THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 61, No. 9, November 1982
Printed in U.S.A.

Database Systems:

Design and Implementation of a Production Database Management System (DBM-2)

By T. C. CHIANG and G. R. ROSE

(Manuscript received September 18, 1982)

The DBM-2 is a transaction-oriented database management system designed to track activities within an organization. Being a production system, DBM-2 has different requirements and design criteria than exploratory database management systems. To support such applications in a production environment, DBM-2 is required to have high performance and flexibility. An extended entity-relationship (E-R) data model provides a basis for DBM-2 in meeting the requirements. The extended E-R data model simplifies the handling of existence dependency between data records by DBM-2. This paper presents the design and implementation of DBM-2. Special attention is given to the implementation of "associations" between records, because the implementation is critical for supporting the E-R model.

I. INTRODUCTION

The DBM-2 is transaction-oriented database management system designed to track business activities in an enterprise. The first application is the Loop Maintenance Operation System (LMOS-2),[1] a transaction system for tracking repair activities of an operating company. The DBM-2 together with the LMOS-2 application software has been in field trial since October 16, 1980, and will be deployed in most of the Bell operating companies in the United States in 1982. Thus, DBM-2 is a real production system. It is programmed in C,[2] and runs under the *UNIX** operating system version 4.0[3] on a DEC PDP 11/70 computer.† Work is being done to transport DBM-2 to a DEC VAX 11/780 computer.

---

* *UNIX* is a trademark of Bell Laboratories.
† DEC is a trademark of the Digital Equipment Corp.

To support transaction systems with tracking applications in a production environment, DBM-2 is required to have high performance and flexibility. A transaction system is one that performs a predefined set of tasks. A transaction performing a task is normally required to have a fast response time and may involve many database retrievals and updates. A tracking system normally has a large database and a heavy transaction load. These requirements put a premium on DBM-2 performance. Furthermore, the functional requirements of our application, probably typical of most transaction systems, are ever changing. This implies that DBM-2 must provide data independence and enable easy change to the database structure and definition. Data independence means that a database can be changed without affecting existing application programs. Ease of change implies that we have to have tools for making the changes and modular system design for isolating changes in the system. However, DBM-2 is not required to have a high-level query language for arbitrary queries, since a transaction accepts a fixed format input from a CRT terminal to perform the predefined task.

The DBM-2 has been managing a database successfully in a telephone repair application. The database is large in volume and complex in storage structures. It has about 300-million bytes of data in 30 files. There are 15 cross-record relationships, and up to 80 fields per record. The transaction load on the application system is rated at 4000 transactions per hour, where the transactions average eight database requests each. The application system is available 22 hours per day. Many new fields as well as new views of existing fields have been added to the database. Files have been reorganized to improve performance. These changes have been made without requiring changes in the existing application programs. Some of the changes have been made without database conversion.

Overviews of DBM-2 have been presented in Refs. 4 and 5. This paper elaborates on how DBM-2 provides flexibility (mainly data independence and ease of change) without sacrificing performance. This paper also shows the judicious choice of modern database and software engineering technology for a successful development of a production-oriented database management system that is running on a minicomputer. There are many factors contributing to the success of DBM-2. They include the use of an extended Entity-Relationship (E-R) model, the internal data structures, mapping between internal and external structures, a robust file system, and the program and process organizations. One of the novel ideas reflected in the design of DBM-2 is the extension of the existence dependencies semantics of the E-R model. In the extended E-R data model existence dependencies among records are considered as a property of an "association." Special attention is

given to the implementation of associations between records, because the implementation is critical to the support of the E-R model.

Section II reviews the extended E-R model. Section III describes the internal data structures and their interface. Section IV describes how the mapping between the external views and internal data structures is done. Sections V and VI present the software architecture and performance statistics of DBM-2. Finally, Section VII presents the conclusion that includes a summary on how each design decision affects the flexibility and performance objectives.

II. EXTERNAL VIEWS

The DBM-2 supports multiple external views of a database using an extended E-R model. The advantage of using the E-R model is that it provides: (*i*) rich enough constructs to capture the semantics of the application, (*ii*) a simple user interface, (*iii*) a high level of abstraction to hide internal data structures from user programs, (*iv*) mechanisms for achieving a high level of performance, and (*v*) structures for ease of implementation. For example, the E-R data model, as defined later, will allow not only relationships between records, but also various existence dependencies between records to be expressed explicitly at the user level. These two kinds of semantics are very important for our applications. Even with the constructs for handling the added semantics, the total number of objects in the data model is small and the operations on the objects are simple. Therefore, it is possible to define a simple user interface in terms of the data model. Such a simple interface encapsulates the complexity of the internal data structures. On the other hand, the constructs in the data model reflect some important aspects of the internal data structures so that high performance and ease of implementation can be achieved. All of these points will be explained in this section, and in Sections III and IV.

2.1 The extended E-R data model

From a user's (external) view, a database is considered as a collection of files and a collection of "associations" among records. A record represents an entity (e.g. a customer or a trouble report) in the real world, while an association represents a set of relationships among entities (e.g. a trouble report by a customer or a repair person responsible for a reported trouble). A file is viewed as a two-dimensional table, where the columns are fields and the rows are records. A relationship can be viewed as a named link between two records of perhaps two different files. In fact, as can be seen in Section III, an association is implemented using a linked list. The use of named links enables one to retrieve an associated record without a key search, and without needing to know the implementation of the link. Thus, an

association provides a convenient way for navigation in a database. In addition to navigation, associations are also used to express update dependencies. Update rules are defined as properties of associations. As described in Ref. 4, four "coupling factors" to represent four sets of update rules have been identified for the particular application. For example, a very tight coupling factor has the following definition.

Definition: Given two files, E1 and E2, and an association A between them, A is said to have a very tight coupling factor from E1 to E2 if and only if:

1. Insertion of a record e2 in E2 is permitted only if there is a record e1 in E1 that record e2 can be associated with,
2. Deletion of a record e1 in E1 implies the deletion of all its associated e2's in E2,
3. No deletion of a record e2 in E2 is permitted if there exists an associated e1 in E1.

Note that coupling factors are directional. In the above definitions, we assume the existence of a record e2 in E2 is dependent on that of a record e1 in E1. A "very tight" coupling factor, in a way, indicates that the two types of records are semantically very close. Closeness in semantics usually is reflected at the physical database level; closely associated records are stored together in one physical record. This implies that associated records can be retrieved without multiple disk accesses, which is extremely important for performance.

The definitions for the other types of coupling factors are relaxations of one or more rules in the above definition. A "tight" coupling factor is defined as the one that satisfies rules 1 and 2 above, a "regular" coupling factor satisfies only rule 1, and a "loose" coupling factor satisfies none. For some applications, one may need to define other coupling factors. An example is a coupling factor satisfying rules 1 and 3.

Coupling factors have many interesting properties, e.g., transitivity. For example, if the deletion of e2 in E2 depends on the deletion of an e1 in E1, and the deletion of e1 depends on the deletion of an e3 in another file E3, then the deletion of e2 depends on the deletion of e3. The DBM-2 performs automatic updates of associated data items according to the declared update rules. This is not only convenient for programming, but also necessary for performance. Process and message switching overhead can also be avoided. (Process organization around DBM-2 is discussed in Section V.) Another interesting property is the interference between two coupling factors. For example, let us assume that two associations A and B between files E1 and E2 have coupling factors C1 and C2, respectively. If C1 and C2 both contain rule 1, but the insertion dependencies are in opposite directions (i.e.,

the dependency of E1 on E2 versus the dependency of E2 on E1), then record insertions cannot be performed on either E1 or E2. The problems of interference between two coupling factors have been discussed in Ref. 6 in detail.

2.2 Data Manipulation Language

A Data Manipulation Language (DML) based on the E-R model is provided to the application programmers as a program interface to the database. The host language for the DML is the programming language C. The DML is a set of C functions for retrieval, insertion, modification, and deletion of records. The DML is divided into a set of regular DML commands and "associated" DML commands. The regular DML commands operate on a single file, while the associated DML commands operate on files via associations. Most of the regular DML commands have the following format:

```
setid = command_name(file, view, condition);
```

where setid is the identifier for the set of records on the command that is operating, file is the file name, view is a projection of the file, and condition a Boolean combination of field name and field value pairs. The associated commands have the format:

```
command_name(setid, assoc, file, view, condition);
```

where setid is the id returned by a previous DML call, and assoc the name of the association. An associated DML command may also return a setid.

The following is an example of the use of DML. (Note that the syntax used in the example is not exact.) Consider a user's view of a database consisting of two files, DEPT and EMP, and an association between these files called DE. DEPT and EMP store, respectively, information about departments and employees. DE captures the semantic of "department of an employee" relationship. The files have the following fields:

```
DEPT(DEPT#, #_of_emp, manager_name);
EMP(EMP#, emp_name, emp_salary);
```

The association DE is 1:*m* and has a loose coupling factor. The database for the example is shown in Fig. 1.

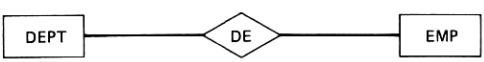


Fig. 1—Database example.

Suppose a user wants to print the manager's name and all the employee names for the employees who earn over $30,000 and work in a department that has more than 30 people.

The following DML calls may be used:

```
V1 = {manager_name};
V2 = {emp_name};
r = Dretrieve(DEPT, V1, "#_of_emp > 30");
while ((r = Dgnext(V1, r)) != EMPTY){
        r1 = Daretrieve(r, DE, EMP, V2, "emp_salary > 30000");
        while((r1 = Dgnext(V2, r1)) != EMPTY)
                printf("%s %sn", V1.manager_name,
                        V2.emp_name);
}
```

Dgnext in the above example is a DML command for getting the next record in the set starting with the first retrieved record. In the above example, the Dretrieve command retrieves a set of records that satisfies the selection criterion and returns a set identifier r. The Dgnext calls fill in field values in the views (i.e., V1 or V2), and move a current record pointer to the next record in the set. For each record in set r, a Daretrieve is called to retrieve associated records in EMP. For each record in an r1 set, the program prints the names of the manager and employees found in V1 and V2.

Since the association DE has a loose coupling factor, a deletion of a particular department will not affect the existence of the employee records. However, if the coupling factor were very tight, the deletion of a department record would trigger the deletions of all the employee records associated with the department record.

2.3 Data Definition Language

The Data Definition Language (DDL) is provided for the Database Administrators (DBA) to interface with the database. There are three commands in the DDL: (*i*) Dfile, (*ii*) Dfield, and (*iii*) Das. Dfile is the command for defining a file with a set of parameters:

```
Dfile filename filecode nxtfile lnkcode nfield vfield
```

where filename and filecode are respectively the file name and internal file code, nxtfile is the file code for the next internal file in a link, lnkcode is the code for the link, and nfield and vfield are respectively the number of fields and the beginning byte of the variable fields. Note that an external file may be implemented as many linked internal files. The implementation of the mapping between an external file and the

internal files and the use of nxtfile and lnkcode are discussed in Section IV. Dfield is the command for defining a field in a file and has the following format:

```
Dfield name code type class begin length usecode
```

where name and code are respectively the field name and field code, type is the field type (e.g., INT and CHAR), class is either fixed or variable, begin and length are respectively the beginning byte and the length of a fixed field, and usecode is the user's field code. The command Das that defines an association between files has the following format:

```
Das name type cfactor file1 I1 file2 I2
```

where name is the association name; type is either 1:1, 1:*m*, or *m*:*n*; cfactor is the code for coupling factor; file1 and file2 are the file names for the files involved; and I1 and I2 are the implementation classes for the two sides of the association (e.g. embedded key, linked keys, or link).

A sequence of DDL commands for a file or an association is stored as a *UNIX* file. The DDL processor (DDLP) converts the source files into a machine readable form, called a database catalog (DBCAT) which is a set of tables defined as C structures. The members in such a C structure store values translated from the arguments of the DDL commands. There are a file table, a field table, and an association table. The DBCAT is read into main memory at system initialization by DBM-2 to perform mapping.

For the database example in Fig. 1, the DDL source files may look as follows:

```
dd1.DEPT:
   Dfile DEPT 1 -1 -1 3 -1
   Dfield DEPT# 0 CHAR FIX 0 5 123
   Dfield #_of_emp 1 INT FIX 6 2 124
   Dfield manager_name 2 CHAR FIX 8 20 125
dd1.EMP:
   Dfile EMP 2 -1 -1 3 -1
   Dfield EMP# 0 CHAR FIX 0 5 126
   Dfield emp_name 1 CHAR FIX 5 20 127
   Dfield emp_salary 2 INT FIX 25 2 128
dd1.DE:
   Das DE 1:m LOOSE DEPT EMBKEY EMP LNKKEY
```

A -1 in a command line means "not applicable." Note that to change fields, add fields, and delete fields, the database administrator only has to change the DDL source files using a database editor provided by DBM-2 and reprocess the source files by using DDLP. The database editor knows about both the external and internal data structures of the database. It has a set of commands for changing the definitions of these data structures, such as adding a field and deleting a field. The existing application programs need not be changed. In some cases, changes have been made without database conversion. For example, if a new external view of a field is to be added to the database, it is only required to add an entry into the field table for a file. This requirement is also true for adding an internal field to a file with a variable record format, since the number of fields in such a record can be varied. These facilities permit quick response to system change requests. The flexibility gained has proven important during the field trial. At times, it has spared the system from expensive database conversions when moving between different versions of application software.

III. INTERNAL VIEW AND DATA STRUCTURES

The internal view is a simple picture of the otherwise quite complex internal data structures. The internal data structures include multi-level indexes, variable and fixed length records, variable and fixed length fields, records with variable numbers of fields, pointers among records, integration of two external records into one internal record, partition of one external record into several internal records, and up to 15 different field types (e.g. INT and CHAR). The internal data structures are designed for high performance and conservation of disk space. The variable record format also provides a mechanism for achieving flexibility.

A variable format record can have a variable number of fixed fields and variable fields (up to the limit imposed by the size of DBCAT). A record is partitioned into a fixed-field portion and a variable-field portion. In the fixed-field portion, only field values are stored for individual fields. In the variable-field portion, a field length is prefixed to a field value. If a field is not presented in the record, only the field length of zero is stored. Furthermore, DBM-2 performs data compression on the basis of a record. The compression algorithm converts three or more contiguous blanks and zeros into two bytes, one byte storing the length of the character sequence before compression and the other specifying a sequence of either blanks or zeros. Hence, storage space can be saved. Since a variable format record allows variable number of fields, deleting or adding a field will not require database conversion.

The internal view consists of a set of internal files (IFs), each of which consists of a set of internal records (IRs). The records have fixed formats, i.e., fixed record length, fixed field length, and fixed number of fields per record. There are operations for getting a record (getrec), adding a record (addrec), deleting a record (delrec), and replacing a record (putrec). All the operations are defined on a single record for a given key.

There may be links between records. Links provide a formal mechanism for creating references from one IR to one or more distinct IRs. The purpose of having links is to eliminate the use of pointers at the higher-level DBM-2 programs. The advantage is that changes in the link-structure implementation, e.g., when a pointer is changed from an integer to a long integer, would not require changes in the higher-level DBM-2 programs. Links are used to implement indexes, associations, and record partition. (Record partition is defined as the partition of an external record into two or more internal records.) Examples of such applications of links are illustrated in Section IV.

A link structure is the key to the simple representation of complex internal data structures in the internal view. A link is defined as a linked list that consists of a header internal record of a particular type and a set of member internal records of another type. The link header must be accessible by key, while the members may be accessed by key or via the link. For performance, a link may be stored on a common block of physical disk space. In this case, the link is referred to as a closely held link. Normally, the members of a closely held link can be accessed only via the link. Closely held links are used to implement record integration, which is defined as an integration of two external records into one internal record.

There are two distinct sets of operations on links. The first set is the set of single-link operations, and the other is the set of multiple-link operations. The single-link operations include: (*i*) get header (using getrec), (*ii*) get member (lgetrec), (*iii*) attach a member to a link (link), and (*iv*) remove a member from a link (unlink). Since headers or members are records of one or more files, record operations can be applied to them (except for the closely held linked members). The existence of a link requires the existence of the header. Therefore, a deletion of a header implies the deletion of the instance of the link headed by the header. For the closely held links, however, it also implies the deletion of all linked members.

Conventionally, secondary indexes are implemented as inverted lists or trees. The conventional way for processing secondary indexes requires some Boolean operations on sets of pointers. The multiple-link operations are for processing secondary indexes. While the secondary indexes may still be implemented as trees or inverted lists, the opera-

tions on these structures are done by using the multiple-link operations. One does not have to know about pointers or implementations. A secondary index of a field value is viewed as a link. The record that stores the indexed field value serves as the header of the link. The members are the data records that have the field value. The multiple-link operations are specially defined Boolean operations on the links. For example, the intersection operation, ^, is defined as the operation on two links that produces a link with a virtual header that is a temporary record created for the identification of the link. The members are the intersection of the two sets of members of the two links involved in the link intersection. Except that the virtual header is not accessible, all other link operations can be applied to the resulting link. For example, given links L1, L2, and L3,

$$X = L1 \text{ ^ } L2$$

is a link, and likewise

$$Y = X \text{ ^ } L3$$

is a link. The lgetrec operation can be applied to members of either X or Y.

IV. MAPPING

The DBM-2 hides the internal view from users by providing a mapping function between the internal and external views. The function has two levels: (*i*) field level and (*ii*) record level. The mapping function is table driven. The information that is needed for the mapping is kept in various tables. There are tables for the descriptions of files, fields, associations, and links. For example, in a file-description table, each entry stores information about a file, such as external file name, associated internal file codes, and pointers to the field table entries for all its fields. The tables are created by using DDL and stored in DBCAT as described in Section 2.2.

4.1 Field-level mapping

At the field level, the mapping function translates 15 different field types into the null terminated character string format (as used in a C program), which is the canonical representation of an external field value. Semantically, an external field value can be one of the many types, for example; ALPHANUMERIC, STANDARD_TIME, and LONG_TIME. The ALPHANUMERIC type is externally a string of letters and/or digits; STANDARD_TIME is a string with a specified format appropriate to time representation, and so on.

To support multiple external views of a field, one internal field type may be mapped into several different external field types. For example,

an internal field that is a LONG (long integer, 32 bits for the PDP 11/70) can be mapped into a STANDARD_TIME type (e.g., 01-01-81 0001A) or LONG_TIME type (e.g., 347145200, the number of seconds passed since the beginning on January 1, 1970).

A user can define new field types by supplying a routine that interprets the new type. A relinking of DBM-2 and a change in DBCAT are necessary then to install the new type.

4.2 Record level mapping

At the record level, the mapping function handles record integration and record partition. It also translates between associations and links. There are three basic methods for the implementation of an association: (*i*) links, (*ii*) embedded foreign keys, and (*iii*) links of foreign keys. A link is the data structure defined in Section III. In a particular record, an embedded foreign key is a key of another record embedded in the given record. A link of foreign keys is a link that consists of a header that is the given record and members that are keys of other records. Figure 2 shows examples of an embedded foreign key and a link of foreign keys, where r1 represents a record and k*i*, $i = 1, 2, 3$, represents a key.

A variety of implementation methods can be derived from these three basic methods. For example, a one-to-many association between records of two different types can be implemented as an embedded key on one side of the association, and a link on the other side.

In the case of record partition, a link connects two or more internal records of different types with one of the internal records being the header and the rest of the records being members of the link. The mapping function accesses the internal records through the link mechanism and concatenates all the internal records to make an external record. Figure 3 shows how the information stored in the file table in DBCAT is used to do the record partition mapping.

In Fig. 3, E_file1 is the external file name, I_filei, i = 1, 2, 3, are the associated internal file codes, NIfilei, i = 2, 3, points to the next internal

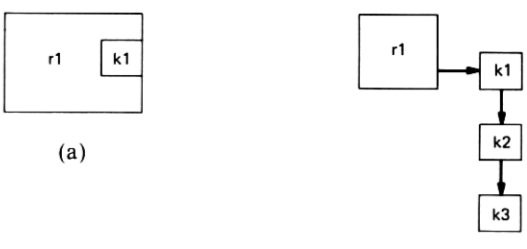


Fig. 2—Examples of foreign keys. (a) Embedded key. (b) Linked keys.

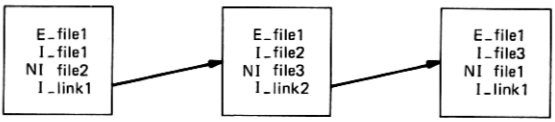


Fig. 3—Record partition.

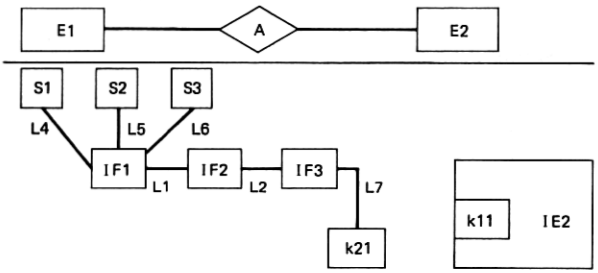


Fig. 4—Example of record level mapping.

file, and I_linki, i = 1, 2, are the links for linking the internal files, and a -1 signifies the end of the chain of file-table entries. The file-description table entries for all internal files of an external file are chained by using the NIfile field.

Secondary indexes are implemented as links. The link operations described in the previous section are used to process the secondary indexes.

In summary, for the record-level mapping, the mapping function will interpret the links and map them into associations, record partitions, or secondary indexes. An example is shown in Fig. 4 to illustrate such mappings in general. Figure 4 also shows the contrast between a simple external view and complex internal data structures. As shown, E1 and E2 are two external files, and A is the association between E1 and E2. IF1, IF2, and IF3 are the internal files for E1, linked together by L1 and L2. S1, S2, and S3 are secondary index files linked to the data files via L4, L5, and L6. k21 is a key field of records in E2 linked to IF3 via L7. This is the implementation of A (linked key) on one side. k11 is a key field for records in E1 and is the implementation of A (embedded key) on the other side.

V. SOFTWARE ARCHITECTURE

In this section, software architecture is described in two parts: (*i*) process organization, and (*ii*) program modules. Process organization is defined as the relationship of the DBM-2 process to other processes in the system. The definition of the DBM-2 process itself in terms of program modules and functional partition is the second topic.

5.1 Process organization

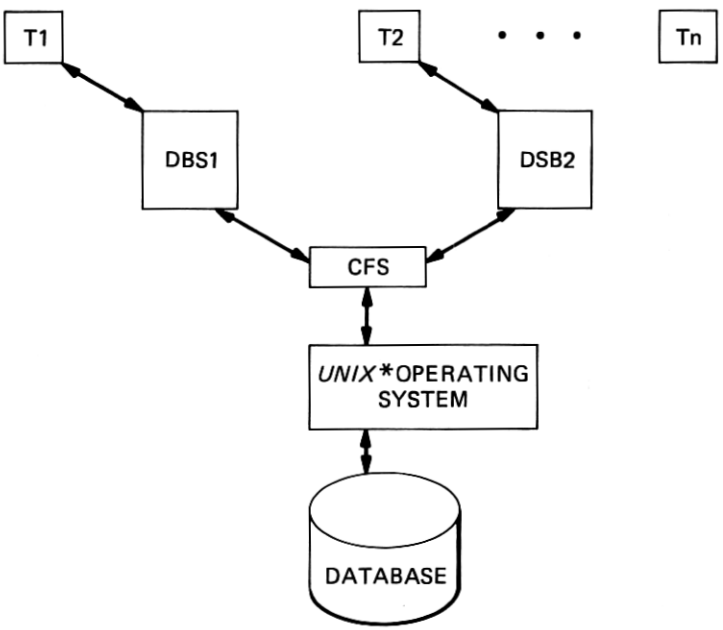
The DBM-2 is implemented as a user process under the *UNIX* operating system, version 4.0+, a version of the *UNIX* operating system augmented for transaction applications with dynamically allocated shared memory. A robust file system (CFS)[6] is used instead of the *UNIX* file system. The *UNIX* file system is considered inefficient for production-oriented database systems, because of its long access time and lack of crash-recovery mechanism and concurrency control. The long access time of the *UNIX* file system is due to its multiple-level directory structure and noncontiguous file organization. CFS is an extent-based system; that is, a file is divided into large chunks of contiguous storage spaces called extents. Each extent in turn is further divided into blocks. The CFS provides block I/O, concurrency control, and crash recovery, and also efficiently maintains a buffer pool as a cache for the data blocks. A transaction is considered as the unit of consistency for both concurrency control and crash recovery. A transaction begins and ends in a consistent database state. All records intended for update are locked by CFS on the behavior of a transaction, until a "commit" command is issued by the transaction. Upon receiving the committed message, CFS issues an atomic write of all the updated records to a temporary file on disk. Then, individual writes of records to the actual database files follow. CFS releases the locks after all the writes are done. For system crashes due to software failures, the updated records that have not been committed by the active transactions will not be written to the database. The updated records that have been committed and written to the temporary file will be written to the database at system restart time. A journal tape is also maintained for recovery from disk crashes. Detailed discussions on the *UNIX* operating system, 4.0+, and CFS are beyond the scope of this paper.

A DBM-2 process (DBS) communicates with other processes via messages and shared memory. The data sent between an application process and a DBS are in a well-defined format that consists of a list of ⟨field name, field value⟩ pairs.[8] The list resides in a work-area shared between an application process and a DBS. DBM-2 is not involved in scheduling and process management, which are done entirely by the *UNIX* operating system. By design, there could be one or more DBS and CFS for an application system. For the LMOS system in the field, two DBS processes are spawned at system initialization and locked in the main memory until the system is brought down. (Of course, the DBS text is shared.) Once a transaction is assigned to a DBS (via a DBOPEN call), that DBS will serve the transaction until the transaction either aborts or normally terminates. There is only one CFS process serving the two DBS processes on a first-come-first-serve basis. The

serving time in CFS has been proved to be small. The process organization is illustrated in Fig. 5, where T1, T2, . . . , Tn represent transaction processes. The arrows represent transaction data flow.

5.2 Program modules

The DBM-2 software has a modular design that minimizes dependencies among the software modules. The DBM-2 software is functionally divided into several modules: (1) DML processor (DMLP), (2) Mapping Function (MAP), (3) Index Access Manager (IAM), (4) Data Format Manager (DFM), (5) Record Access Manager (RAM), (6) DDL Processor (DDLP), and (7) Database Editor (DBE), and other DBA tools. The interface between two modules is simple. An interface is a set of *C* function calls specifying the operations for a particular module to perform. The interprocess communication mechanism is hidden below the interface. This ensures flexible implementation. Dependent on memory size and execution time requirements, each module can be implemented as either a process or a subroutine. For example, in one version of DBM-2, RAM resided in the same address space as DMLP, MAP, IAM, and DFM. However, as the system evolved, the text space of the combined DBM-2 process was exhausted. Subsequently, RAM was



* TRADEMARK OF BELL LABORATORIES

Fig. 5—Process organization.

moved to the CFS address space. Successive implementations of modules have had increased functionality. This is particularly true of RAM, for which the design of the most general version is discussed below.

In addition to these software modules, the DBCAT file stores information about the database, e.g., file schemata and association definitions. The graphical representation of the software architecture for DBM-2 and the surrounding processes is shown in Fig. 6, where E1, . . . , En represent the external views in the application processes.

The DBCAT is created by the DBA using the DDL, and is updated by the DBA using the DBE. The DDL processor has been implemented using "Yet Another Compiler Compiler."[9] Other DBA tools includes audit and load/unload programs.

The DMLP has two parts. The first part is a set of subroutines residing in the user program space. Within each of the subroutines, the parameters of a DML call are set up in the shared work-area and a message is sent to the second part of DMLP on the DBM-2 side. The second part of DMLP then interprets and executes the DML command by calling upon all the modules involved. The result is sent back to the first part of DMLP on the user side. The mapping function performs the translations that have been described previously.

The IAM handles the semantics of access paths, e.g., the maintenance of indexes, by using the mechanism provided by RAM. For example, an index file that stored the indexes is just another internal file to the

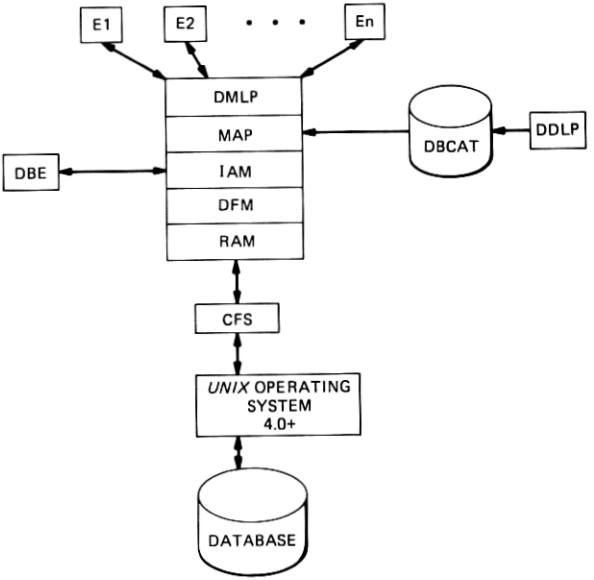


Fig. 6—The DBM-2 software architecture.

RAM, but to the IAM, it represents an access path to the data file. The RAM provides access methods to both the index files and the data file, while IAM must maintain the consistency between all the index files and the corresponding data file.

The interface to IAM consists of a set of functions for retrieving and updating the indexes. For example, the function getindex(file, select) performs an optimal search and returns a set of indexes for file, according to the selection criterion, select.

The Data Format Manager (DFM) handles variable format records and data compression. Some flexibility can be achieved by using variable formatted records, because fields can be added, changed, and deleted without data conversion. Data compression coupled with variably formatted records conserves disk space, but consumes CPU time.

The RAM provides data access and maintenance functions designed to support the internal view. These functions are above the level of the CFS. The RAM also provides access to the internal records. Each internal record is partitioned into a data area and an (optional) key area. Apart from this partitioning, no interpretation of content is performed by RAM.

The RAM module is a simple set of *C* function calls that is an implementation of the operations on records and links described in Section III. The basic record retrieval function getrec(file, rcd, key) illustrates this simplicity. All successful retrieval operations return a retrieval identifier r for use in subsequent linking and update operations. For example, the function linkrec(link, r, r′) creates a link between the two internal records identified by r and r′. There are twelve functions in all.

The implementation approach for RAM IF operations is to provide a limited number of underlying physical file organizations and to map internal files onto physical files via control tables. This is accomplished by supplying a specific algorithm that corresponds to each general operation for each type of file. Initially two physical file organizations have been implemented.

The first is a data organization for binary-search that provides access to internal records via a full key and also, with predefined restrictions, to subsets of internal records that match the significant bytes of a given key. This file type, which is restricted to main memory applications, is designed to minimize retrieval time at the expense of insertion and deletion time.

The second is a hash organization intended primarily for disk files. It supports both fixed- and variable-length records and permits a mixture of record types to be recorded. The latter capability is the vehicle for providing closely held records.

Links are implemented by reserving a fixed-length link area within

each internal record in which IR addresses are recorded. The standard format for IR addresses is block and mark, where mark is a unique identifier with respect to a particular block. Direct access is performed by reading the block and searching for the particular mark. This scheme permits internal records to be moved within a block for garbage collection.

There is also the concept of a set which is a subset of internal records in an internal file with partial keys that match a given partial key. Once a set is defined, its internal records may be accessed sequentially.

VI. PERFORMANCE

In general, a database management system adds overhead to an application system in two areas: (*i*) main memory usage, and (*ii*) processing time. The version of DBM-2 presently supporting the first application in the field requires about 95 Kbytes of main memory, of which about 30 Kbytes are in the data space. DBM-2 CPU time is discussed from two points of view: (*i*) absolute CPU time required to retrieve a single external record, and (*ii*) percentage of transaction CPU time that is spent in DBM-2.

The CPU time for DBM-2 to retrieve a set of fields of a record from a file is obtained empirically as follows:

$$t = 5 + 0.6f + I,$$

where t is the CPU time in ms, f is the number of fields specified by a user, and I the interprocess communication time. For example, retrieving 33 fields of a typical record of the application database takes about 35 ms, of which 10 ms are in I. From a recent study of system performance,[10] the CPU time for a typical transaction is about 1273 ms, of which 363 ms are taken by DBM-2 (120 ms of this time are spent in interprocess communication). So, DBM-2 requires about 28 percent of the total CPU time used by a typical transaction. The average fraction of CPU resources consumed by DBM-2 during a day of operation is about 22 percent.

VII. CONCLUSION

In summary, DBM-2 provides a flexible production applications. Fields can be added or changed without reprogramming, and in most cases, without database conversion. Application programs are easier to write because of the encapsulation of internal structures. The cost of the added flexibility is increased CPU utilization: at most 20 to 30 percent for our application.

The many factors contributing to the success of DBM-2 have been reviewed. The extended E-R model, the table driven software, the

variable-record format, and the modular and multiple-level design of DBM-2 have provided us with flexibility. Reflecting data semantics in the physical database design, process organization, the large data cache, the interprocess communication primitives, and the use of a robust file system (rather than the *UNIX* file system) contribute to the good performance.

VIII. ACKNOWLEDGMENTS

The development of the DBM-2 project has been done by the members of the Database Systems Group in the Advanced Transaction Systems Department, under the direction of R. F. Bergeron. We would like to thank Mr. Bergeron for his support and advice. We would also like to thank the other members of the group for their contributions to the project. Special thanks go to A. Weinstein for his implementation of the Record Access Manager for the first application, and to D. H. Carter for his implementation of the robust file system. We would also like to thank the Advanced Operating and Communication Group for providing the *UNIX* operating system related support.

REFERENCES

1. R. L. Martin, private communication.
2. B. W. Kernighan, and D. M. Ritchie, *The C programming Language*, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978.
3. D. M. Ritchie, and K. Thompson, "The *UNIX* Time Sharing System," B.S.T.J., *57*, No. 6, Part 2 (July–Aug. 1978), pp. 1905–1929.
4. T. C. Chiang, and R. F. Bergeron, "A Data Base Management System with An E-R Conceptual Model," Proc. of Int. Conf. on Entity-Relationship Approach to System Design and Analysis, December, 1980.
5. T. C. Chiang, and G. R. Rose, "Design and Implementation of An E-R Data Base Management System (DBM-2)," Proc. Second Int. Conf. on Entity-Relationship Approach, October, 1981.
6. T. C. Chiang, unpublished work.
7. D. H. Carter, unpublished work.
8. M. Rochkind, private communication.
9. S. C. Johnson, "YACC: Yet Another Compiler Compiler," Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, New Jersey.
10. J. Tsay, private communication.