A Database Management System for the Federal Courts

JACK R. BUCHANAN, RICHARD D. FENNELL
Federal Judicial Center
AND
HANAN SAMET
University of Maryland

A judicial systems laboratory has been established and several large-scale information management systems projects have been undertaken within the Federal Judicial Center in Washington, D.C. The newness of the court application area, together with the experimental nature of the initial prototypes, required that the system building tools be as flexible and efficient as possible for effective software design and development. The size of the databases, the expected transaction volumes, and the long-term value of the court records required a data manipulation system capable of providing high performance and integrity. The resulting design criteria, the programming capabilities developed, and their use in system construction are described herein. This database programming facility has been especially designed as a technical management tool for the database administrator, while providing the applications programmer with a flexible database software interface for high productivity.

Specifically, a network-type database management system using SAIL as the data manipulation host language is described. Generic data manipulation verb formats using SAIL's macro facilities and dynamic data structuring facilities allowing in-core database representations have been developed to achieve a level of flexibility not usually attained in conventional database systems.

Categories and Subject Descriptors: H.4.2. [**Information Systems Applications**]: Types of Systems—*decision support*; H.2.3 [**Database Management**]: Languages—*data manipulation languages*; J.1 [**Computer Applications**]: Administrative Data Processing—*government*

General Terms: Design

Additional Key Words and Phrases: SAIL, network model

1. INTRODUCTION

The administrative case-tracking activities of the federal courts have traditionally been accomplished through primarily manual record-keeping techniques. However, in response to increasing case processing requirements within the federal judiciary and as a result of the passage of the Speedy Trial Act of 1974 [23], the

Authors' addresses: J. R. Buchanan, Decision Making Information Systems, Inc., McLean, VA 22101; R. D. Fennell, Federal Judicial Center, Washington, D.C. 20005; H. Samet, Computer Science Dept., University of Maryland, College Park, MD 20742.

federal courts were given the mandate and necessary funding to develop large-scale computing systems to provide automated support for court management. It was a unique opportunity to manage the delivery of high technology to this traditionally conservative institution [3, 4]. Many major application areas were identified [5, 11, 16], including management of criminal, civil, bankruptcy, and appellate cases, scheduling and calendaring, jury management, document processing, archival record maintenance, and electronic mail. Databases were planned that would eventually hold millions of records within both national and distributed data centers.

The newness of the application areas, together with the experimental nature of the initial prototypes, meant that the system building tools had to be as flexible and efficient as possible for effective software design and development. The size of the databases, the expected transaction volumes, and the long-term value of the court records required a data manipulation system capable of providing high performance and integrity. The resulting design criteria, the programming capabilities developed, and their use in system construction are described herein. This database programming facility has been especially designed as a technical management tool for the database administrator, while providing the applications programmer with a flexible database software interface for high productivity.

The database programming facility described here was developed at the Federal Judicial Center (FJC) in Washington, D.C. The FJC serves as the research and development agency of the federal judiciary.

The federal judicial process is based upon the interaction of various organizationally autonomous, though cooperating, agencies and organizations. There are 94 district (trial) courts and 12 appellate courts, plus the Supreme Court, each with its own caseload to administer and adjudicate. The public at large also participates in the court process as jurors, litigants, defendants, and witnesses. Each of these organizations and individuals has a need for timely and correct information. The Office of the Clerk within each court is the hub and nerve center for this information exchange network and is the natural location for an integrated database. The information systems under development at the FJC are intended to streamline clerical processing, give the judges and other managers greater management control, and provide data for research in judicial administration.

Within each of the various application domains, the databases of each court are disjoint and are physically maintained as separate database areas, although the database structures are defined by a common schema. Currently, the various databases are maintained on several DECsystem-10 computers located in Washington, D.C. Court access is provided through terminals and printers located in the courts and connected to the data centers via a value-added telecommunications network.

The various information systems are designed to be "model directed" [3, 4], in that mathematical models of court procedure are embedded in the systems so as to direct all system response to user input, and monitor the processing of cases relative to correct procedures and regulatory statutes, such as the Speedy Trial Act.

The major software components necessary to implement these information systems are the host language processor, the database management system (DBMS), and the host language/DBMS interface. Our objectives were to develop a comprehensive data manipulation facility that would be flexible and efficient for information system construction, and in which the boundaries between system components would be "natural" and even transparent to the system developer.

In particular, these components have been constructed or augmented to efficiently utilize a network database and solve the problem of multiple record currency by allowing extended "in-core" database representations, whose structure can be determined at execution time by the application program independently of the main database.

The host language selected was SAIL [19]. A CODASYL network-type database management system, DBMS-10 [10, 20], was also chosen. The interface between these two components was specially designed to enhance the capabilities of both components.

In Section 2 the database for a particular application area, that involving the management of criminal cases, is described. In Section 3, desirable implementation language features and programming management tools are discussed. The DBMS is characterized in Section 4, along with a description of the host language/DBMS interface.

2. THE CRIMINAL DOCKETING DATABASE

The maintenance of the official court docket for each case is a major responsibility of the Clerk's office. Since the docket is the most fundamental reference document used to determine case status and progress, its contents provide data that are pertinent for case tracking and reporting. The line items in the docket are called "events" and correspond one-to-one to legally significant court transactions (e.g., defendant arraigned, motion granted, etc.). A database containing a court's dockets provides an extremely useful source of information for case flow management.

The data inputs for entry onto the docket sheet are obtained from a variety of sources, including open court proceedings as recorded by a judge's Courtroom Deputy, directly from a judge's chambers, or from the Clerk's office itself through filings made by attorneys, court-related agencies such as the U.S. Marshal's Office, or the public. The associated documents are then given to a Docket Clerk in the Clerk's office, who enters the appropriate summary information on the docket sheet. This activity is known as *docketing*. With an automated system, *interactive docketing*, or *event posting*, may be done using a computer terminal.

2.1 Database Contents and Structure

The information contained in the criminal docketing database includes, for each case:

(1) parties to the case,
(2) offenses charged and to be adjudicated,
(3) docketed events, in chronological sequence for each case,
(4) judges assigned to the case,
(5) attorneys associated with the case,

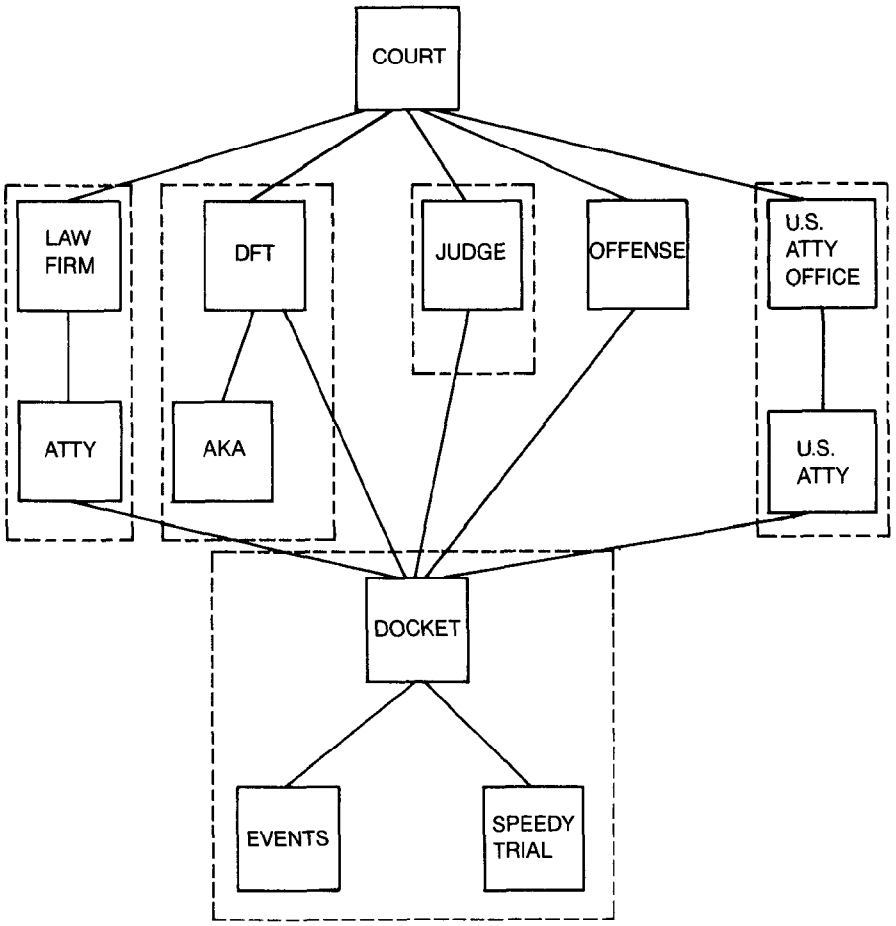

Fig. 1. Simplified criminal database.

(6) time interval and schedule constraints, and
(7) related cases.

The relationships that exist among these entities must be represented in a form that allows for the efficient extraction of management information. The information contained in each court's database is to be used primarily for the management and information needs of the court community dealing with that particular court. The information will also prove useful in preparing various required national statistical reports on case management and the administration of justice, as well as being useful as a resource for doing research into better court management methods.

A simplified database diagram (using the data structure diagram technique of [1], where rectangles denote record types and arrows denote set relationships) is shown in Figure 1. When a case is opened, a DOCKET record is established for each defendant (DFT). Defense counsel (ATTY), prosecutors (U.S. ATTY), a JUDGE, and alleged offenses (OFFENSE) are linked to the DOCKET, if they are known at that time. A variety of different EVENT types may then be posted

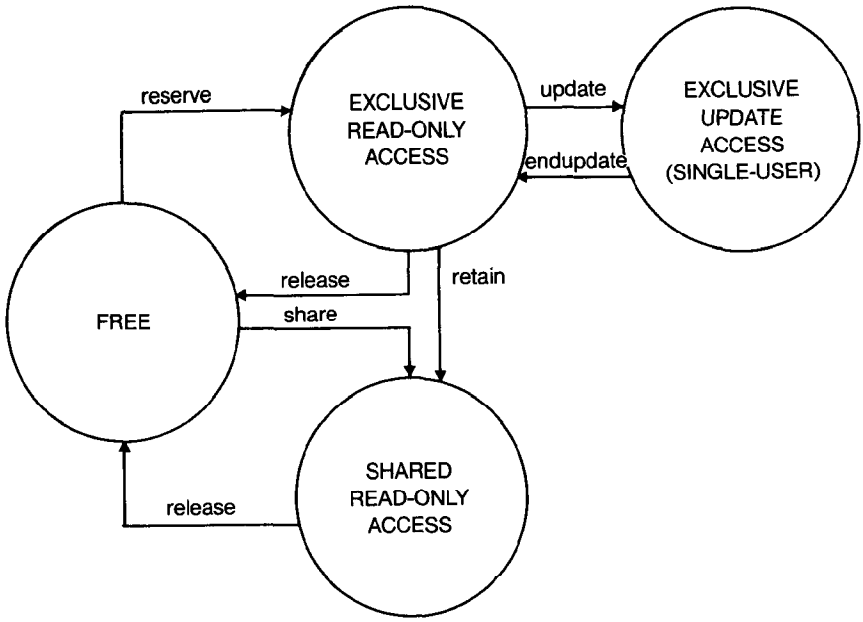

Fig. 2. Concurrent database update control algorithm.

as the case proceeds. Case monitoring relative to the Speedy Trial Act is carried out and recorded (SPEEDY TRIAL). The actual database contains some 40 different record types related within about 50 different set types.

Database consistency is maintained over concurrent updates from multiple users by defining subsets of the database to be *resource classes* and locking on these logical entities [12, 15]. The resource classes are defined by the regions of the database enclosed within dotted lines, as shown in Figure 1. For each database transaction, including docketing, a corresponding set of resource classes is defined for locking during the update portion of that transaction.

The algorithm for controlling concurrent database updates using a state diagram is shown in Figure 2. Each resource class is controlled independently. A resource in the FREE state means that no user has a claim upon that resource for reading or writing. The SHARED state for a resource may be shared by multiple read-only users. The EXCLUSIVE state provides exclusive reading and writing privileges for a resource and is a preparatory state for writing to the database. The UPDATE state provides exclusive reading and writing privileges for a resource. The DECsystem-10 operating system allows only one job at a time to have a file open for writing. Therefore, only one user can be in the UPDATE state at a time across all resources for a given database.

This control algorithm is enforced as a programming convention under which the user's program will evoke the commands given on the arcs of the diagram in Figure 2 as procedure calls with the resource class name (ATTY, DFT, JUDGE, U.S. ATTY, or DOCKET) as an argument. The DOCKET resource class is further identified by docket number so as to permit concurrent event posting to separate dockets.

In the next two sections, the database software facilities developed for project management and programming are described.

3. SYSTEM IMPLEMENTATION LANGUAGES

The design and implementation of large software systems is very much influenced by the programming environment available for supporting the development of such systems. Choice of a suitable programming language in which to implement software systems is critical in ensuring the timely success of the original implementation, and it is even more important in ensuring the long-term maintenance and enhancement of the software throughout its life cycle. Of course, an appropriate programming language is not the only tool necessary in providing a productive programming environment. Text editors, linkage editors, automated validation and verification programs, and other software tools all contribute to providing a suitable environment for systems building, not to mention the influence of a friendly operating system, file organization, and so on.

3.1 Language Features

While all the above noted environmental factors affect the flexibility and ease with which software systems may be designed and constructed, and many of the widely discussed attributes of modern programming languages [13] are relevant for our consideration, the following language features are especially helpful in providing a language environment conducive to the design and implementation of large-scale database management systems (see also [17]):

(1) Sufficient data types, data structures, and associated operators to permit natural manipulation of database elements and records. Such data types might include integer, real, Boolean, string, array, record, pointer, set, and list.

(2) Boolean expression evaluation, in conjunction with appropriate program control constructs (e.g., conditional statements, repetitive control statements, and compound statements).

(3) Procedures and functions, with call-by-value and call-by-reference parameters.

(4) Dynamic storage allocation, for use in the creation of dynamic data structures (such as an in-core database representation, as described in Section 4.1) and for use in stack-oriented operations such as recursive procedures calls and lexical nesting of the scopes of variable definitions (block structure).

(5) Macro definition facilities, including parameter passing.

(6) Conditional compilation and related compile-time control constructs and operations (e.g., compile-time expressions and type-checking predicates, and compile-time conditional statements).

(7) Program source libraries, elements of which can be copied into an application program via a compile-time directive.

(8) Independent compilation capability, with appropriate environment declaration and module interfacing facilities to allow the subsequent linking of independently compiled program modules into an integrated software system. Independent compilation is especially useful in the construction of large software systems, particularly if some parts of the system must be written in different programming languages.

The SAIL programming language [19] provides all of these software development facilities, and hence was deemed an appropriate implementation language in which to program the criminal docketing database system for the federal courts.

3.2 Example

As an example of the use of SAIL as a host language in a database management system, consider the following program fragment. The task is to traverse a set named ASSIGNED-CASES owned by a JUDGE record, extract an integer data item called DOCKET-NUMBER from each DOCKET record which is a member of the set, and save the list of DOCKET-NUMBERs for subsequent processing. The exact instance of the set occurrence is identified by the owner record, JUDGE, having the value "JONES" for the data item JUDGE-NAME. Since SAIL has a data structuring facility (known as a RECORD_CLASS, similar to a PL/I structure [2]), we define a data structure called LISTX and a procedure to add items to the front of the list. The data structure LISTX has two fields: ELEMENT, which is of type INTEGER, and NEXT, which is of type RECORD_POINTER. NEXT points to another instance of the LISTX data structure. The procedure ADDTOLIST has two arguments: a pointer to the head of an instance of LISTX and the integer to be added to this instance. Storage for a record structure may be dynamically allocated using the primitive NEW_RECORD(record-type), which returns a pointer to the newly created storage. Fields of a record are referenced as "record-type:field-name[record-pointer]."

```
        .
        .
        .
record_class LISTX (integer ELEMENT;
                    record_pointer (LISTX) NEXT);
procedure ADDTOLIST (reference record_pointer (LISTX) HEAD;
                     integer VAL);
  begin
    record_pointer (LISTX) TEMP;

    TEMP := new_record (LISTX);
    LISTX : ELEMENT[TEMP] := VAL;
    LISTX : NEXT[TEMP] := HEAD;
    HEAD := TEMP;
  end;
```

The COBOL/DML and SAIL encodings for the set traversal task are given in Figure 3. The critical difference is the step "Add DOCKET-NUMBER in DOCKET to result list". It is not immediately obvious how the concept of a list would be implemented in COBOL; since COBOL does not provide dynamic storage allocation, one would have to decide how much storage should be allocated statically for the list structure, and then be concerned with checking for overflow conditions as elements are appended to this fixed-size list. Such problems do not arise when using SAIL, owing to its data structuring facility and dynamic storage allocation.

Note that a code sequence similar to that of Figure 3 could be used to build an in-core representation of the ASSIGNED-CASES set of the database by retriev-

COBOL Program:

```
  move 'JONES' to JUDGE-NAME in JUDGE.
  find JUDGE record.
  if ASSIGNED-CASES set EMPTY go to NONE-SUPPLIED.
NEXT.
  find next DOCKET record of ASSIGNED-CASES set.
  if ERROR-STATUS = 0307 go to ALL-FOUND.
  get DOCKET.
  "Add DOCKET-NUMBER in DOCKET to result list".
  go to NEXT.
ALL-FOUND.
```

SAIL Program:

```
JUDGE_NAME := "JONES";
find_calc (JUDGE);
if EMPTY_SET (ASSIGNED_CASES) then go to NONE_SUPPLIED;
while true do
  begin
    find_next (DOCKET, ASSIGNED_CASES);
    if ERROR-STATUS = 0307 then DONE;
    get (DOCKET);
    addtolist (HEAD, DOCKET_NUMBER);
  end;
```

Fig. 3. Sample DBMS-10 application.

ing and linking entire DOCKET records. This would be useful in situations requiring repeated access to various records of the set, as might occur during report generation, without having to reestablish database currency and making repeated accesses to secondary storage.

4. THE DATA MANIPULATION SYSTEM

There are currently three different approaches to the design of database management systems [9, 14]. These are the network approach as described in the CODASYL report [7, 25]; the hierarchical approach of the IBM IMS system [26]; and the relational approach [8]. Despite their differences, each attempts to separate the process of describing the data organization from the process of manipulating the data. It is this separation that differentiates these systems from conventional approaches to the design of application software. The actual data manipulation process is carried out using a set of commands which create, access, and modify entries in the database.

We have selected the network approach for our data manipulation environment. This rather general approach was designed by the CODASYL Committee to be used in an environment supported by the COBOL [6] programming language, as evidenced by the similarity between the CODASYL data description facility (DDL) and the data division specifications of COBOL. Some implementations of the CODASYL report have used FORTRAN as a data manipulation language [18, 24]. This has required the use of a functional notation, as distinguished from the COBOL verb notation, to describe the basic data manipulation language (DML) operations.

The CODASYL report presents several languages to deal with various aspects of data management, including a Schema Data Description Language (Schema DDL), a Subschema Data Description Language (Subschema DDL), and a Data Manipulation Language (DML). We have selected SAIL as the host language in which to embed the DML, for the reasons outlined in Section 3.

This section describes the software interface that allows the SAIL programming language to be extended to utilize and enhance the DBMS-10 network-type database management system. SAIL is sufficiently rich in data types that in this implementation the DDL is unaltered and no modification to the SAIL compiler is necessary. We first describe the use of an in-core database consisting of SAIL record structures. Next, we present the methods by which the application programmer may utilize the DML verbs in an extended data manipulation facility. This was implemented using SAIL's extensive compile-time system [21, 22].

4.1 Overview and Enhancements

DBMS-10 provides a means for the user to arrange a database on secondary storage only. Data is defined in terms of aggregates, called records, whose subcomponents, of possibly varying type, are known as data items. All of the data manipulation commands are oriented toward fetching, modifying, and storing data in this database. DMBS-10 programs communicate with the database through the use of a block of storage known as the User Work Area (UWA). In this block, storage is allocated for a single instance of each data item of each record type.

Allowing the user to build an in-core representation of his database is a desirable capability in a database management system. This is best seen by observing that, in a typical database implementation, when a user obtains one instance of a record type from the database (i.e., he locates it via a FIND and fetches it via a GET), he has no convenient way of keeping it in temporary memory while obtaining another instance of this record type. Of course, he can allocate temporary storage for the various fields; however, this becomes cumbersome when it is desirable to keep track of more than two instances of a record type. Alternatively, instances of certain record types can be refetched from the database. In fact, this strategy is the one generally followed by programmers in order to ensure correctness of operation and documentation clarity. However, this process may be rather expensive in that currency may have to be reestablished. Such a process involves backtracking, and unless the programmer is very careful, he may not obtain the records he thinks he is obtaining. For an example of the pitfalls of such a technique see [25]. Fortunately, SAIL, unlike COBOL or FORTRAN, has a data structuring facility which lends itself to this application. As mentioned earlier, this facility consists of record classes and a dynamic storage allocation capability; together, they enable the formation of lists and sets at run-time.

The implementation described here facilitates the use of lists and sets by defining a SAIL record type for each DBMS-10 record type. A preprocessor generates a SAIL RECORD_CLASS declaration for each DBMS-10 record type defined in the DDL with subfield names identical to the DMBS-10 data item names. For example, see Figure 4, where a DBMS-10 record definition is shown

```
DBMS-10 Record:

RECORD NAME IS JUDGE.
02  CIRCUIT        TYPE FIXED BIN REAL.
02  DISTRICT       TYPE FIXED BIN REAL.
02  JUDGE-NAME     PIC X(40) USAGE IS DISPLAY-7.
02  YEARS          TYPE FLOAT BIN REAL.

SAIL Record:

record_class JUDGE (integer !TYPE;
                    integer DBKEY;
                    integer CIRCUIT, DISTRICT;
                    string JUDGE_NAME;
                    real YEARS);
```

Fig. 4. Sample record definition.

along with the corresponding SAIL record. In addition, primitives are provided for the user to create instances of these record classes and to load them from the UWA in one statement. A similar provision has been made for storing an instance of a SAIL record in the UWA with one statement. These statements are actually calls to procedures which have been defined at the same time that storage for the UWA was allocated.

As mentioned earlier, one of the principal advantages of the data structuring capability is the avoidance of the need to reestablish currency. The actual cost of reestablishing currency is dependent on several factors. These include the size of the I/O buffers, the database accessing activity since the establishment of the desired record as the current of RUN-UNIT, and the page replacement strategy employed by the database control system (DBCS). For example, if the desired record instance is still in the I/O buffers, then the cost of the actual access is relatively low in comparison with the cost of searching the database on secondary storage. The search may be direct (i.e., involve only one disk operation), or it may be indirect, thereby causing several disk accesses by virtue of a need to chase pointers. Contrast the uncertainty associated with this technique with the notion of being able to refer to instances of the desired record in an in-core database. The ability to create an in-core database representation provides a logical capability for the user to temporarily store various instances of his data records without sacrificing the organizational clarity imposed by the data definition language (DDL).

One of the primary reasons for the introduction of database management systems is a desire to separate the act of defining the structure of the database from the process of accessing and manipulating it. However, we have just seen that this is only true in theory. In practice, efficiency considerations play an important criteria in evaluating programs. This means that the applications programmer can substantially improve the performance of his program given a knowledge of the internal structure of the database control system. For example, knowledge of the buffer sizes and that a least-recently-used page replacement scheme is employed by the database control system can be used to advantage by the programmer. Costly disk accesses can be avoided by scheduling references to

certain instances of records in such a way as to ensure that they are still in the buffers. However, substantial programming cost might be expended to achieve this improvement in application performance, and the resultant application code will be dependent upon the implementation details of the database control system. In contrast, the ability to create an in-core database fragment preserves a greater degree of separation between the data description and data manipulation components without sacrificing efficiency.

In order to facilitate the efficient use of an in-core database, with each record in the database there exists a unique value, called a *database key*, which serves to locate the record. Knowledge of the database key can greatly speed up the accessing and modification of records in secondary storage, since the record can thereafter be located with one disk operation. This is in contrast with the possibility of several disk accesses by virtue of a need to traverse linked lists in the process of locating a record. We provide this capability by adding to each SAIL record type a field known as DBKEY which contains the database key of the DBMS-10 record.

In order to be able to detect record types at run-time (SAIL records are essentially compile-time structures with respect to type-checking), we also add to each SAIL record type, corresponding to a DBMS-10 record type, a field, called !TYPE, which is the integer representation of the first five characters comprising the name of the record type. This feature is quite useful in defining in-core database sets whose members are records of varying types.

The inclusion of a database key field in the SAIL record class for each DBMS-10 record type can be combined to yield some efficiency in transactions that involve the updating of entries in the database. When a user wishes to modify an existing instance of a DBMS-10 record which occurs in his own in-core database, the modification operation is interpreted by the system to do the following:

(1) Update the UWA to reflect the contents of the SAIL record.
(2) Use the database key associated with the SAIL record to locate the corresponding instance of the DBMS-10 record in the DBMS database. This is done via a FIND_DIRECT command which suppresses all currency updates except for the RUN-UNIT.
(3) Perform the database MODIFY operation.

Efficiency is increased as a result of step (2). The database key enables the user to locate the instance of the record in the DBMS-10 database without having to reestablish currency. Observe that we are trading processing on extended storage for processing in primary memory. Another advantage of such a mechanism is the possible freeing of the programmer from needing to know how the database is organized. Even more important is the following extension. Suppose queries and updates to the database are to be made by relatively unsophisticated users. In such a case, these users could converse with an intermediate processor which translates their queries and updates into a sequence of calls on the database, where this command processor could make use of the in-core database capability.

Most often, a database is organized with certain applications in mind. The goal is to render the most frequent operations as efficient as possible. This goal is

usually achieved by providing extra database access paths between various data entities. The access paths are particularly evident in the definition of sets and the relationships between members of the set, and between the members and the owner of the set. The access relationships are defined using constructs such as LINKED TO NEXT, LINKED TO PRIOR, and LINKED TO OWNER. With the use of run-time structures, such as an in-core representation of the database, the need for an a priori organization of the database with static links and their associated overhead can be reduced. Subject to limitations of primary memory in which to construct in-core database representations, the database designer no longer needs to provide set linkages merely to render infrequent operations efficient.

4.2 The Data Manipulation Language

A SAIL program accesses the database primarily via the use of a set of DML verbs. The verbs are implemented using the SAIL macro facility in which they look like function calls. Use of verbs is facilitated by a compile-time symbol table, which enables many of the verbs to be generic, unlike some COBOL and FORTRAN implementations. Since the SAIL/DBMS interface provides generic DML verbs, the user need not qualify record names, set names, and area names by their respective qualifiers RECORD, SET, and AREA. Similarly, for several of the built-in primitives, there is an interchangeability in the parameters allowed between record names and record pointers, strings, character sequences, and STRING variables, thereby facilitating the implementation of libraries of general purpose database manipulation procedures.

Primitives to communicate between the UWA and in-core database. As mentioned earlier, for each record type defined in the database, we have a corresponding SAIL record-class declaration. In order to transfer data between the in-core database and the UWA, a set of three procedures is automatically generated for each record type, denoted by ⟨record-name⟩. For example, for the JUDGE record type, the procedures NEW_JUDGE, FETCH_JUDGE, and STORE_JUDGE will be automatically defined. The function of each of these procedures is as follows:

(a) NEW_⟨record-name⟩
  (1) Allocate a SAIL record of type ⟨record-name⟩.
  (2) Load the contents of the UWA corresponding to ⟨record-name⟩ into the new SAIL record.
  (3) Store the database key associated with the current instance of ⟨record-name⟩ in the DBKEY field of the SAIL record.
  (4) Store ⟨record-name⟩ in the !TYPE field of the SAIL record.
  (5) Return the record-pointer to the newly allocated SAIL record.

(b) FETCH_⟨record-name⟩ (⟨record-pointer⟩)
  Same an NEW_⟨record-name⟩, except that an existing SAIL record (i.e., the one pointed to by ⟨record-pointer⟩) is reused.

(c) STORE_⟨record-name⟩ (⟨record-pointer⟩)
  Load the UWA locations corresponding to ⟨record-name⟩ from the SAIL record pointed to by ⟨record-pointer⟩.

One of the important capabilities of the system is the use of a compile-time symbol table to enable a flexible set of DML verbs to function in a limited generic way. In order for the DML verbs (e.g., GET, STORE, MODIFY, INSERT, REMOVE, and DELETE) to be able to properly handle record pointers (in addition to the usual record names), we provide a mechanism for declaring them. For example, the following results in ⟨record-pointer⟩ being declared to be a pointer to a record of type ⟨record-name⟩:

DBMS_RECORD_POINTER (⟨record-name⟩, ⟨record-pointer⟩)

Area management. The concept of an area, which is analogous to a file, has also been enhanced. In a given database, a record type may appear in more than one area. In order to enable processing of a number of areas containing the same record type, there exists a mechanism to indicate the area in which STORE operations are to deposit the newly created instance of the record type. This is accomplished by the use of a variable, called an ⟨area-id⟩, specified with each record type definition. The contents of this ⟨area-id⟩ variable identify the name of the area to be affected for all STORE operations involving this record type. The following primitive is provided to enable assignments to this variable:

SET_AREA_FOR_STORE (⟨area-id⟩, ⟨area-name⟩)

The ⟨area-id⟩ is an ⟨area-id⟩ variable, the name of a record type, or a STRING variable whose value is a record type. In the latter two cases, the ⟨area-id⟩ variable associated with the record type is loaded with the appropriate area name. This is useful because the programmer does not have to remember the ⟨area-id⟩ associated with a specific record type. The ⟨area-name⟩ is the name of an area, a string, or a STRING variable containing a DDL area name.

In the definition of the DML verb STORE, we shall see the use of an optional third argument, which is the name of an area. In this case, the ⟨area-id⟩ associated with the target record name is set to the specified area name, but only for the duration of the STORE operation. Once the STORE is completed, ⟨area-id⟩ reverts to its previous value. This is especially attractive when the same variable is declared to denote the area in which a record is to be stored for different record types.

DML verbs. In the calling sequences for DML verbs, SAIL record pointers and string variables may be used instead of record names. When a SAIL record pointer is used, the necessary action depends on the verb. For example, in the case of a STORE operation, the record referenced by the record pointer is loaded into the UWA and then stored in the DBMS-10 database. The use of a string variable whose value is the name of the record (or the name of a set or area, as appropriate) is practical because it enables the programmer to avoid the need for separate DML verb calls for different record types (as well as sets and areas). Such a capability is extremely useful in building libraries of general purpose routines.

In order to demonstrate the adaptation of CODASYL DML to SAIL, we illustrate how some of the DML verbs are used in SAIL. Our exposition stresses the enhancements over the standard definition. A number of examples are also given to contrast the use of SAIL and COBOL. In these examples, a command is given in COBOL, followed by the SAIL encoding on the command.

The FIND verb is a complex command, having several variations, which results in the evaluation of its arguments to yield an instance of a record which becomes the current of RUN-UNIT. All variations of the FIND command may specify a suppress option, which indicates what, if any, currency updates are to be suppressed. In the following, the symbol ⟨aggregate⟩ denotes a ⟨set-name⟩ or an ⟨area-name⟩. The symbol ⟨item⟩ denotes a record type, or the choice of ANY or NULL, both meaning an arbitrary record type. The symbol ⟨suppress⟩ denotes one of the values ALL, RECORD, AREA, SET, a list of set names, or NULL. As an example, consider the FIND_NEXT verb.

FIND_NEXT (⟨aggregate⟩, ⟨item⟩, ⟨suppress⟩)

This command locates the occurrence of record ⟨item⟩ with the next higher database key relative to the current record of ⟨area-name⟩, or the next record relative to the current record of ⟨set-name⟩ in the logical order of the set. For example,

```
COBOL:  FIND NEXT JUDGE RECORD OF CHICAGO AREA.
SAIL:   FIND_NEXT (CHICAGO, JUDGE);
```

The GET verb results in the transfer of the contents of the data items of the object record into the UWA. It is generally used to fetch a record for processing, once the record has been located via a FIND command which established it as the current of RUN-UNIT. The symbol ⟨data-names⟩ denotes a list of data item names, or it may be NULL, indicating all fields of the current record are to be retrieved.

GET (⟨record-name⟩, ⟨data-names⟩)

The above command performs a GET of the current of RUN-UNIT, transferring the contents of the selected data items of the object record into the user work area.

record-pointer := GET_NEW (⟨record-name⟩)

This command performs a GET (⟨record-name⟩), followed by the allocation of a new SAIL record to hold the newly fetched record. The SAIL record is loaded from the user work area, and a pointer to the newly allocated SAIL record is returned as the value of this primitive. The actual implementation makes use of the procedure NEW_⟨record-name⟩.

GET (⟨record-pointer⟩, ⟨data-names⟩)

This form of the GET command is the same as GET_NEW (⟨record-name⟩), except that the SAIL record referenced by ⟨record-pointer⟩ (rather than a newly allocated record) is loaded from the user work area (using the FETCH_⟨record-name⟩ procedure). If ⟨data-names⟩ is NULL, then the entire record is loaded. Otherwise, only the individual fields specifically requested are loaded. One note of caution is that if ⟨data-names⟩ is not NULL, then the DBKEY field of the SAIL record pointed to by ⟨record-pointer⟩ will not be updated. For example,

```
COBOL:  GET JUDGE.
SAIL:   GET (JUDGE);
```

The STORE verb results in the acquisition of space and a database key for a new occurrence of a record in the database. The new record occurrence contains

the values of the appropriate data items in the UWA. The new object record is inserted into all sets for which it is defined to be an AUTOMATIC member. Also, new set occurrences are established for each set for which the object record is defined to be an owner.

STORE (⟨record-name⟩, ⟨suppress⟩, ⟨area-name⟩)

As for the FIND verb, the suppress option indicates what, if any, currency updates are to be suppressed. As noted previously, ⟨area-id⟩ is a location that is declared to contain the area information for ⟨record-name⟩. If ⟨area-name⟩ is specified in a call to the STORE verb, then the current value of ⟨area-id⟩ is saved, SET_AREA_FOR_STORE (⟨area-id⟩, ⟨area-name⟩) is performed, and ⟨area-id⟩ is restored to its previous value upon completion of the STORE operation. The ⟨area-name⟩ is either an area name, or a string or STRING variable containing an area name. For example,

STORE (JUDGE, ALL, CHICAGO);

will have the effect of storing the current record of type JUDGE in area CHICAGO while suppressing all currency updates.

STORE (⟨record-pointer⟩, ⟨suppress⟩, ⟨area-name⟩)

In this form of the STORE verb, the data associated with the SAIL record referenced by ⟨record-pointer⟩ is moved into the UWA and the appropriate STORE operation is performed. The actual implementation makes use of the procedure STORE_⟨record-name⟩.

DELETE (⟨record-name⟩, ⟨option⟩)

The DELETE verb results in the object record being made unavailable for further processing by DML primitives. In addition to naming the record being deleted, the user may specify the manner of disposition of sets in which the indicated record participates as an owner. The symbol ⟨option⟩ denotes one of the values ALL, SELECTIVE, ONLY, or NULL.

DELETE (⟨record-pointer⟩, ⟨option⟩)

In this form of the DELETE verb, the database key associated with the DBKEY field of the record referenced by ⟨record-pointer⟩ is made the current of RUN-UNIT by use of a FIND_DIRECT command which suppresses all currency updates. Once this is done, the record is deleted.

In addition to providing a complete set of DML verbs, predicates are also provided for determining owner-member relationships between the current of RUN-UNIT and various sets, in addition to being able to identify empty sets. For example,

MEMBER_OF_SET (⟨set-name⟩)

determines whether the current of RUN-UNIT is a member of ⟨set-name⟩, where ⟨set-name⟩ may be ANY or NULL or a set name.

Database programming administration. The actual organization of the SAIL/DBMS database programming system is designed to support a hierarchy of users consisting of the database administrator (DBA) and the application programmers.

The DBA is charged with the logical and physical database design, maintaining the integrity of the database and, in our case, providing a library of routines to access the database in such a way as to take advantage of the in-core database capability described earlier. Application programmers make use of these routines, as well as DML verbs, to write application programs.

In order to accommodate both groups of users, the data definition file (DDL) is processed as previously described by a program which automatically generates a library of procedures for accessing the in-core database, the compile-time symbol table, the SAIL record definitions, and storage allocation requests for the UWA. The DDL is centrally controlled by the DBA, who can thereby ensure that certain data access programming conventions are followed, rather than depending on the application programmers to abide by centrally established guidelines. The DBA uses these automatically generated primitive procedures to build higher-level procedures geared to a particular application for use by the application programmers. Thus, centralized management of critical aspects of data manipulation programming is maintained, while allowing the programmers flexibility in the use of procedures in their applications. The compile-time symbol table, the SAIL record definitions, and the UWA declarations are used by both classes of users.

Another component of the system is a file containing macro definitions for the DML verbs. This file is used by both groups of users in compiling their programs. There are several advantages to such a scheme. First, if a not too radically different database management system were to be employed, then user programs need not be recoded. Only the DML verb definitions would have to be changed. Second, extensions can be made to the database management system in a manner relatively independent of the compiler of the high-level language. This depends to a large extent on the availability of a powerful macro processing facility, such as that provided by SAIL [22].

5. CONCLUSION

The database management system described in this paper has been successfully utilized in the development of the criminal docketing system, which became operational in late 1976. Currently, 15 of the largest federal District Courts (out of a total of 94) utilize the system. These are the courts in Washington, D.C., Boston, Manhattan, Brooklyn, Atlanta, Detroit, Chicago, Kansas City, San Antonio, Houston, Portland, San Francisco, Los Angeles, San Diego, and Phoenix. The caseloads of these courts constitute about 40 percent of all federal defendant felony filings. Similar database systems are being constructed for civil, appellate, and bankruptcy cases and other court applications.

The database programming system described herein has provided effective technical management tools for the database administrator and application project managers. The programmers have shown high productivity in its use, and the database programming interface has provided the necessary flexibility to be responsive to change in this dynamic new application area.

ACKNOWLEDGMENTS

We would like to acknowledge the valuable discussions we had with John Hulme, Norbert Kubilus, and John Reiser during the design of the system.

REFERENCES

1. BACHMAN, C.W. Data structure diagrams. *Database 1*, 2 (Summer 1969).
2. BEECH, D. A structured view of PL/I. *ACM Comput. Surv. 2*, 1 (March 1970), 33–64.
3. BUCHANAN, J.R., AND FENNELL, R.D. Model directed information systems for management of the federal courts. *Manage. Sci. 27*, 8 (Aug. 1981).
4. BUCHANAN, J.R., AND FENNELL, R.D. An intelligent information system for criminal case management in the federal courts. In *Proc. 5th International Joint Conference on Artificial Intelligence* (Aug. 1977).
5. BUCHANAN, J.R. Management information systems for the federal courts. In *Proc. of the Conference on Interactive Information and Decision Support Systems*, Office of Naval Research, Wharton School of Management, Nov. 1975.
6. COBOL. American National Standard programming language COBOL. X3.23–1974, American National Standards Institute, Inc., New York, 1974.
7. CODASYL DATABASE TASK GROUP. *April 1971 Report.* ACM, New York, 1971.
8. CODD, E.F. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377–387.
9. DATE, C.J. *An Introduction to Database Systems.* Addison-Wesley, Reading, Mass., 1975.
10. *DECsystem-10 Database Management System Programmer's Procedures Manual.* Document DEC-10-APPMA-B-D, Digital Equipment Corp., Maynard, Mass., 1977.
11. EBERSOLE, J.L., AND HALL, J.A. COURTRAN, an information system for the courts. *J. Comput. Law*, Rutgers Univ., 1972.
12. ESWARAN, K.P., GRAY, J.N., LORIE, R.A., AND TRAIGER, L.I. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
13. FENNELL, R.D. Choosing a programming language for the implementation of large software systems. Federal Judicial Center Internal Report, April 1979.
14. FRY, J.P., AND SIBLEY, E.H. Evolution of database management systems. *ACM Comput. Surv. 8*, 1 (March 1976), 7–42.
15. HULME, J. Notes on the design of the criminal database. Federal Judicial Center Internal Report, Sept. 1975.
16. NIHAN, C.W. COURTRAN II, an assessment of applications and computer requirements. Federal Judicial Center Report, Sept. 1974.
17. PARSONS, F.G., DALE, A.G., AND YURKANAN, C.V. Data manipulation language requirements for database management systems. *Comput. J.* (May 1974).
18. RAPIDATA CORP. A FORTRAN DML implementation for DBMS-10. Fairfield, N.J.
19. *SAIL User Manual.* J.F. Reiser, Ed., Stanford Artificial Intelligence Laboratory Memo AIM-289, Computer Science Dept., Stanford Univ., Stanford, Calif., Aug. 1976.
20. SALMOND, K., AND HULME, J. A study of database management system. Federal Judicial Center Internal Report, Nov. 1975.
21. SAMET, H. A coroutine approach to parsing. *ACM Trans. Program. Lang. Syst. 2*, 3 (July 1980), 290–306.
22. SAMET, H. Design of a macro system for high-level languages: a retrospective and prospective view. Computer Science TR-1353, Univ. of Maryland, College Park, Dec. 1983.
23. Speedy Trial Act, Public Law 93-619, 18 U.S.C. 3161, Jan. 1975.
24. STACEY, G.M. A FORTRAN interface to the CODASYL Database Task Group Specifications. *Comput. J.* (May 1974), 124–129.
25. TAYLOR, R.W., AND FRANK, R.L. CODASYL Database Task Group specifications. *ACM Comput. Surv. 8*, 1 (March 1976), 67–103.
26. TSICHRITZIS, D.C., AND LOCHOVSKY, F.H. Hierarchical database management: a survey. *ACM Comput. Surv. 8*, 1 (March 1976), 105–123.

Received 1978; revised December 1982; accepted May 1983