefore $a_i = a$ for every i, and

$$M_d = n \sum_{d+2} v_i + na(d+2).$$

Further, if, for every i, $v_i = v$, we have

$$M_d = n(v+a)(d+2).$$

We now define the raw database storage ratio R_m as the ratio of the raw database storage requirements in the GPC environment to that in the DBC environment. Therefore, if $v_i = v$ for every i, we have

$$R_m = M_g/M_d = (vd + t\ell)/((v+a)(d+2)).$$

In Figure 3, we have tabulated the mass storage ratio R_m for cases where the tuple identifiers are of 4 bytes (t=4), the average length of

ℓ \ d	2	3	4	5	10
0	0.25	0.30	0.33	0.36	0.42
1	0.50	0.50	0.50	0.50	0.50
2	0.75	0.70	0.67	0.64	0.58
3		0.90	0.83	0.79	0.67
4			1.00	0.93	0.75
5				1.07	0.83

(i) v=2, t=4, a=2.

ℓ \ d	2	3	4	5	10
0	0.33	0.40	0.44	0.48	0.56
1	0.50	0.53	0.56	0.57	0.61
2	0.67	0.67	0.67	0.67	0.67
3		0.80	0.78	0.76	0.72
4			0.89	0.86	0.78
5				0.95	0.83

(ii) v=4, t=4, a=2.

ℓ \ d	2	3	4	5	10
0	0.38	0.45	0.50	0.54	0.63
1	0.50	0.55	0.58	0.61	0.67
2	0.63	0.65	0.67	0.68	0.71
3		0.75	0.75	0.75	0.75
4			0.83	0.82	0.79
5				0.89	0.83

(iii) v=6, t=4, a=2.

ℓ \ d	2	3	4	5	10
0	0.40	0.48	0.53	0.57	0.67
1	0.50	0.56	0.60	0.63	0.70
2	0.60	0.64	0.67	0.69	0.73
3		0.72	0.73	0.74	0.77
4			0.80	0.80	0.80
5				0.86	0.83

(iv) v=8, t=4, a=2.

ℓ = The number of links of a relation
d = The degree of a relation
v = The average length in bytes of the values
t = The length of the TID in bytes
a = The average length in bytes of attribute codes

Note: The more is the number of the links (ℓ); the better is raw database storage ratio for DBC.

Figure 3. Raw Database storage ratio R_m of conventional Disk Devices and DBC Mass Memory.

the attribute codes is of 2 bytes (a=2) and various v and d. Since the number of attributes in a file is small, a length of 2 bytes for attribute code, a_i, should be sufficient. The average length, v, of the value part of an attribute-value pair is varied in steps of 2, from 2 to 8. Since the number, ℓ, of links defined on a relation is not likely to exceed the number of attributes, d, (unless an attribute appears in a number of links, each connecting two relations), we may assume for practical purposes that $\ell \leq d$. Thus, we notice from Figure 3 that R_m is usually less than one. Furthermore, since the number of links defined on a relation is usually one or more, the value of R_m is likely to be greater than 0.5. That is,

$$0.5 \leq R_m \leq 1.0$$

We, therefore, conclude that the raw database

storage requirement in a DBC environment may be greater than (and even double of) the storage requirement in a GPC environment.

Index Storage Requirements

Additional storage is required for the definition and for the indices of the database. The database definition consists of the characteristics of every relation (such as relation name, degree, attribute names and types), the names and definitions of links and images, and the definition of triggers and assertions. It constitutes the conceptual view of the database. Because the definition is stored in the front-end computer, it is independent of the database machine on which the database is being created. We shall, therefore, make no further attempt to estimate the storage requirement for the views.

More important is the amount of storage occupied by the indices. The size and structure of the indices varies from one realization of the database to another, depending on the computer which supports the database. This is particularly true when one computer uses conventional location-addressed secondary storage and the other employs partitioned content-addressable memories. We shall now analyze the index storage requirement in the two different environments.

A. In the GPC Environment - For each image, System R maintains a multi-page index structure in the form of a B-tree [14]. Every value of the underlying column names or combined column names is represented in the index, thereby making it possible to determine the address of every tuple satisfying an equality predicate based on these names and name combinations. In System R, the tuples are not stored in the B-trees; instead, the TIDs pointing to the tuples are stored. The B-trees in System R are defined as follows. Each page is a node within the tree and contains an ordered sequence of index entries. For each non-leaf node, an entry consists of a pair (sort value, pointer). The pointer addresses another page in the same structure which may be either a leaf page or another non-leaf page. In either case, the target page contains entries for sort values less than or equal to the given one. For the leaf nodes, an entry is a combination of sort values along with an ascending list of TIDs for tuples having exactly those sort values. The leaf pages are chained in a doubly-linked list, so that sequential access can be supported from leaf to leaf.

To compute the storage requirement per image we use the following nomenclature:

- n = the relation cardinality (the number of tuples in the relation);
- w = the length of an internal pointer, in bytes;
- t = the length of a TID, in bytes;
- v = the average length in bytes of a value of the column name on which the image is defined. We assume that the image involves a single column name, since this is the most common case;
- s = the order of the B-tree, which is the maximum number of pointers from any node. The order depends on page size, on average key length v, and on the length w of an internal pointer;
- i = the image cardinality (the number of distinct sort values in the image); and
- b_g = page size in bytes.

We begin by computing the expected minimum number of leaf nodes in the B-tree. We then compute the order s of the B-tree. Next we compute the minimum number of non-leaf nodes, thereby completing the storage analysis.

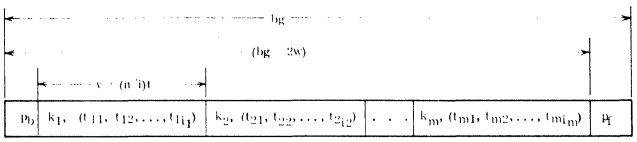
Since the average number of TIDs per sort value is n/i, we may expect

$$(b_g - 2w)/(v + (n/i)\ t)$$

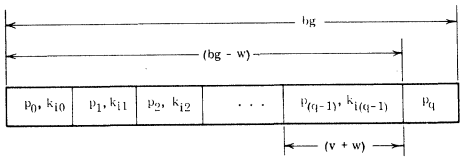
sort values per leaf-node (see Figure 4). Hence, the minimum number of leaves E is given by

$$E = \lceil i(v+(n/i)t)/(b_g - 2w)\rceil = \lceil (iv + nt)/(b_g - 2w)\rceil.$$

k (subscripted) stands for a sort value, t (subscripted) for a TID, and p (subscripted) for the address of a non-leaf node.



(a) Structure of a leaf node in the B-tree. p_0 is a backward pointer to the preceding leaf page; and p_f is a forward pointer to the next leaf page.



(b) Structure of a non-leaf node in the B-tree.

Figure 4. Nodes in a B-tree.

The order s of the B-tree, which is the maximum number of pointer fields in each non-leaf node, is given by

$$s = \lfloor (b_g - w)/(v + w)\rfloor + 1.$$

Given the values of s and E, it is not difficult to show that the minimum number of non-leaf nodes is

$$I = \lceil E/s\rceil + \lceil E/s^2\rceil + \ldots + \lceil E/s^u\rceil$$
$$\geq E(s^u - 1)/(s^{u+1} - s^u)$$

where $u = \lceil \log_s E\rceil$ is the minimum level of the B-tree. In most practical situations, the fan-out is large. Therefore, even if the depth u of the tree is small (say, 2 or 3), $s^u >> 1$. Hence,

$$I = E/(s - 1).$$

Finally, the minimum storage requirement per image D_g, is the sum of the pages for non-leaf nodes and the leaves:

$$D_g = (E + I) \text{ pages} = E(s/(s-1))b_g \text{ bytes.}$$

B. In the DBC Enviornment - Even though the RDBI maintains no directories corresponding to the

images and links defined on relations, some minimal directories are, in fact, maintained in the structure memory of the DBC. We will now try to estimate the size of such directories.

To begin with, we may recall that there are entries for only two classes of keywords: those with attribute RELATION and those with attribute CLUSTER. Since these keywords are also defined to be clustering keywords, the DBC assigns a unique cluster number to all records having the same two keywords <RELATION, r-name> and <CLUSTER, c-num>. Thus, a cluster in the DBC consists of the set of records S such that two records R1 and R2 are in S if and only if both <RELATION, r-name1> and <CLUSTER, c-num1> are in R1, both <RELATION, r-name2> and <CLUSTER, c-num2> are in R2 where r-name1 = r-name2 and c-num1 = c-num2.

A directory in the structure memory of the DBC is of the form

<keyword, (index1, index2, ..., indexh)>

where each index, in turn, takes the form (cylinder #, cluster #, security atom #1). We shall not consider the security atom #. We use the following nomenclature:

- a = the length of a (coded) attribute name in bytes;
- v = The average length in bytes of the value part of the keywords with attributes RELATION and CLUSTER; this length is expected to be smaller than the average length (denoted earlier also as v) of the value parts of a relational tuple, but we assume them to be same;
- q = The average length, in bytes, of a DBC records;
- b_d = The size of a disk cylinder, in bytes;
- c = The number of clusters of a relation (usually of the order of the number of cylinders required to store the relation);
- m = The length of a cylinder # in bytes;
- k = The length of a cluster # in bytes; and
- j = The average number of cylinders spanned by a cluster, i.e., the average number of cylinders in which there is at least one record belonging to the cluster.
- r = The ratio of the number of clusters of a relation to the number of cylinders occupied by the relation.

The number of different index terms (cylinder #, cluster #) for a relation is simply equal to cj. Since, for any given relation, there is only one directory keyword with attribute RELATION, the corresponding directory must have all the index terms for the relation. On the other hand, there are up to c directory keywords with attribute CLUSTER, and each of the corresponding entries has an average of j index terms. Thus, the directory memory requirements for a relation is given by

D_d = storage for the entry with keyword whose attribute is RELATION + storage for all entries with keywords whose attribute is CLUSTER

$$=((a+v) + cj(m+k)) + c((a+v) + j(m+k))$$
$$=(c+1)\times(a+v) + 2cj(m+k).$$

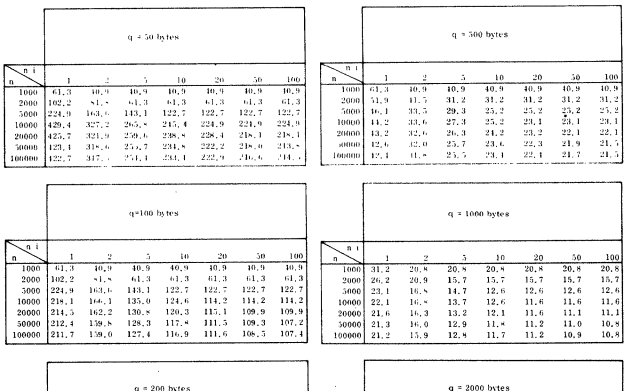
We observe that the directory memory requirement per relation, D_d, of the DBC is independent of the total number of images defined on a relation. This contrasts with the fact that in a GPC environment the storage requirement per relation is the sum total of the storage requirements for all images on a relation. If there are L images on a relation and each image requires the same space D_g, then the directory memory requirement per relation, in the GPC enviornment, is LD_g. We define the directory storage ratio R_d as the ratio of the index storage requirement in the GPC environment to that in the DBC environment. If there are L images per relation and every image is of equal size, we then have

$$R_d = LD_g/D_d$$

In the computation of D_d, the value of j, which is the number of cylinders spanned by a cluster, is a dependent parameter. It depends on the cluster size, cylinder size, loading factor of the database and the storage pattern. Through a number of simulation experiments, it was determined that in most practical situations, the value of j falls between 1 and 2.

The directory storage ratio R_d may now be computed. We assume that there is only one image per relation, i.e., L = 1. The values used for the various parameters are: a = 2 bytes, j = 2, v = k = m = t = w = 4 bytes, the page size b_g = 4000 bytes, r = 5, the cylinder size b_d = 500,000 bytes, the relation cardinality n is taken from the set {1000, 2000, 5000, 10000, 20000, 50000, 100000}, the ratio n/i is taken from the set {1, 2, 5, 10, 20, 50, 100} and the length q of a DBC record is taken from the set {50, 100, 200, 500, 1000, 2000}.

Using the fact that the number of cylinders required for a relation is $\lceil nq/b_d \rceil$ and the fact that c, the number of clusters of the relation, is r times the aforementioned number, we can now compute the directory storage ratio R_d. These calculations are tabulated in Figure 5. Observe



q = 50 bytes

n \ i	1	2	5	10	20	50	100
1000	61.3	40.9	40.9	40.9	40.9	40.9	40.9
2000	102.2	81.8	61.3	61.3	61.3	61.3	61.3
5000	224.9	163.6	143.1	122.7	122.7	122.7	122.7
10000	429.4	327.2	265.6	245.4	224.9	224.9	224.9
20000	425.7	321.9	250.6	238.8	228.1	218.1	218.1
50000	423.1	317.8	254.7	234.8	222.2	218.0	213.8
100000	422.7	317.4	254.1	234.1	222.9	210.6	214.4

q = 500 bytes

n \ i	1	2	5	10	20	50	100
1000	61.3	40.9	40.9	40.9	40.9	40.9	40.9
2000	51.9	41.7	31.2	31.2	31.2	31.2	31.2
5000	16.1	33.5	29.3	25.2	25.2	25.2	25.2
10000	11.2	33.6	27.3	25.2	23.1	23.1	23.1
20000	13.2	32.6	26.3	24.2	23.2	22.1	22.1
50000	12.6	32.0	25.7	23.6	22.3	21.9	21.7
100000	12.4	31.8	25.5	23.4	22.1	21.7	21.5

q=100 bytes

n \ i	1	2	5	10	20	50	100
1000	61.3	40.9	40.9	40.9	40.9	40.9	40.9
2000	102.2	81.8	61.3	61.3	61.3	61.3	61.3
5000	224.9	163.6	143.1	122.7	122.7	122.7	122.7
10000	218.1	166.1	135.0	124.6	114.2	114.2	114.2
20000	214.5	162.2	130.8	120.3	115.1	109.9	109.9
50000	212.4	159.8	128.3	117.8	111.5	109.3	107.2
100000	211.7	159.0	127.4	116.9	111.6	108.5	107.4

q = 1000 bytes

n \ i	1	2	5	10	20	50	100
1000	31.2	20.8	20.8	20.8	20.8	20.8	20.8
2000	26.2	20.9	15.7	15.7	15.7	15.7	15.7
5000	23.1	16.8	14.7	12.6	12.6	12.6	12.6
10000	22.1	16.8	13.7	12.6	11.6	11.6	11.6
20000	21.6	16.3	13.2	12.1	11.6	11.1	11.1
50000	21.3	16.0	12.9	11.8	11.2	11.0	10.8
100000	21.2	15.9	12.8	11.7	11.2	10.9	10.8

q = 200 bytes

n \ i	1	2	5	10	20	50	100
1000	61.3	40.9	40.9	40.9	40.9	40.9	40.9
2000	102.2	81.8	61.3	61.3	61.3	61.3	61.3
5000	114.2	83.1	72.7	62.3	62.3	62.3	62.3
10000	109.9	83.7	68.0	62.8	57.6	57.6	57.6
20000	107.7	81.4	65.7	60.4	57.8	55.2	55.2
50000	106.4	80.0	64.2	59.0	55.8	54.8	53.7
100000	105.9	79.6	63.8	58.5	55.9	54.3	53.7

q = 2000 bytes

n \ i	1	2	5	10	20	50	100
1000	15.7	10.5	10.5	10.5	10.5	10.5	10.5
2000	13.1	10.5	7.9	7.9	7.9	7.9	7.9
5000	11.6	8.4	7.4	6.3	6.3	6.3	6.3
10000	11.1	8.4	6.9	6.3	5.8	5.8	5.8
20000	10.8	8.2	6.6	6.1	5.8	5.5	5.5
50000	10.7	8.0	6.4	5.9	5.6	5.5	5.4
100000	10.6	8.0	6.4	5.9	5.6	5.4	5.4

Figure 5. Directory storage ratio R_d for single image per relation.

that other parameters remaining unchanged, after the number of records, n, has reached a high enough value, further increase in n does not have much effect, since both D_g and D_d tend to increase proprotionately with large n. Further, we observe that as DBC record length increases, fewer and fewer records are accommodated in a cylinder, thereby increasing the number of index terms and hence the storage ratio R_d.

We notice that for a reasonable record length between 100 and 1000 bytes, the DBC directory memory requirement lies between 0.05% and 10% of that of a conventional system. Furthermore, if there are more than one image per relation (which is often the case), then the directory memory requirement in a GPC environment increases proportionately with the number of images. The DBC directory memory requirement, in contrast, remains steady.

Query Execution Time

Query execution time is perhaps the single most important measure of performance of a database management system. For SEQUEL statements being handled in a conventional GPC environment, the system first parses the statements and then uses an optimizer to determine a good access strategy from among a number of possible access strategies. In addition to the parsing and optimization times, the execution time of a query consists mainly of

(1) the time to access a number of index pages and search their contents to determine a list of eligible TIDs,
(2) the time to access a number of data pages in order to fetch the eligible tuples, and
(3) the CPU time to determine the final response set form the list of eligible tuples.

For a given query, a single predicate of a predicate conjunct in the query may be used for determining the eligible TIDs. After the corresponding tuples are retrieved, they are placed in the final response set only if they satisfy all the other predicates in the predicate conjunct. In a DBC environment, the execution time of a query consists mainly of

(1) hardware search time of the structure memory to determine the eligible cylinders, and
(2) the time to search each eligible cylinder for records satisfying a predicate conjunct.

We make the following practical assumptions for the analysis:

(1) For every cylinder accessed by the DBC, we include an extra processing time for the structure memory to determine the index terms and compute the cylinder numbers. Therefore, a constant factor K (>1) will be used to multiply the number of accesses to the mass memory, thereby accounting for query processing time in the structure memory.
(2) Binary search of the index pages in a GPC environment takes a negligible amount of time compared to the time to acces each page.
(3) The time to access an index page, the time to access a data page in the GPC environment and the time to access a cylinder of the DBC are all equal to the latency time plus rotation time needed to access a disk cylinder.

In the ensuing discussion, we consider a common type of queries, namely, the single-relation queries. The analysis for the more complex two-relation queries may be found in [9] and are not considered in this paper. We may point out, however, that the advantages in using the DBC for such complex queries are similar in magnitude as for the single-relation queries. The following analysis is in the style of [10], and the time to execute a query is determined in terms of the number of accesses to the physical blocks.

A single-relation query is exemplified by the following SEQUEL statement which lists the names and salaries of programmers who earn more than $10,000:

```
SELECT   NAME,SAL
FROM     EMP
WHERE    JOB = 'PROGRAMMER'
AND      SAL > 10000
```

This is an example of a query with a single predicate conjunct. In general, a query may be a disjunction of X predicate conjuncts. The query may then be treated as X queries each with a single predicate conjunct. We, therefore, only restrict ourselves to queries with a single predicate conjunct. Furthermore, the predicates are assumed to be involved with simple comparisons of a field with a value so that they can be matched with an image. More complicated predicates, such as EMP X.MGR = EMP Y.EMPNO, cannot be matched by an image. Finally, since the consideration of links involves a straightforward extension of the analysis given below, we will consider images only.

The following notations are introduced to simplify the ensuing discussion:

n = the relation cardinality;
p = the number of predicates in the query;
h = the coefficient of CPU time (1/h is the number of tuple comparisons which are considered equivalent in cost to one page access);
i = the image cardinality;
K = the coefficient of DBC's structure memory processing time, the time required to determine index terms (K>1);
f = the number of index-page accesses per index search in the GPC environment (For a given storage device and given key length, it is a function of the relation cardinality n and the image cardinality i. Normally, it has a value lying between 2 and 4);
B_g = the average number of tuples (of a relation) per data page (subscript g refers to the GPC environment);
B_d = the average number of records per DBC

cylinder (subscript d refers to the DBC environment); and

j = the average number of cylinders spanned by a cluster in the DBC.

The optimizer in System R has the option to select an access strategy among a variety of choices. The most important of these are listed below. In each case, the execution-time ratio R_t may be determined by computing the ratio of the time T_g required to execute a query in the GPC environment to the time T_d required in the DBC environment.

Option 1. The attribute of a given predicate is identical to the column name for which a clustering image has been created. Furthermore, the predicate is an equality predicate.

Since the expected number of tuples that satisfy the predicate is n/i, the expected number of data pages to be accessed in the GPC environment is $\lceil n/(iB_g) \rceil$. Since each of the retrieved tuples must now be compared against the other (p-1) predicates, the total time required in the GPC environment is

$$T_g = \underbrace{\lceil n/(iB_g) \rceil}_{\text{I/O time for data pages}} + \underbrace{(p-1)hn/i}_{\text{CPU time}} + \underbrace{f}_{\text{I/O time for the indices}}$$

T_g may actually be somewhat less because some of the retrieved tuples may have to be eliminated from further consideration by the successive comparison with other predicates. Furthermore, since the number of tuples retrieved, which is n/i, is expected to be very small, we may even neglect the CPU time required for comparing predicates. Therefore, T_g is simplified to

$$T_g = \lceil n/(iB_g) \rceil + f.$$

In the DBC environment, whenever the equality predicate matches a clustering image, only one cluster need be searched. Therefore,

$$T_d = jK$$

where the factor K accounts for the structure memory processing time. Finally, the execution-time ratio is

$$R_t = T_g/T_d = \lceil n/(iB_g jK) \rceil + f/(jK).$$

Option 2. The column name for which a clustering image is in existence matches the attribute of a predicate which is not an equality predicate. Assuming that half the tuples of the relation satisfy the predicate, the expected times are

$$T_g = \lceil n/(2B_g) \rceil + (p-1)hn/2 + f$$

and

$$T_d = \lceil nK/(2B_d) \rceil.$$

Option 3. A non-clustering image is available. The column name for which the image has been created matches the attribute of an equality predicate. If this image is used in a GPC environment, then one page access will be required for each of the n/i expected tuples that satisfy the predicate. Without the advantage of secondary clustering information (in the query), the DBC has to access the entire relation. Therefore,

$$T_g = \lceil n/i \rceil + (p-1)hn/i + f$$

and

$$T_d = \lceil nK/B_d \rceil.$$

Option 4. A non-clustering image is available for the attibute of a non-equality predicate. If this image is used in the GPC environment, then

$$T_g = \lceil n/2 \rceil + (p-1)hn/2 + f$$

and

$$T_d = \lceil nK/B_d \rceil.$$

Option 5. A clustering image is used. The column name for which the clustering image exists matches no attribute of any predicate. Even though no attribute of the given predicate matches any column names on which images are formed, we pick a clustering image for accessing the tuples (because these tuples are next to each other). In this case, all the tuples must be examined in the GPC environment. Therefore,

$$T_g = \lceil n/B_g \rceil + phn + f$$

and

$$T_d = \lceil nK/B_d \rceil.$$

Option 6. A non-clustering image is used which matches no attribute of any predicate. Since, we may justifiably assume that every relation has a clustering image (or clustering link), this choice will actually never have to be made in a GPC environment. In any case, if the choice were indeed to be made, then

$$T_g = n + phn + f$$

and

$$T_d = \lceil nK/B_d \rceil.$$

Option 7. Suppose there are $p_e \geq 1$ equality predicates and $p_n \geq 1$ non-equality predicates each of which has a matching image, then the $(p_e + p_n)$ images may be searched. A TID list is generated for each predicate. These lists may be sorted separately and then intersected to determine the final TID list to be searched. We then have,

$$T_g = (n/(i^{p_e} 2^{p_n}) + (p_e + p_n)f.$$

We have neglected the predicate comparison time, since the final list of TIDs will be very small; we have also neglected the time to sort the TID lists, which may be appreciable if the lists are long. Notice that when $p_e \geq 2$, the first term in T_g (i.e., the number of tuples which satisfy all of the p_e equality predicates) is likely to be quite small. In such a case, we may write

$$T_g = (p_e + p_n)f.$$

In the DBC environment, we have

T_d = jK, if an equality predicate matches a clustering image; and

$T_d = \lceil nK/B_d \rceil$, otherwise.

In Figure 6, we have tabulated the values of execution time ratio R_t for each of the seven options mentioned above. We have used the following figures: K = 1.2, f = 3, p = 2, h = 0.0001, j = 2, B_d/B_g = 50, B_g is taken from the set {3, 20, 100, 500}, the ratio n/i is taken from the set {1, 2, 5, 10, 50, 100}, n is taken from the set {1000, 5000, 20000, 100000} and $(p_e + p_n)$ is taken from the set {2, 3}.

The assumption of B_d/B_g = 50 requires a little explanation. A disk cylinder normally consists of 20 to 40 tracks. The track size to page size ratio in a conventional system usually varies from 1 to 5. Finally, the size of a DBC record varies from 1 to 2 times the size of the corresponding tuple of a conventional system. Taking these factors into consideration, we have arrived at a reasonable figure of 50 for the ratio B_d/B_g.

We observe a number of important facts from the tables in Figure 6. Whenever there is an equality predicate matching an image (e.g., Options 1 and 3), very few pages need to be searched in the GPC environment, because of the choice of large image cardinalities. Therefore, in these cases the index search time f dominates the query execution time T_g. In Option 1, the DBC has to search only one cluster because the equality predicate matches a clustering image. In Option 3, however, the DBC has to content-search the entire relation. So, for very large relations and very large records, the GPC environment is clearly more favorable in Option 3. Similar reasoning holds for Option 7 if the clustering image does not match even one of the equality predicates. In all other cases, the DBC performs one or more orders of magnitude better than a conventicnal system. In short, the DBC works much better than a conventional system, whenever <u>any one</u> of the following holds:

(1) The record size is small, say 50 to 200 bytes.
(2) The relation is of small or medium size, say less than 20,000 tuples.
(3) Many records (say, greater than 50) are satisfied by an equality predicate, so that many records have to be retrieved by either system.
(4) The image cardinality is medium, say, n/i > 100, which is typical for large relations. This observation actually follows from (3).
(5) A given query does not have any equality predicate that matches an image.

The GPC environment, in contrast, works out as good or better than the DBC only when <u>all</u> the following conditions hold:

(1) The relation is large, say greater than 20,000 tuples.
(2) The records are large, say 500 bytes or larger.
(3) The query has an equality predicate that matches an image.
(4) The cardinality of the above image is very large, say n/i > 10.

Option 1

n/i \ B_g	3	20	100	500
1	1.67	1.67	1.67	1.67
2	1.67	1.67	1.67	1.67
5	1.67	1.67	1.67	1.67
10	2.08	1.67	1.67	1.67
50	5.42	2.50	1.67	1.67
100	9.58	3.33	1.67	1.67

Option 2

n \ B_g	3	20	100	500
1000	34.35	28.05	8.05	1.05
5000	41.94	42.75	28.25	8.25
20000	41.75	42.00	34.67	24.00
100000	41.70	41.82	42.33	36.00

Option 3 where n/i = 1

n \ B_d	250	1000	5000	25000
1000	0.80	2.00	4.00	4.00
5000	0.17	0.67	2.00	4.00
20000	0.04	0.17	0.80	4.00
100000	0.01	0.03	0.17	0.80

Option 3 where n/i = 2

n \ B_d	250	1000	5000	25000
1000	1.00	2.50	5.00	5.00
5000	0.21	0.83	2.50	5.00
20000	0.05	0.21	1.00	5.00
100000	0.01	0.04	0.21	1.00

Option 3 where n/i = 5

n \ B_d	250	1000	5000	25000
1000	1.60	4.00	8.00	8.00
5000	0.33	1.33	4.00	8.00
20000	0.08	0.33	1.60	8.00
100000	0.02	0.07	0.33	1.60

Option 3 where n/i = 10

n \ B_d	250	1000	5000	25000
1000	2.60	6.50	13.00	13.00
5000	0.54	2.17	6.50	13.00
20000	0.14	0.54	2.60	13.00
100000	0.03	0.11	0.54	2.60

Option 3 where n/i = 50

n \ B_d	250	1000	5000	25000
1000	10.60	26.50	53.01	53.01
5000	2.21	8.83	26.50	53.01
20000	0.55	2.21	10.60	53.01
100000	0.11	0.44	2.21	10.60

Option 3 where n/i = 100

n \ B_d	250	1000	5000	25000
1000	20.60	51.51	103.01	103.01
5000	4.29	17.17	51.51	103.01
20000	1.07	4.29	20.60	103.01
100000	0.21	0.86	4.29	20.60

Option 4

n \ B_d	250	1000	5000	25000
1000	100.61	251.52	503.05	503.05
5000	104.34	417.38	1252.13	2504.25
20000	104.22	416.88	2001.00	10005.00
100000	104.19	416.74	2083.71	10001.80

Option 5

n \ B_g	3	20	100	500
1000	40.64	26.60	13.20	5.20
5000	41.83	42.33	27.00	14.00
20000	41.75	41.96	41.40	47.00
100000	41.72	41.87	42.63	44.60

Option 6

n \ B_d	250	1000	5000	25000
1000	200.64	501.60	1003.20	1003.20
5000	208.50	834.00	2502.00	5004.00
20000	208.41	833.63	4001.40	20007.00
100000	208.38	833.53	4167.63	20004.60

Option 7, where $p_e + p_n = 2$, no clustering image

n \ B_d	250	1000	5000	25000
1000	1.20	3.00	6.00	6.00
5000	0.25	1.00	3.00	6.00
20000	0.06	0.25	1.20	6.00
100000	0.01	0.05	0.25	1.20

Option 7, where $p_e + p_n = 3$, no clustering image

n \ B_d	250	1000	5000	25000
1000	1.80	4.50	9.00	9.00
5000	0.38	1.50	4.50	9.00
20000	0.09	0.38	1.80	9.00
100000	0.02	0.08	0.38	1.80

Note: If the clustering image matches an equality predicate of the query in Option 7, then R_t is a constant 2.50 when $p_e + p_n = 2$ and R_t is 3.75 when $p_e + p_n = 3$.

Figure 6. Execution time ratio R_t for single-relation queries.

It may be mentioned that a relational database management system like System R will expend a considerable amount of resources in storing and executing the query optimizer. Furthermore, as a large software package, the optimizer requires a considerable amount of CPU time for execution. In a DBC environment, this software is unnecessary. A simple examination of a SEQUEL query indicates whether one of its predicates matches the clustering image (or link) of the associated relation. If such an image (or link) is available, then the RDBI prepares a DBC query that includes a predicate with the attribute CLUSTER. Thus, elaborate query optimization in the GPC environment is replaced by a simple decision in the DBC environment.

Execution of update requests by the DBC is not as efficient as the execution of retrieval requests. Insertions and modifications are normally requested a record at a time. Therefore, the savings achieved by using the DBC is possibly a few accesses to the indices (images or links) that may need to be updated in a conventional system. This is because there are usually fewer attributes on which directories are maintained in the DBC. Furthermore, directory accesses are at least partially overlapped with mass memory accesses. Deletions, however, may be requested in terms of groups of records. For example, it may be required to delete from the database all records belonging to DNO = 100 . Such requests are executed by the DBC's mass memory with as good a performance as is achieved for retrieval requests. Corresponding updates to the structure memory may be done in a relatively slack (low-activity) period by making use of a look-aside buffer [4]. Overall, the performance of the DBC is adequate in the execution of update requests. But the gains are accentuated in a retrieval-intensive operating environment with, say, more than 50% of all requests being retrieval requests.

CONCLUDING REMARKS

A performance analysis was done in this paper in which we have compared the storage requirements and query execution times of a relational system being supported on a conventional computer versus the same system being supported on the DBC. It has been observed that while the mass memory requirement in the DBC is usually between one and two times the requirement in a conventional system, there is a tremendous saving in index storage and very large reductions in the execution time of queries when a DBC is being used. Specifically, the usual directory memory requirement and query execution times are likely to be one or more orders of magnitude better than those of a conventional system. The reason for this performance enhancement lies in the very large block size of the DBC's on-line mass memory, the content-addressability of each block, and the clustering of DBC records primarily by relation names.

Because of the very large block size, directories are small. Every mass memory access allows the DBC to inspect a very large number of records. Because of the content-addressability of each block, the response set of a query is usually the same set of records returned by the mass memory. Therefore, no additional CPU time is needed to compare the retrieved records against other predicates that form the same query. Clustering of all records belonging to any given relation ensures that any single-relation query, whatever its composition, will require at most as many mass memory accesses as there are blocks occupied by the relation.

Further speed gains, which do not show up in the analysis, may follow. They are due to the various other functional features of the DBC such as hardware sorting, automatic memory management, and hardware to compute the common set-functions such as average, maximum, minimum and sum. All these seem to indicate that a very favorable level of performance can be achieved with the use of database machines.

REFERENCES

[1] Baum, R. I. and Hsiao, D. K., "Database Computers - A Step Towards Data Utilities", IEEE Transactions on Computers, C-25, 12, December 1976.

[2] Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer - A Summary", Proc. Third Workshop on Computer Architecture for Non-numerical Processing, Syracuse, N.Y., May 17-18, 1977.

[3] Baum, R. I., Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer - Part I: Concepts and Capabilities", Tech. Rep. No. OSU-CISRC-TR-76-1, Sept. 1976, The Ohio State University, (ADAO 34154).

[4] Hsiao, D. K., Kannan, K. and Kerr, D. S., "Structure Memory Designs for a Database Computer", Proc. Nat. ACM Conf., ACM, N.Y., 1977. Also available as (ADAO 35178).

[5] Kannan, K., Hsiao, D. K., and Kerr, D. S., "A Microprogrammed Keyword Transformation Unit for a Database Computer", Proc. Tenth Annual Workshop on Microprogramming, New York, October 1977.

[6] Kannan, K., "The Design of a Mass Memory for a Database Computer", Proceedings of the 5th Annual Symposium on Computer Architecture, Palo Alto, Calif., April 1978. Also available as (ADAO 36217).

[7] Hsiao, D. K., Kerr, D. S., and Ng, F. K., "DBC Software Requirements for Supporting Hierarchical Databases", Tech. Rep. No. OSU-CISRC-TR-77-1, April 1977, The Ohio State University (ADAO 39038).

[8] Banerjee, J., Hsiao, D. K., and Kerr, D. S., "DBC Software Requirements for Supporting Network Databases", Tech. Rep. No. OSU-CISRC-TR-77-4, June 1977, The Ohio State University, (ADAO 41651).

[9] Banerjee, J. and Hsiao, D. K., " DBC Software Requirements for Supporting Relational Databases", Tech. Rep. No. OSU-CISRC-TR-77-7, November 1977, The Ohio State University, (ADAO 49180).

[10] Astrahan, M. M., et al., "System R: Relational Approach to Database Management", ACM Trans. on Database Systems, 1, 2, June 1976, 97-137.

[11] Chamberlin, D. D., et al., "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Cotnrol", IBM Journal of Research and Development, 20, 6, November 1976, 560-575.

[12] Ozkarahan, E. A., Schuster, S. A. and Sevick, K. C., "Performance Evaluation of a Relational Associative Processor", ACM Trans. on Database Systems, 2, 2, June 1977, 175-195.

[13] Ampex Corporation, "DM-PTD Parallel Transfer Drive Engineering SPecifcation", Dwg. No. 3308829-01, Issue 2, Sept. 1977.

[14] Bayer, R., McCreight, E., "Optimization and Maintenance of Large Ordered Indexes", ACTA Informatica, 1, 3, 1972, 173-189.

A METHODOLOGY FOR SUPPORTING EXISTING CODASYL DATABASES WITH NEW DATABASE MACHINES*

Jayanta Banerjee and David K. Hsiao

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

In this paper, an attempt is made to show that conventional database management system software, in particular those of CODASYL type, can be effectively replaced by database machines with good performance. The replacement of CODASYL system software involves two main steps: (1) In order to preserve the notions of CODASYL records, sets, areas, and others, we need a methodology for database transformation so that an existing CODASYL database may be transformed into suitable formats for storing and retrieving in the database machine. (2) For the purpose of allowing existing application programs written in a CODASYL data sublanguage to store, retrieve and manipulate CODASYL data in the new environment without reprogramming, we need to be able to translate the data sublanguage calls dynamically into the commands of the database machine. Such process is termed query translation.

In this paper, a database transformation methodology and a query translation process are presented which ensure that the content-addressability and parallel read-out capability of the database machine are used to advantage. The machine in consideration is known as the database computer (DBC) and is also briefly reviewed. DBC is one of the 'typical' new back-end machines for database management which utilize the emerging hardware and the modification of existing hardware for performance gain and capacity increase.

Key words: CODASYL data model; database machines; database management systems; database transformation; DBC; network data model; query translation; relative performance.

1. Background

Conventional computers are not specifically designed for database management tasks; they require large and complex software to carry out such tasks. Consequently, both the performance and reliability of the computer system suffer considerably. Large databases of the future are likely to be managed by special-purpose database machines, instead of by conventional database management software running on general-purpose computers [1], [2]. Database machines which are made of specialized hardware for database management tasks may contribute to appreciable performance improvement and system reliability, since the conventional database management software is mostly eliminated and its underlying hardware is being relieved for more conventional tasks. In addition, the elimination of the conventional database management software and relief of the conventional hardware coupled with the recent advances and price

*The work reported herein is supported by the Office of Naval Research through contract N00014-75-C-0573.

reductions in memory and processor technologies may allow special-purpose database machines to compete with conventional software and hardware cost-effectively.

Database machines are characterized by their capability of providing very large on-line database stores (say, 10^{10} bytes and beyond), high-volume processing (so that even for a small amount of results a large amount of related data can be processed readily), and block-oriented content-addressability (where data are searched, retrieved and updated by content in response to predicates). DBC is a database machine with the aforementioned basic characteristics. For very large storage capacity, DBC utilizes moving-head disks. To achieve high-volume processing, the moving-head disks are modified to allow parallel read-out of all the tracks of a cylinder in one disk revolution. To provide content-addressability, the disk controllers are modified to incorporate microprocessors, one for each track of a cylinder. Furthermore, both the parallel reading of the tracks of a cylinder and content-addressing of the tracks of the same cylinder can be accomplished in one disk revolution time. DBC also employs new and existing technologies for other components. The use of charge-coupled devices (CCDs) or bubble-domain memories for a structure memory is such an instance. The structure memory maintains indices and security-related information about the database. With this information in the structure memory, DBC can restrict the content-addressable search for and high-volume processing of the authorized data to a few cylinders, instead of the entire database store.

For a detailed exposition of DBC concepts as well as DBC architecture and design, the reader may refer to [3], [4], [5], [6], [7]. In this paper, we will briefly discuss those components of the DBC which are necessary in the study of the DBC's capability in supporting CODASYL databases.

2. The Database Computer (DBC)

DBC is a back-end database machine for very large on-line databases. As illustrated in Figure 1, database application programs are still residing in a front-end general-purpose computer. The operating system of the front-end computer continues to coordinate the execution of these programs. However, the traditional database management system is absent from the front-end computer and is replaced by an interface. Data management calls are now relegated by the operating system to the interface, known as DBI, which is a small software package keeping account of these calls, translating these calls to DBC commands, and routing these commands to DBC. The discussion of the interface will be expounded in Section 5. Finally, the database is stored in DBC, instead of traditional secondary storage such as disks.

2.1. A Brief Look at the DBC Organization

DBC stores its entire database in an on-line secondary storage known as the mass memory (MM) as depicted in Figure 1. The mass memory is made of moving-head disks modified to provide tracks-in-parallel read-out and has enough logic to content-address any cylinder in one disk revolution time.

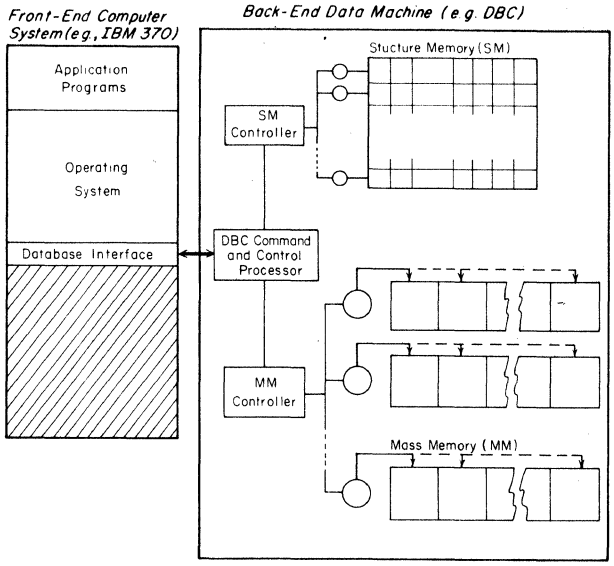


Figure 1. Basic Organization and Operating Environment of DBC

Since the mass memory has logic to content-address only one cylinder at a time, it is necessary that some directories be maintained of the information content of the database. These directories are expected to be much smaller than conventional indices because (1) an address in these directories is a cylinder number, rather than a tuple consisting of cylinder number, track number, page number, and record offset; (2) very few directory entries need be maintained, since cylinders are very large (0.5Mbytes per cylinder) compared to tracks or pages; and (3) proper clustering of data can place related data in very few cylinders. DBC provides a hardware clustering mechanism for such purpose. The directories, as well as security-related information (which we shall not discuss in this paper), are stored in the structure memory made of either CCDs or bubble-domain memories. The structure memory (SM), like the mass memory, is also block content-addressable. However, it has smaller capacity (1% of mass memory size), smaller blocks (up to 5K bytes per block), and higher speed (20 times faster than the block access rate of the mass memory).

The DBC command and control processor is in charge of communicating with the software interface of the front-end system. The interface translates a database management call to one or more DBC commands. Given a command from the front-end, the DBC command and control processor decodes it, determines the cylinders to be searched in order to satisfy it by referencing the structure memory, issues appropriate orders to the mass memory, and transfers response data back to the front-end system. The DBC command and control processor also coordinates all activities of DBC, including data clustering.

2.2. Data Representation

Data is stored and manipulated in DBC as collections of records. A record consists of a record body and a set of variable-length attribute-value pairs, where the attribute may represent the type, quality or characteristic of the value. The record body is composed of a (possibly empty) string of characters which are not used for search purposes. For logical reasons, all the attributes in a record are required to be distinct. An example of a record is shown below:

(<TYPE, EMP>, <JOB, MGR>, <DEPT, TOY>, <FLOOR, 4>).

This record consists of four attribute-value pairs. The value of the attribute JOB, for instance, is MGR. Attribute-value pairs are called keywords, since they characterize records and may be used as 'keys' in a search operation. Keywords for which directory entries are maintained in the structure memory are called type-D keywords. Records can be grouped into files on the basis of ownership, security and other purposes.

DBC interfaces with front-end systems by accepting a large repertoire of high-level database management commands, by delivering collections of records or portions of records as responses, and by indicating successful or unsuccessful execution of the commands. Some of the commands, called record access commands, may be used for specifying a collection of records in the database and for carrying out an intended operation on these records, such as retrieval, deletion and modification. Other commands may be used for database loading, record insertion, initialization, etc.

An important feature of record access commands is that they allow natural expressions for specifying a record collection. A record collection may be specified in terms of a keyword predicate, or simply, a predicate, which is a triple consisting of an attribute, a relational operator (such as, $=$, $\neq$, $>$, $\geq$, $<$, $\leq$) and a value. For instance, (SALARY > 10000) is a predicate. A record collection may also be specified in terms of a conjunction of predicates, called a query conjunction. Finally, a record collection may be specified in terms of a disjunction of query conjunctions, called a query.

Certain attributes of a file may be designated by the file creator as clustering attributes. Correspondingly, keywords having clustering attributes are called clustering keywords. By clustering the data, a query can be satisfied in as few disk revolutions as there are query conjunctions of the query.

3. The CODASYL Databases

The CODASYL databases are of the network type as documented originally in the DBTG report [8]. In this section, we shall extract some of the important data definition and manipulation facilities specified in the DBTG report for discussion.

3.1. Data Definition Facilities

The CODASYL record is similar to a COBOL record. A record type (or record name) is defined as a collection of hierarchically related data item names or field names. The hierarchy of field names is specified by a template. Any occurrence of the record type, or simply a record, will have specific values for these data items. Thus, a record type or record name is a generic name for all the record occurrences that have the same template.

Relationships between records are indicated through set types. A set type consists of a single record type called the owner record type and one or more other record types called the member record types. Record occurrences of the owner record type are termed owners and of the member record types members. Thus, a set type asserts the existence of associations between records of heterogeneous types in the database. This allows the designer to interrelate diverse record types and to associate various entities in the database into a network-like model of real-world database management problems. It should be emphasized at this point that the owner record of a set type is prohibited from being one of the member records of the same set type.

As in a record type, a set type also has occurrences. Each occurrence of a set type must contain one occurrence of the owner record type and a number of occurrences of each of its member record types. All the occurrences of a set type are pairwise disjoint. In other words, a record occurrence cannot appear in two different occurrences of the same set type.

The database may be divided into logical subdivisions called areas. Each record occurrence is placed in only one area. This subdivision may be done on the basis of frequency of record access, security requirement and physical clustering needs. When a record occurrence is first stored in the database, it is assigned a database key. A database key is a unique identifier of a record occurrence. Thus, instead of physical addresses, database keys may be used as pointers. For each application program (also referred to as a run-unit) a table of currency status indicators must be maintained. These indicators are actually database keys identifying for each of the following the most recently accessed record occurrence:

(1) current record occurrence of the run-unit,
(2) current record occurrence of each area,
(3) current record occurrence of each set type, and
(4) current record occurrence of each record type.

Most data manipulation statements refer to the current record occurrence of the run-unit. Others are with respect to the current record occurrence of an area, set or record type. Insertion of a member record occurrence into a set type often requires the selection of a set occurrence, which may be the current set occurrence.

The location mode of a record type determines the DBTG strategy to be used for initial record placement in the database when a new record occurrence is being stored. The location mode of a record is said to be direct when the user is allowed the facility of specifying the database key for

each record occurrence stored in the database. The record occurrence may then be placed in an area and location determined by the database key. If the location mode is defined to be calc, then the database key for a record occurrence is computed by a procedure that uses some combination of data items within the record as arguments. The data items are called calc keys. The location mode of a record may also be declared to be via a set (type) in which the record type is a member record type. In this case, the CODASYL system first selects an occurrence of that set type based on its set occurrence selection criterion. It then uses the ordering policy of the set type to determine the logical position of the record within the set type. Finally, it places the record occurrence in such a position that adjacent record occurrences within a set occurrence are physically "close" or clustered.

The order in which member records are inserted in a set occurrence may be either in a sorted order or in an order dependent on the time sequence in which they are inserted and the position of the current record in the set. A member record may be inserted first (or last) meaning that its position in the set is next (or prior) to the owner record of the current set occurrence. If the ordering of a set is declared to be next (or prior), then any member record occurrence will be inserted in a position next (or prior) to the logical position of the current record of the set type. Finally, a set type can be ordered by some data items.

A CODASYL system may be asked to select a set occurrence among all the occurrences of a set type. Automatic set occurrence selection is necessary when the system is required to find a set occurrence in which a member record occurrence is to be inserted or found. Therefore, there is a set occurrence selection method defined for each member record type and one for the owner record type.

3.2. Data Manipulation Facilities

The user writes his programs using a general-purpose language that hosts the CODASYL data manipulation language (DML). DML facilitates operations on set types, usually by 'navigating' through their set occurrences. The starting point of most DML statements is the current record of the run-unit. Others can be based on the current record occurrence of a set type, area or record type. A find statement may be used in order to establish a record occurrence as the current record of the run-unit and also optionally as the current record of an area, record type or set type. The delete statement may be used to delete the current record occurrence of the run-unit and to delete also the mandatory members of all set types in which the deleted record is an owner. The get statement retrieves the current record of the run-unit and places it in the user working area. A store statement is used in order to place a new record occurrence in the database. To manually insert record occurrences into set types, the insert statement is employed; and any optional member record occurrence may be removed from a set type by using the remove statement. To modify the values of data items in a record occurrence, the modify statement is used, which may also change the membership of the record occurrence from one set occurrence to another (of the same set type).

4. Database Transformation

An existing CODASYL database may be supported on DBC by converting the database to conform to the DBC representation of data. This one-time conversion is known as database transformation. Existing database management applications need not be reprogrammed. Instead, a database interface software (DBI) residing in the front-end computer will translate in real-time the data management calls into DBC commands. This process is known as query translation. Database transformation is the subject of our discussion in this section and query translation will be the subject of our discussion in the next section.

In representing a CODASYL database on DBC, our first goal is to preserve the original information such that all operations previously performed on the CODASYL database may still be performed on the DBC database with the same effect. Our second goal is to achieve performance gain by taking advantage of the DBC hardware content-addressability, parallel read-out and clustering capabilities. A CODASYL database is usually accompanied by a collection of indexes. An index is maintained for each search key declared in the schema. We, however, intend to represent the entire database with very few indexes. This will be possible due to the content-addressability of DBC. Secondly, any record in DBC must contain as keywords all information on which a user may choose to conduct a search. Thirdly, we would like to locate a record without navigating through a sequence of other records and this implies the elimination of pointers within records. Under the above guidelines, let us take up the problem of representing a CODASYL database.

4.1. Representation of a Record

A CODASYL record occurrence is transformed into a DBC record by using attribute-value pairs, i.e., keywords. Since a record type is structured as a hierarchical configuration of data items, a keyword may be created for each elementary data item. Given a record occurrence with specific values for the individual data items, a keyword of the following format is created for each elementary data item and is made a part of the corresponding DBC record:

<ITEM.data-item-name, data-item-value>

where ITEM is a literal. If the names of the elementary data items are not unique, then they are qualified by the names of data items at higher levels. For example, consider a record type with the following structural definition:

```
RECORD NAME IS R
    ...
    02 A
       03 B
          04 C PIC 9(2)
          04 D PIC 9(2)
       03 E PIC X(5)
    02 C PIC X(5)
```

For each record occurrence of this type, the corresponding DBC record includes the keywords:

<ITEM.B.C, value-of-B.C>
<ITEM.D, value-of-D>
<ITEM.E, value-of-E>
<ITEM.R.C, value-or-R.C>.

Storing data items as keywords of attribute-value pairs increases the storage requirement since the data item names of the CODASYL record occurrence are now stored as attributes in every DBC record. However, in the DBC implementation, the data item names will be coded. Furthermore, no pointers will be embedded in the DBC records due to DBC's content-addressability.

There is a tremendous advantage in storing data item values as keywords of attribute-value pairs. Since they are not type-D keywords, there is no storage overhead for directory maintenance. Yet a random search can be efficiently conducted by DBC for records containing arbitrarily specified data item values. Conversely, to conduct a random search on arbitrary data items, a conventional CODASYL system will require an index on every data item. In the absence of such an index an exhaustive search of the database will be necessary.

The fact that a record occurrence belongs to a particular record type is indicated by means of the keyword

<TYPE, record-type>.

Thus DBC records can be content-searched on equality predicates based on record type. Similarly, the fact that a record occurrence is assigned an area is represented in the DBC record by the keyword

<AREA, area-name>.

This allows the notion of logical area to be supported, even though the records may not be physically grouped into areas.

The database key for a record occurrence may be generated by the system or it may be determined by the run-unit. The manner in which a database key is to be generated for occurrences of a specific record type is governed by the location mode of the record type. Once the database key has been determined, a keyword of the following form will be included in the DBC record:

<DBKEY, database-key>.

4.2. The Notion and Assignment of L-numbers

In CODASYL implementation, a database key generated on the basis of location mode identifies a physical address. This leads to a degree of data dependence which we intend to overcome in the DBC implementation. Instead of allowing a database key to represent a physical address, DBC can maintain database keys in the structure memory. Every database key <DBKEY, database-key> could be declared as a type-D keyword. Thus, given a database key, DBC can determine the cylinder number of its corresponding record. However, an abnormally large amount of storage would be required, since there will be a directory entry for every record in the database.

We, therefore, introduce at this point the concept of L-numbers. An L-number is a logical number assigned to every record occurrence in the database. It will not be used to identify a record but rather to aid DBC in locating the cylinder housing the record. We shall later note that both record type and L-number will be used in locating records in DBC. An L-number will actually act as a record type partition number. All record occurrences of a particular record type will be placed in disjoint partitions. Every such partition can be identified by a record type and an L-number.

The assignment of L-numbers is as follows. Let there be n cylinders and r record types in the database. Thus all occurrences of each record type can be accommodated in m (=n/r) cylinders on the average. We therefore need to assign m L-numbers uniformly among all the occurrences of each record type. We can thereby hope that all occurrences of a given record type and having a given L-number will fit into a single cylinder. However, because of the variability of the number of occurrences per record type, we shall use mp L-numbers (1,2,...mp) where $p > 1$. Possibly P=4 or 5, which is a design decision.

If the location mode is direct, then the database key of a record occurrence is hashed in order to determine its L-number from the range 1 through mp. If the location mode is calc, then the calc keys are hashed to an L-number. Finally, if the location mode of a record is via a set, then the appropriate set occurrence is first determined by using the set occurrence selection procedure for the given set type. Once the set occurrence is determined, the L-number of the owner of this set occurrence is known. We then assign this L-number to the record occurrence in consideration.

Once the L-number of a record occurrence to be stored has been assigned, the following keyword is included in that record occurrence:

<L-NUMBER, L-number>

where L-number is the one determined for the record occurrence. This keyword will be used both as a type-D keyword and a clustering keyword. Thus all records of a particular type with identical L-numbers will likely be stored in the same cylinder. Furthermore, since the possible number of L-numbers is small (only mp of them), the size of the directory and, therefore, the structure memory storage requirement, will also be small.

If, during the execution of a run-unit, a record occurrence is to be retrieved based on its location mode (direct or calc), then the L-number will first be calculated using the same hashing procedure that was used for storing the record occurrence. A DBC retrieval command will then be sent. The command will include the predicate (L-NUMBER = L-number) as part of the query. When location mode is via a set, the L-number only serves the purpose of clustering the records in a set occurrence; the location mode of such a record

is not used for retrieval purposes.

4.3. Representation of Set Types

A set type, as we have observed, consists of one owner record type and one or more member record types. A set occurrence will be identified by an occurrence of its owner record and it may consist of an arbitrary number of member record occurrences. It is important to remember that all occurrences of a given set are pairwise disjoint implying that no tenant (owner or member) may exist in two occurrences of the same set type. We shall now illustrate how the member records of a set type are represented in DBC and how their logical positions are indicated.

(a) Set Membership

Since a member record occurrence belonging to a set occurrence is also identified by an owner record occurrence, we assume that the record occurrence r is a member of the set type called set-type and that the corresponding set occurrence is identified by owner-database-key, which is the database key of the owner record occurrence. We then include in r the keyword

<SET.set-type, owner-database-key>

in order to identify the set occurrence in which it is a member. For each set in which r is a member, a keyword of this form will be included in r.

Although the database key of the owner record occurrence uniquely identifies a set occurrence, it is not enough to store only that in a member record occurrence. This is due to performance reasons. Given a member record occurrence, we shall often be required to locate its owner in a given set, and this cannot always be done from a knowledge of the database key alone. We also need to know the L-number (record type partition number) of the owner record occurrence. Assume, once again, that a record occurrence r is a member of the set type called set-type and that the owner of the corresponding set occurrence has an L-number termed owner-L-number. We then include in r the following keyword:

<OWNER-L-NUMBER.set-type, owner-L-number>.

A record type may also be declared to be the owner record type of an arbitrary number of set types. For each set type of which a record occurrence r is an owner, we include in r the keyword

<SET.set-type, OWNER>

where set-type is the name of the set type.

(b) Set Ordering

In the CODASYL system, the logical position of a member record within a set occurrence may also be considered to be of significance. The member records may be ordered in two different ways. The ordering may be based on the order of insertion of a new entry into a set type and also on the current record of the set type. In the second method, the ordering is based on certain sort keys of the member records.

In case the set ordering of a given set type is to be determined by the order of insertion of records (such as when order is declared to be NEXT, PRIOR, FIRST or LAST), then a sequence number may be assigned to the member records of each set occurrence. The sequence number s_i of the i-th member is such that $s_i < s_{i+1}$ for $1 \leq i \leq n-1$, where n is the total number of members in the set occurrence. The assignment of sequence members is fairly straightforward, as shown in [9]. It may only be required to examine the sequence numbers of the two records adjacent to the record to be inserted for determining its sequence number. Once the sequence number of a record within an occurrence of a given set type is decided, the number is stored as a keyword of the record in the form:

<SET-POSITION.set-type, sequence-number>.

In case a set occurrence is to be sorted by some data items, then no special keyword need be included to represent the order. By using the hardware sorting module of DBC, records of the set occurrence may be sorted on their way to the front-end computer.

4.4. Type-D Keywords and Clustering

The choice of type-D keywords is always based on the type of keywords that appear in a query. Every query conjunction should have at least one predicate that consists of a type-D keyword. If this condition is not satisfied, then the query can be answered only by exhaustively searching every disk cylinder of the mass memory. With the requirements of data manipulation in mind, we have decided on type-D keywords as all those that have one of the following attributes:

(1) TYPE
(2) AREA
(3) L-NUMBER.

Some type-D keywords will also be made clustering keywords. We will primarily be interested in clustering set occurrences, since set traversal is the most important operation in a network database.

(a) Clustering Method I

We may cluster by L-numbers, because all members of a set occurrence have the same L-number if the location mode of the member records have been declared to be via that set. Thus an entire set occurrence will be accommodated in as few cylinders as possible.

(b) Clustering Method II

A second clustering method is to cluster primarily by record type and secondarily by L-number. Thus all occurrences of the same record type will be placed in as few cylinders as possible. For example, if a record type R has 10,000 occurrences and each cylinder can accommodate 2,000 of them, then it is conceivable that only 5 cylinders will

contain all the occurrances of R. Clustering secondly by L-number will normally ensure that all occurrences of R that have the same L-number will be placed in the same cylinder.

(c) Choice of a Clustering Method

The first method is useful when many record types are located via a single set type S. In that case, an occurrence of S is likely to be placed in a single cylinder because all the members of that set occurrence have the same L-number. The second method will place an occurrence of S in possibly n cylinders if there are n different member record types.

On the other hand, if a record type R has a location mode direct or calc and is declared to be a member of a set type S, then the second method is far better. In the first method, a set occurrence of S is likely to be scattered over many cylinders, a number not much smaller than the number of records in that set occurrence. In the second method, a set occurrence of S will spread over approximately m cylinders, where m is the number of cylinders required to contain all the occurrences of record type R, and this number is likely to be much smaller than the number of member records in a set occurrence.

In conclusion, we shall use the second clustering method. We note that conventional implementations of CODASYL databases do not cluster by record type, because the database keys are user-specified when the location mode is direct. Thus, traversing a set type S, whose members do not have a location mode via S, will require many more accesses in a conventional CODASYL database system than in DBC.

4.5. Directory Storage Requirement

Directory entries are stored in the DBC structure memory for every type-D keyword. Our choice of type-D keywords is directed towards efficient processing of data manipulation operations and also towards minimizing the directory memory requirement. What follows now is a gross analysis of the directory storage requirement for a CODASYL database. As we shall see, this requirement is indeed small.

Let the database consist of r record types, a areas and n cylinders. Let m = n/r. We shall then use mp L-numbers where p is a small number greater than 1. Let us further assume, for simplicity, that each type-D keyword requires four bytes of storage and each cylinder number can be represented in two bytes.

Each directory entry will consist of a type-D keyword and one or more cylinder numbers. Since we cluster records by their types, all records of a given type will be accommodated in n/r cylinders on the average. Since we cluster secondarily by L-numbers, all records with a given L-number will be spread over r cylinders (because all record occurrences of the same type and same L-number will possibly be clustered within a single cylinder). If record types do not, in general, span more than a single area, then we may expect r/a record types to be assigned to each area. Therefore, each area will be spread over (n/r).(r/a) = n/a cylinders. Thus, under the above assumption, records are automatically clustered by area as well.

We tabulate below the memory requirement for storing the directories in the structure memory.

Type of keyword	No. of such keywords in the directory	No. of cylinder references	Total directory memory requirement (in bytes)
TYPE	r	n/r	r(4 + 2n/r)
AREA	a	n/a	a(4 + 2n/a)
L-NUMBER	np/r	r	(np/r)(4 + 2r)

Thus, the total directory memory requirement is

$$(4r + 2n) + (4a + 2n) + (4np/r + 2np)$$

$$= 2(2a + 2r + 2n + 2np/r + np) \text{ bytes.}$$

As an example, if the database consists of 10,000 cylinders, 10 areas and 100 record types and if p is chosen to be 5, then the directory memory requirement is approximately 143,000 bytes. This is an extremely small fraction of the database size. In fact, if the cylinder size is 10^6 bytes, then the directory memory size is less than 0.01 percent of the size of the database.

5. Query Translation

Even though CODASYL operations work on a single record at a time, it is still possible to take advantage of DBC's content-addressability and parallel read-out capability by simultaneously accessing all records having the same property (such as, all records belonging to the same set occurrence). Since user transactions tend to request groups of related records (notwithstanding the use of single-record commands), the DBC capabilities account for a large saving in the number of database accesses.

User programs issue CODASYL data manipulation language (DML) statements, which are routed to the database interface (DBI). The DBI, in turn, translates the DML statements into equivalent DBC commands for execution by DBC.

5.1. Organization of the Database Interface (DBI)

The overall organization of the DBI is depicted in Figure 2. The DML statements issued by a user program are passed on to the DBI which consists of a DML translator, a system buffer manager, a system buffer (ISB) and several auxiliary data structures. The DML translator (DMLT) translates DML statements into DBC commands and monitors the execution of these commands by DBC. The system buffer manager (SBM) does the buffer storage management in the front-end computer. Auxiliary data structures, the set information table (SIT) and the area information tables (AIT) serve to improve the system performance. The motivation for and the organization of SIT will be discussed later. The current-pointer (C-P) is the buffer

address of the current record occurrence of the run-unit.

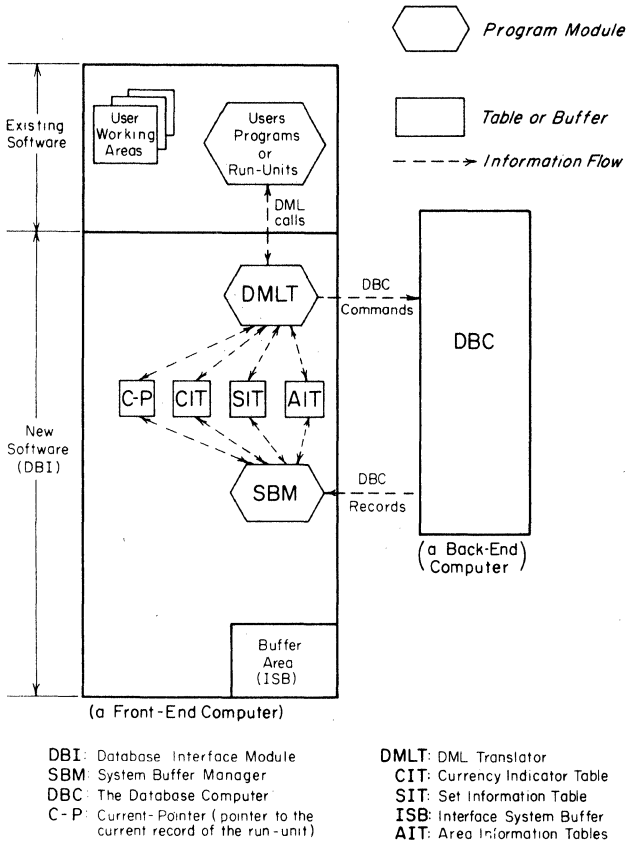


Figure 2. The Database Interface, DBI

One record occurrence of each record type can be made available to the user in an area called the user working area (UWA). The UWA has just enough space to accommodate an occurrence of each record type. The portion of the UWA reserved for a given record type is commonly referred to as the UWA for that record type. The user can directly manipulate any data in his UWA. To fetch a record from the database, the user issues a get call. It is intercepted and processed by the DBI; an appropriate record occurrence is then placed in the UWA by the DBI.

The current records (of run-unit, set types, etc.) are established by the run-unit using DML find statements. The DBI stores currency information in the currency indicator table (CIT). The information maintained in the CIT consists of:

(1) For each area:
   a) area name
   b) record type of the current record of the area
   c) L-number of the current record of the area
   d) database key of the current record of the area.

(2) For the run-unit:
   a) record type of the current record of the run-unit
   b) L-number of the current record of the run-unit
   c) database key of the current record of the run-unit.

(3) For each record type:
   a) record type
   b) L-number of its current occurrence
   c) database key of its current occurrence.

(4) For each set type:
   a) set type
   b) information about the current record of the set type
      i) whether current record is a member or the owner
      ii) record type of current record
      iii) L-number of current record
      iv) database key of current record
      v) position information about current record
      vi) owner record type
      vii) L-number of the owner
      viii) database key of the owner

5.2. The Set Information Table

A great majority of the time, the user application program of a conventional CODASYL system may be traversing one or more set occurrences. Typically, a run-unit may traverse through set types in a hierarchical order as illustrated in Figure 3, where each of three different set types have only one member record type. More specifically, the set type SETX with owner type A and member type B has currently three set occurrences labeled X1, X2, and X3 in the database. SETY has only two occurrences and SETZ has three. The user application program may perform the following traversal routine. The set occurrence X2 is obtained directly by locating the owner record occurrence a2. X2 is now completely traversed by accessing every member bi in X2 and by traversing every occurrence Yj of SETY in which bi is an owner. While traversing an occurrence of SETY, in turn, every member ck in that occurrence is accessed and every occurrence Zm of SETZ in which ck is an owner is also traversed. Thus the traversal order is the following sequence of set occurrences: X2, Y1, Z1, Z2, Y2, Z3. In implementing this traversal sequence we will have in the buffer complete set occurrences, at most one of each set type. The buffer will have set occurrences as shown below at different stages of processing:

```
Stage 1.  X2           : process b6, b7
Stage 2.  X2, Y1       : process c1
Stage 3.  X2, Y1, Z1   : process d1, d2, d3, d4
Stage 4.  X2, Y1       : process c2, c3, c4
Stage 5.  X2, Y1, Z2   : process d5, d6
Stage 6.  X2, Y1       : process c5
Stage 7.  X2           : process b8
Stage 8.  X2, Y2       : process c6, c7
Stage 9.  X2, Y2, Z3   : process d7, d8, d9
Stage 10. X2, Y2       : process c8, c9, c10
Stage 11. X2           : process b9
```

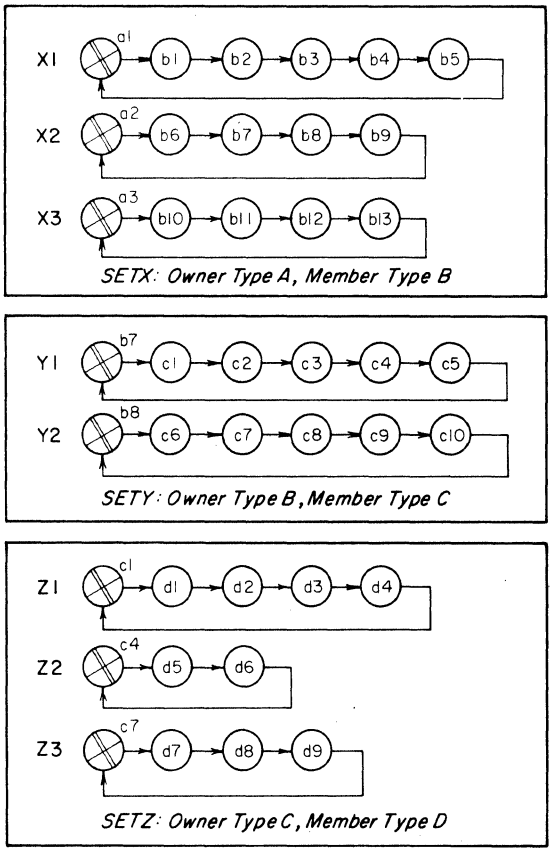


Figure 3. Example of Set Traversal

To traverse a set occurrence, the application program must name it the current set occurrence. Thus at stage 5 of the above buffer configuration, X2, Y1 and Z2 are the current set occurrences of SETX, SETY and SETZ, respectively.

The set information table (SIT) is used in order to keep track of the buffer (ISB) information related to set occurrences. Accesses to the database can thereby be saved if any necessary record is already existent in the buffer storage. The SIT will have an entry for every set type in the database, to indicate whether its current set occurrence resides in the buffer or not. An entry of the SIT consists of the following:

(1) Set type
(2) Pointer to owner of current set occurrence -- null if current set occurrence is not in the buffer
(3) Pointer to current record of the set -- null if current set occurrence is not in the buffer
(4) Record type of member record -- valid only if all occurrences of a single member record type (and no other) of the current set occurrence is in the buffer
(5) Number of member records in the set occurrence as stored in the buffer.

The pointers are buffer addresses, where entire set occurrences are stored in consecutive locations. The member records are ordered and the logical adjacency is reflected in physical adjacency. In case physical adjacency cannot be maintained, pointers (links) are employed. If the current occurrence of a set type is in the buffer, then its owner record contains information on its database key and L-number. The fourth item of an entry in the SET is useful, when a user is interested in traversing the occurrences of only a particular member record type in a set occurrence; but it is not concerned with the other member record types. In this case, the DBI will retrieve only that part of the set occurrence which consists of member records of the specified type, thus preventing a wastage of buffer space and the time required to order unnecessary records.

5.3. Retrieving Entire Set Occurrences

If an entire set occurrence is needed, a general procedure to retrieve the set occurrence is given below. The identification of the set occurrence (owner-type, owner-db-key, owner-L-number) is provided to the routine. The given set type is S.

(1) Let R_1, R_2, ..., R_n be the member record types of S whose location modes are via set type S. Any occurrence of these record types will have an L-number which is the same as that of its owner. Let R_{n+1}, ..., R_m be the other member record types of set type S.
(2) The required set occurrence is now retrieved as follows.
   (i) For each record type R_i, $1 \le i \le n$, issue the following command to DBC "retrieve all records that satisfy <TYPE=R_i> ∧ <L-NUMBER=owner-L-number> ∧ <SET.S=owner=db-key>". (This usually requires one database access for each record type, due to clustering by record type and L-number.)
   (ii) For each record type R_i, $n+1 \le i \le m$, issue the command "retrieve all records that satisfy <TYPE=R_i> ∧ <SET.S=owner-db-key>". (This may require more than one access, since secondary clustering information can not be used.)

In the above procedure, the entire set occurrence is retrieved, except the owner record occurrence. Normally, the owner record occurrence will already be present in the buffer. If it is necessary to retrieve the owner record occurrence as well, then issue an extra command to DBC to "retrieve the record satisfying

<TYPE=owner-type> ∧ <L-NUMBER=owner-L-number>

∧ <DBKEY=owner-db-key>".

5.4. Traversing Set Types

Conventionally, a set type is traversed by executing a sequence of DML find statements of the form

FIND NEXT (or PRIOR, or LAST, etc.) RECORD OF SET set-name.

Statements of this type retrieve the database key of the next or prior record with respect to the current record of the specified set, or they retrieve the n-th record (which may be first or last or any position) of the current set occurrence of the specified set. Another version of this find statement also indicates a member record type (e.g., find the tenth record of type R in the current set occurrence of set type S).

Because of the set information table (SIT) and because of the fact that we may retrieve entire set occurrences into the buffer, set traversal by using a succession of find statements of the above type will mostly involve sending a record from the buffer to the user application program in the new database interface (DBI) environment. Only when the required set occurrence is not already in the buffer, must a command be sent to DBC to retrieve that set occurrence.

To execute a find statement for traversal of set type S, the following procedure may be invoked:

(1) From the currency indicator table (CIT) determine the owner record occurrence of the current record occurrence of S. Let it be identified by

(owner-type, owner-L-number, owner-db-key).

(2) Use the pointer in the SIT to determine if the current set occurrence is in the buffer.
(3) If the current set occurrence is in the buffer, then go to step 5; otherwise, fetch that set occurrence.
(4) Update the SIT to indicate the fact that the current set occurrence is now in the buffer.
(5) Find the new current record of the set. It is already in the buffer.
(6) Update the currency indicator table (CIT).

5.5. Retrieving a Record or a Group of Records

The user application program may have one or more DML find statements in order to select a record or a group of records with some given property. The simplest type of find statement is the one in which the user specifies a record type R and a database key D. In this case the location mode of the record is direct. Retrieval of such a record can be done by using the query <TYPE=R> ∧ <DBKEY=D>. However, records are clustered also by their L-numbers. Hence the above query may require a search of more cylinders than actually are required, although record type is an attribute of a type-D keyword. We, therefore, compute the L-number L of the record occurrence from its database key by hashing, as described. The revised query, then, is <TYPE=R> ∧ <DBKEY=D> ∧ <L-NUMBER=L>.

Another find statement requires locating the owner record occurrence of type R in set type S, the occurrence of which is determined by the current record of a set X (or of a record type R_1 or of an area A or of the run-unit). The following steps may be performed to find the required owner record S.

(1) From the CIT entry for set X (or record type R_1, or area A or run-unit), extract the L-number L, the database key D and record type R_2 of the current record.
(2) Issue a command to the DBC to "retrieve the record satisfying the query

<TYPE=R_2> ∧ <L-NUMBER=L> ∧ <DBKEY=D>".

This retrieves the current record occurrence r of set X.
(3) From record r, extract the keyword with attribute SET.S, whose value is, say, owner-db-key. Also extract the keyword with attribute OWNER-L-NUMBER.S, whose value is, say, owner-L-number.
(4) The required owner record occurrence of S is now retrieved by using the query

<TYPE=R> ∧ <L-NUMBER=owner-L-number> ∧ <DBKEY=owner-db-key>.

Another type of find statement locates an occurrence of record type R, whose location mode is calc. The data items d1, ..., dn specified in the location mode clause are initialized by the run-unit and stored in the user working area (UWA). The user may retrieve all occurrences of type R that have the same values for the data items d1, d2, ..., dn by executing a sequence of find statements of this type, but qualified by the next-duplicate-within clause; for example,

FIND NEXT DUPLICATE WITHIN RECORD TYPE R.

Such a sequence of find statements can be very easily executed, and, in fact, in only a single access to the database. By hashing the data items d1, ..., dn, determine an L-number L. Now issue a retrieval command to DBC based on the query

<TYPE=R> ∧ <L-NUMBER=L>
∧ <RECORD.d1=value-of-d1> ∧ . . .
∧ <RECORD.dn=value-of-dn>.

All the required records have the same values for L-number and the specified data items. Thus, they are all retrieved simultaneously. Only a single access will be required because of clustering based on record type and L-numbers.

Finally, we shall consider a find statement in which a set name S and the values of certain data items d1, ..., dn are specified for a record type R. An occurrence of S is selected based on either the current record of the set or the set occurrence selection criterion for S. A member record occurrence of type R is now to be located that has given values for data items d1, ..., dn. To execute such a statement, an occurrence s of set S is first selected. Let this occurrence be identified by the owner record with L-number L and database key D. The required member record occurrence in this set occurrence is found by using one of the two following queries depending on whether

the location mode of R is via the set S:

(i) If the location mode of R is via S, then the L-number of the required member record occurrence is the same as that of its owner. Therefore, use the query

<TYPE=R> ∧ <L-NUMBER=L> ∧ <SET.S=D>
∧ <RECORD.dl=value-of-dl> ∧ . . .
∧ <RECORD.dn=value-of-dn>.

(ii) Otherwise, the L-number of the member record occurrence is not known, so, use the query

<TYPE=R> ∧ <SET.S=D>
∧ <RECORD.dl=value-of-dl> ∧ . . .
∧ <RECORD.dn=value-of-dn>.

This query will, in fact, retrieve all records of the set occurrence s that contain the specified data items. Thus any subsequent find statement requesting duplicates can be executed without further references to the database.

5.6. Relative Performance

The DBC implementation of a CODASYL database contributes to a large gain in performance over a conventional implementation. Because of the limitation of space, we shall provide only a first-order estimation. For more thorough analysis of the performance issue, the reader may refer to the methodology presented in [10], [11]. In terms of the number of database accesses required to execute a user transaction, the DBC performance gains are due to the following:

(1) A disk cylinder, which is at least 40 times as big as a page of a conventional system, can be content-searched by DBC in a single revolution.
(2) The clustering policy of DBC allows all occurrences of a record type to be placed together. In other words, the member records of a set occurrence are no longer scattered over many physical blocks; instead, they actually lie close to one another.

In locating a single record based on its database key (if location mode is direct) and based on its calc keys (if location mode is calc), DBC performs at least as well as a conventional system because of the policy of determining L-numbers based on location mode. Since the record type and L-number of the target record will be known, DBC will usually require only one access to the database.

More frequent, however, is the operation of searching a record based on its participation in set types. To find an arbitrary member record satisfying a given predicate, a conventional system will have to go through n/2 other member records, on the average, if the corresponding set occurrence has n member records. This calls for approximately n/2 page accesses if the set occurrence is not clustered (i.e., the location mode of the member records is not via that set). It will require about n/(2p) page accesses if the set occurrence is clustered and a page can accommodate p records.

In DBC, however, the same search for a record among n member records of a set occurrence will require approximately m/c cylinder accesses if the set occurrence is not clustered, and about n/c cylinder accesses if the set occurrence is clustered. Here, m is the total number of occurrences of the member record type and c is the cylinder size in terms of records. In the second case, the members of the given set occurrence have the same L-number. Since these records are clustered by record type and L-number which are both known, only n/c accesses will be necessary.

Since the page size p is at least 40 times smaller than the cylinder size, DBC will perform about 40 times better when n is large and the set occurrence is clustered. If the set occurrence is not clustered, then DBC will perform 10 to 100 times better for reasonable values of n, m and c.

A very frequent operation on a CODASYL database is the traversal of set types. The user normally navigates through the database via set types. Quite frequently, the system may also have to go through set types, for example, to carry out an update. It is easy to observe that DBC performs as well in traversing a set type as it does in locating a member record within a set type.

We have not mentioned in this paper how DBC performs updates. The algorithms for database update will be found in [9]. All updates are preceded by a database search, which is conducted very efficiently by DBC. In the actual update operation, DBC performs many times better than a conventional system if the update calls for automatic set traversal.

6. Concluding Remarks

We have presented in this paper a methodology for the implementation of a CODASYL database on a database computer, DBC. Since DBC can content-search one cylinder at a time and the CODASYL database is clustered, most access requests can be satisfied in as few cylinder accesses as possible. The parallel read-out capability of DBC is used for accessing entire set occurrences or groups of related records. Since the information content of these records is partially known to DBC, they can be effectively used in carrying out future DML requests.

A first-order estimation indicates that the performance of DBC is one or two orders of magnitude higher than conventional systems. This is due to the effective use of DBC hardware and choice of proper data clustering strategy. Thus, DBC can be used in supporting a CODASYL database and replacing the conventional CODASYL data management system with improved performance.

References

1. Baum, R.I., and Hsiao, D.K., "Database Computers - A Step Toward Data Utilities," IEEE Transactions on Computer, Vol. C-25, No. 12, Dec. 1976, pp. 1254-1259.

2. Hsiao, D.K., and Madnick, S.E., "Data Base Machine Architecture in the Context of Information Technology Evolution," Proceedings of the Third International Conference on Very Large Data Bases, Japan, Oct. 1977. (Available from either ACM or IEEE Computer Society.)

3. Banerjee, J., Baum, R.I., and Hsiao, D.K., "Concepts and Capabilities of a Database Computer," ACM Transactions on Database Systems, (to appear) Vol. 4, No. 1, March 1979.

4. Hsiao, D.K., and Kannan, K., "The Architecture of a Database Computer - A Summary," Proc. of the 3rd Workshop on Computer Architecture for Non-Numeric Computation, Syracuse, May 16-17, 1977. (Available from ACM.)

5. Kannan, K., Hsiao, D.K., and Kerr, D.S., "A Microprogrammed Keyword Transformation Unit for a Database Computer," Proceedings of the Tenth Annual Workshop on Microprogramming, Oct. 1977, Niagara Falls, New York, pp. 71-79.

6. Hsiao, D.K., Kannan, K., and Kerr, D.S., "Structure Memory Designs for a Database Computer," Proceedings of ACM '77 Conference, Oct. 1977, Seattle, Washington, pp. 343-350.

7. Kannan, K., "The Design of a Mass Memory for a Database Computer," Proceedings of the Fifth Annual Symposium on Computer Architecture, April 1978, Palo Alto, California, pp. 44-51. (Available from IEEE Computer Society.)

8. CODASYL Data Base Task Group Report, April 1971, ACM, New York.

9. Banerjee, J., Hsiao, D.K., and Kerr, D.S., "DBC Software Requirements for Supporting Network Databases," Technical Report OSU-CISRC-TR-77-4, The Ohio State University, Columbus, Ohio, June 1977.

10. Banerjee, J., and Hsiao, D.K., "The Use of a Database Machine for Supporting Relational Databases," Proceedings of the Fourth Workshop on Computer Architecture for Non-Numeric Processing, Syracuse, New York, August 1-3, 1978.

11. Banerjee, J., and Hsiao, D.K., "Performance Evaluation of a Database Computer in Supporting Relational Databases," Proceedings of the Fourth International Conference on Very Large Data Bases, Berlin, Germany, Sept. 13-15, 1978.

PART III

AN IMPLEMENTATION OF DBC

EVOLUTION OF A DATA BASE COMPUTER

O.H. Bray, H.A. Freeman, and J.R. Jordan
Sperry Univac
St. Paul, Minnesota 55165

1. INTRODUCTION

Data Base Management (Software) Systems (DBMS's) have as their basic objective the improvement of an organization's control and utilization of its data resources. This goal is met by improving the availability, integrity, and security of the data base. With current computer systems, however, trade-offs must be made among the various DBMS objectives. Basically, the trade-off decision is between performance and functionality. Regarding performance, both response time and throughput are critical and must meet the users' requirements. Functionality covers availability, integrity, and security. Unfortunately, increased functionality is obtained only at the expense of performance. With data independence, for example, the DBMS translates the data automatically from its stored form to the form that the user expects. The result is that the system is easier to use, allowing the data to be restructured and modified without having to change the application programs. This automatic data translation, however, requires additional processing time and storage, thus directly affecting both response time and throughput. From another perspective, however, this performance versus functionality trade-off is simply the question of machine efficiency versus people efficiency. Considering the dramatic decreases in hardware costs as opposed to the rapid increases in personnel costs, the direction of this trade-off is apparent.

One means of using hardware to improve the performance of current Data Base Management (Software) Systems while offering increased functionality to reduce labor costs is with a data base computer (hardware) system. Using such new technologies as LSI, VLSI, microprocessors, magnetic bubble memories, and charge-coupled devices, data base computers are well-suited to the information storage and processing needs of the 1980's. In addition to offering improved performance, these special purpose computers free the general purpose host's resources for other tasks, provide a hardware assist to the security problem, and offer a more effective way of sharing data in a network.

Sperry Univac is investigating various ways of improving Data Base Management System performance and has several research efforts in this area. This paper describes the results of one of these research efforts.

2. PREVIOUS APPROACHES

Two different approaches to a data base computer system have been proposed and implemented to date. The first is to off-load the DBMS from the host onto a general-purpose minicomputer. Although construction of this type of system has proven feasible, the performance benefits are not realized.[1] The main reason for this lack of performance improvement is due to the use of conventional sequential processors to perform data management functions without any hardware assists.

The second approach is to use specialized devices to perform part or all of the data management tasks. ICL has the only announced product to date in this area with their Content Addressable File Store (CAFS).[2] ICL combined a minicomputer with special selection logic to offer high speed selection of records based on content. CAFS is an outgrowth of the three basic specialized processor approaches, CASSM[3], RAP[4], and the DBC[5].

Of the three basic data base computer architectures, DBC appeared the most attractive from a commercial computer company's point of view. CASSM is based on a disk technology that apparently will be obsolete in the 1980's. Also, in requiring read and write heads for every track, CASSM would be very expensive in supporting large data bases. RAP also appears viable for only very small data bases. Also RAP may require an extensive amount of staging which may be a severe bottleneck. In addition to attempting to avoid these problems, DBC is the only data base computer that claims to support CODASYL as well as Relational DBMS's. Since all of Sperry Univac's current data base management users have a CODASYL implementation[6], this is an extremely attractive feature.

3. CURRENT APPROACH

In order to confirm the benefits of the DBC approach and to determine what the functional requirements of a data base computer should actually be, an application investigation and analysis effort was first undertaken. A sample of the designers and users of data base applications was polled and the responses to specific questions were recorded. The application areas covered ranged from the transaction only, fast response time, simple access method airline reservations applications to the heavily batch-oriented subscriptions processing applications. The results of the survey and the subsequent analysis proved to be quite useful in establishing a set of requirements for a data base computer.[7] It showed that DBC's proposal to cluster data in large blocks corresponding to a cylinder on a moving-head disk would be most efficient for most applications. It brought out the fact that most applications do not limit the number of people allowed to update. Finally, the survey showed that many of Sperry Univac's DBMS users would have data bases in the 10 to 50 billion byte range in the 1980's.

With the data base computer system requirements formulated, the DBC design was then revised and extended. The basic approach of accelerating both retrieval of the directory information and the data remained the same. Parallel transfer of large blocks of data which are then processed in parallel on a content-addressable basis is the key to this design. To this capability, interprocessor communication was added to allow for the "full" relational join and for complete sorts. Also, a solid state associative processing element was invented in order to identify unique search key and sort field values. Described in the following section, the resulting design appears to offer significant performance improvement at a relatively modest cost.

4. DATA BASE COMPUTER DESIGN

4.1. ARCHITECTURE

The architecture of the extended Data Base Computer is shown in Figure 1. In the original DBC design, two loops were used for processing commands or queries. The data loop elements accessed and stored the data base and post-processed retrieved records. The structure loop elements determined the authorized records for accesses and clustered records to be inserted in the data base. Since the functionality of both the data and structure loops utilizes the same parallel processing techniques, this design incorporates both of these functions into a single structure.

The Data Base Computer design has six major components:

1. Data Base Computer Controller (DBCC),
2. Processing Elements (PE's),
3. Memory Modules (M's),
4. Key Processor (KP),
5. Parallel Transfer Disk Controller (PTDC), and
6. Parallel Transfer Disks (PTD's).

These components and their interconnection are described in the following paragraphs.

Implemented in the form of a minicomputer, the Data Base Computer Controller accepts commands from the host. They are either data manipulation-level commands from a CODASYL based DBMS or high-level query type commands of a relational nature such as Sequel[8] or QLP[9] commands. The commands are then processed by the DBCC and an appropriate set of parameters and commands is generated for PE's, KP, and/or PTDC operations.

The data base and directory information resides on a series of Parallel Transfer Disks. The Parallel Transfer Disk Controller handles the transfer of information between the drives and the buffers (M's). The PTDC provides the usual error correction, defect processing, and other features commonly found on disk controllers today. Both the Parallel Disks and the Controller do not involve any technological breakthroughs to achieve parallel transfers. Ampex Corp. has modified one of their 9300 series 300 megabyte disks to offer the transfer of up to 9 disk tracks in parallel.[10]

Information from the appropriate Parallel Transfer Disk tracks is transferred simultaneously (i.e., in one disk rotation time) into the associated buffers (i.e., Memory Modules). A given Memory Module has two banks to allow overlap of input and output operations. Each one is connected so as to serve two Processing Elements. The Memory Modules are sized to contain four disk track's worth of data and at least one half of a disk track's worth of directory or "structured memory"-type of information. For the Ampex PTD with 20 KB per track, the M's would each contain 96 KB of random access memory.

After the Memory Modules have been loaded with a track's worth of data, the Processing Elements (microprocessors) start operating on the data asynchronously performing the required functions. The odd numbered PE's and the even numbered PE's are connected respectively together via a busing mechanism. This interconnection of Processing Elements and their connection to adjacent Memory Modules permits all of the operations described in the next subsection. Data

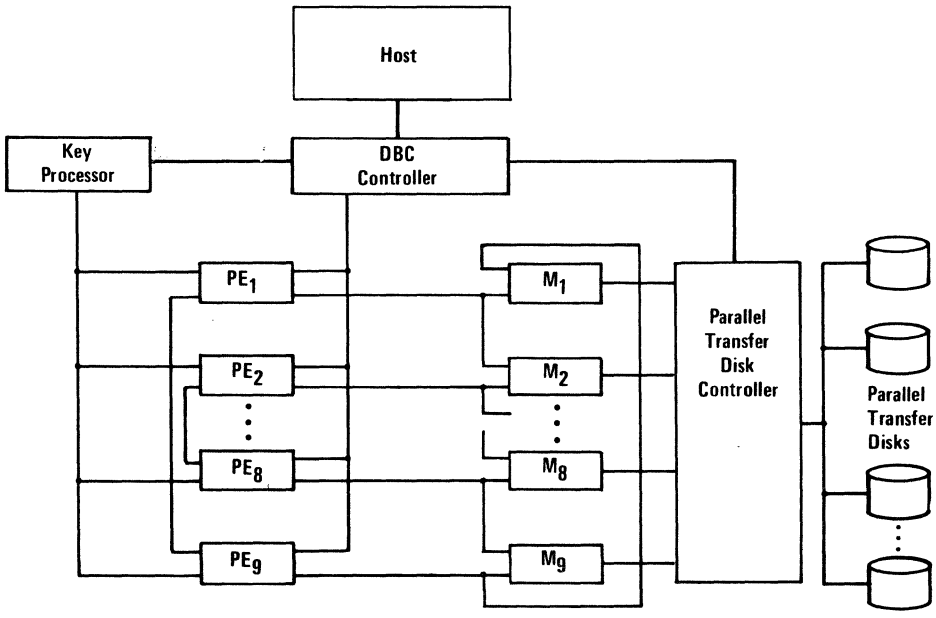


Figure 1. Data Base Computer Architecture

obtained from these operations is subsequently passed back to the host or sent out to the disks.

The Key Processor (KP) is an unique element that is used to accelerate certain data base operations by providing a temporary partitioning of a file over an attribute value space. As shown in Figure 2, the Key Processor consists of five major components:

1. Control logic,
2. Search Elements (SE's),
3. Memory Modules (M's),
4. Insertion logic, and
5. Interface logic.

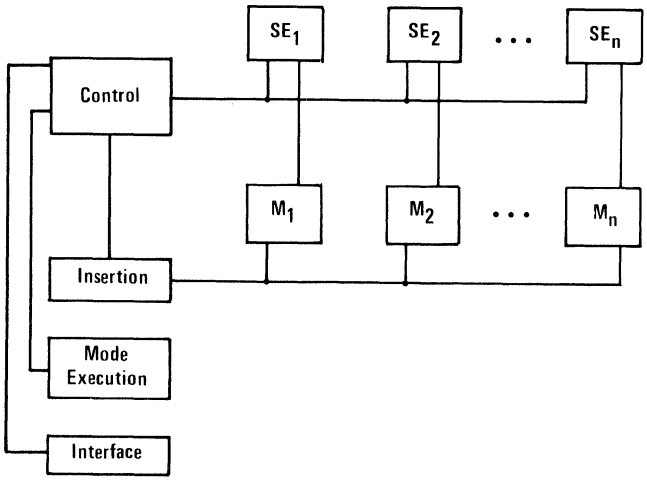


Figure 2. Key Processor

The Key Processor has been designed to use conventional RAM chips in segmenting a list of search arguments by a parallel binary search and insertion technique. When a request from a PE arrives for the KP, the argument, x, is sent to each SE. If the argument is already in an SE's Memory Module, a positive response is sent to the Control logic. If a negative response (no response) is received, the Control logic selects a Memory Module and initiates the insertion of x into M using the Insertion logic.

Four modes of operation are programmed into KP in order to perform the requested task for the Processing Elements. These include:

1. Presence: The KP responds positively to the requesting PE if the presented argument is already in the KP. If the argument is not there, it will then be inserted.
2. Entry Number: The KP assigns arrival numbers to each unique entry and when requested, returns the entry number to the requesting PE.
3. Count: A counter associated with an argument already in the KP is incremented by one.
4. Value: The counter or data value of a certain argument is returned to the requestor.

The use of these operations in functions performed by DBC is given in the next subsection.

4.2. DBC OPERATION

In general, all of the components in DBC operate synergistically to perform the functions described in this subsection.

SEARCH. A cylinder slice at a time is loaded from a particular PTD into the Memory Modules. Initiated by DBCC, all of the PE'S search their respective M's for the records that meet the search criteria. Using microprocessors for PE's and buffering the information in the Memory Modules allows for search arguments of arbitrary boolean complexity. The records that are found that qualify are either sent to the DBCC or moved to a different portion of the M for further processing. A diagram of the elements involved in the search operation is shown in Figure 3. Typically, a track's worth of information is loaded into A_i, qualifying records in need of further processing are moved to B_i, and records to be output to disks are moved to C_i.

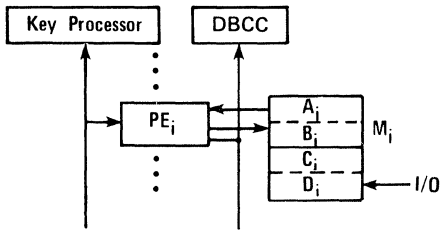


Figure 3. Search

PROJECTION. This operation involves selecting only a portion of the qualifying records. If Projection with the elimination of duplicates is desired, the Key Processor in Figure 3 is involved. In this case, the argument string of the qualifying record is presented to KP which is operating in the Presence mode. If the argument string is already in KP, the record is a duplicate and discarded. Otherwise, the new string is stored in KP and the record is prepared for output to the host or disk.

FULL JOIN. In the Full Join, a record from a relation (or file) A is concatenated with a record from a relation (or file) B if they have the same value of a common attribute. The Full Join is performed in several steps. First, for each record in A, the particular PE transmits the argument string to the Key Processor which is operating in the Entry Number mode. When a number, k, is returned by KP, the record is then added to list number k in the requesting PE's Memory Module. Second, after all of the records in A have been treated, the records in B are processed. In this step, each of the argument strings from the records in B are sent to KP which is now operating in the Value mode. If the particular argument string of the record is not in KP, the record is discarded. Otherwise, the list number, k, containing the appropriate records to be concatenated, is reported. The requesting PE then concatenates the record from B with all of the records from A

in its memory on list k to form a new set of records A' and then outputs them (See Figure 4). The Full Join is still not complete, however, at this point. The record from B also has to be concatenated with all of the records in its adjacent memory as well as to all of the other memories. The other memories are reached by broadcasting the record from B to all of its connected PE's which in turn concatenate the record with the appropriate records from A in their own and their adjacent memory. Thus the record from B is "joined" to all of the records that match in all of the Memory Modules. When all of the records from B have been processed in this manner, the Full Join operation has been completed.

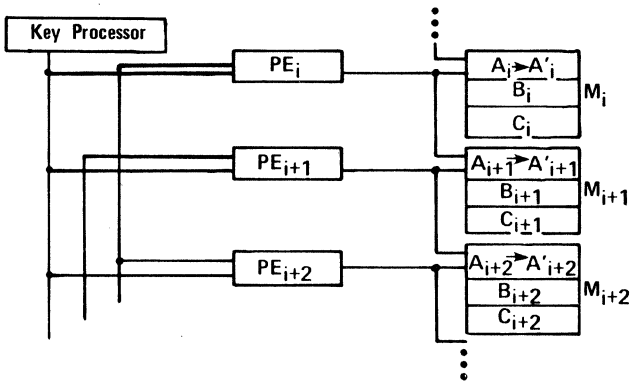


Figure 4. Join

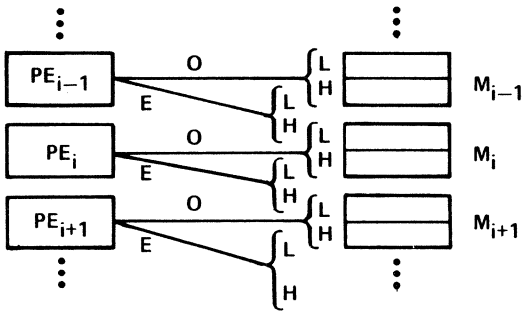
IMPLICIT JOIN. The Implicit or Half Join selects records from a relation (or file) A that have a common domain with records in relation (or file) B. No concatenation is required and, therefore, is a simpler operation than the Full Join. During the first phase of the Implicit Join, the unique argument strings of file A are stored in the Key Processor. In the second phase, argument strings or values from file B are checked by KP operating in the Presence mode. All matches with values in KP will result in the output of records from B. Records from B that do not have a corresponding value in the KP are simply discarded. It should be noted that communication between PE's is not required for this join function.

ADDITION. In order to add a record to a file or a tuple to a relation, the DBCC first selects a cylinder slice which is then loaded into the memories M's. Then on a space available basis, a particular PE is chosen to insert the record into its "track".

UPDATE. If the records from a particular file are fixed length, the individual records can be modified in place and then written back to the disk. For variable length records, the changes are made to the records in one segment of the Memory Modules as they are copied into another segment. When a track's worth of records has been collected in the new segment, they are then written back to the disk.

SORT. The sorting operation is shown in Figure 5. Each PE first does a standard sort/merge of the contents of its own Memory. This is followed by a merge of the contents of the lower portion of its memory with the upper portion of its adjacent memory. These two operations are alternated until the file is completely sorted.

DELETION. For this operation, records are read from the Memory Modules one at a time and tested by the PE's. Records not to be deleted are then copied to the output. The Key Processor can also be used in this operation. In this case,



O: Sort/Merge own records
E: Sort/Merge own and neighbors' records

Figure 5. Sort

the search keys of the records to be deleted are stored in KP and tests against them are made.

More complex operations such as Divide or Set Intersection, involving multiple Joins or Projections, are also possible. In performing joins or projections, the situation may arise where the memory space in the Key Processor becomes full. If this should occur, each PE continues to process its segment until the end but will place each record that does not fit in KP on an overflow list. When all the PE's have completed this operation, the KP's memory space is cleared and the overflow records are processed. For those cases where the key value or attribute string is too large to fit into KP, the value is hashed to a value that will fit. KP in this case, is put in the Entry Number mode and the number is reported back to the requestor. The actual value is then placed on the list corresponding to the number that KP returns. In subsequent operations when the actual value is needed, the appropriate list is traversed to obtain this value. Thus, Key Processor can also be used as a hashing aid.

5. DATA BASE COMPUTER STATUS

The Data Base Computer described in the previous section performs all of the functions required to support a CODASYL or Relational Data Base Management System. In addition to the functionality offered, users of these systems require the capability to rollback the data base to a previous commit point and to recover from errors and system failures. These rollback and recovery features are currently being designed to be incorporated into DBC. Techniques similar to those used in current DBMS's appear appropriate here.[11] An audit trail tape may be connected to the DBC Controller to hold copies of the data to be changed and the changes requested.

Although corrupted data will always occur due to user or program errors, the effects of hardware failures can be reduced through the use of fault tolerant techniques. Since DBC is constructed from sets of identical hardware modules, additional modules can be added as spares and switched into operation when a failure in an existing module is detected. One such "failure tolerant" configuration is shown in Figure 6. In this case, any number of spare PE's and M's may be added and interconnected as shown. To ensure that there is no single point of failure in this system, DBCC, PTDC, and the interconnecting buses would all have to be replicated.

Another research effort now in progress is directed toward determining the types of configurations and situations in which the Data Base Computer would be used. For large scale host systems (1100's, 370's, etc.), DBC would probably

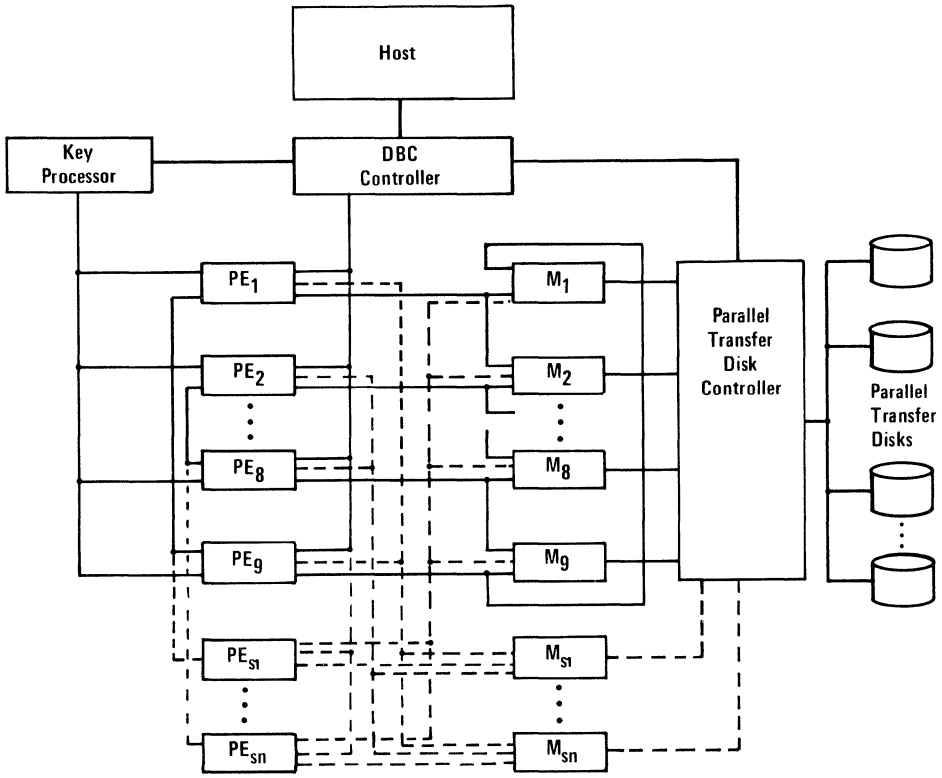


Figure 6. Failure Tolerant Configuration

be constructed as previously described in this paper. In order to reduce the cost of DBC for use with minicomputer hosts, the Controller portion may be implemented as software in the host. For an end-user facility, communication hardware may be added. In any case, the modularity of DBC lends itself to incorporation over a wide range of system configurations.

Currently, a second round of research effort is occurring. Modifications that may be necessary to support the previously discussed extensions are being investigated. Components, such as microprocessors for the PE's and random access memory chips for the M's, are being selected. The software structure and DBC operation algorithms are being prepared. Finally a performance model is being constructed to simulate the operation of DBC to confirm the analytic calculations of from one to two orders of magnitude performance improvement over current DBMS's operating on general purpose computers.

6. CONCLUSION

Initial investigation found that the original Data Base Computer approach[5] to be most appealing in light of Sperry Univac's need to support existing customers' data bases and applications while at the same time offering them cost/performance improvement and additional capabilities. Analyzing current users applications and projecting their future needs resulted in a set of functional requirements for a data base computer. Applying these requirements to the proposed design[5] resulted in a revised architecture that offered the benefits sought at a price and level of complexity that is extremely attractive. Improvements to the design to aid in recovery operations and the performance modeling should be complete by the time this paper is published. The result should be a special purpose computer that will significantly improve the users ability to mange the ever growing body of data characteristic of our modern technological society.

7. REFERENCES

1. Leavitt, D., "Two-Year Project Pays Off: 'Back-End' DBMS Succeeds," *Computerworld*, Feb. 20, 1978, p. 1.

2. Verity, J., "Data Base Growth Spurs Back-End Unit Evolution," *Electronic News*, Mar. 20, 1978, p. 36.

3. Su, S. Y. W. & G. J. Lipovski, "CASSM: A Cellular System for Very Large Data Bases," *Proceedings International Conference on Very Large Data Bases*, Sept. 1975, pp. 456-472.

4. Ozkarahan, E. A., S. A. Schuster, & K. C. Smith, "RAP — Associative Processor for Data Base Management,"

AFIPS Conference Proceedings, Vol. 44, 1975 NCC, June 1975, pp. 379–388.

5. Banerjee, J. & D. K. Hsiao, "DBC — A Database Computer for Very Large Databases," *IEEE Transactions on Computers*, C–28, No. 6, June 1979.

6. Sperry Univac, *Data Management System (DMS 1100)*, UP–7907 Rev. 3, 1977.

7. Bray, O. H., & H. A. Freeman, "Data Usage and the Data Base Processor," *Proceedings ACM '78*, Dec. 1978, pp. 234–240.

8. Chamberlin, D. D., & R. F. Boyce, "SEQUEL: A Structured English Query Language," *Proceedings ACM Workshop on Data Description, Access, and Control*, 1974, pp. 249–264.

9. Sperry Univac, *Query Language Processor (QLP 1100)*, UP–8231 Rev. 1, 1977.

10. Ampex Corp., *PTD–930x Parallel Transfer Drive*, Product Description 3308829–01, Oct. 1978.

11. Sperry Univac, *Data Management System (DMS 1100) System Support Functions*, UP–7909 Rev. 4, 1978.