CODASYL Data-Base Management Systems

ROBERT W. TAYLOR

IBM Research Laboratory, San Jose, California 95193
and
RANDALL L. FRANK

Computer Science Department, University of Utah, Salt Lake City, Utah 84112

This paper presents in tutorial fashion the concepts, notation, and data-base languages that were defined by the CODASYL Data Description Language and Programming Language Committees. Data structure diagram notation is explained, and sample data-base definition is developed along with several sample programs. Advanced features of the languages are discussed, together with examples of their use. An extensive bibliography is included.

Keywords and Phrases: data base, data-base management, data-base definition, data description language, data independence, data structure diagram, data model, DBTG Report, data-base machines, information structure design

CR Categories: 3.51, 4.33, 4.34

NTRODUCTION

The data base management system (DBMS) specifications, as published in the 1971 Report of the CODASYL Data Base Task Group (DBTG) [S1], are a landmark in the development of data base technology. These specifications have been the subject of much debate, both pro and con, and have served as the basis for several commercially available systems, as Fry and Sibley noted in their paper in this issue of COMPUTING SURVEYS, page 7. Future national and international standards will certainly be influenced by this report.

This article presents in tutorial fashion the concepts, notation, and data-base languages that are defined by the "DBTG Report." We choose the term DBTG to describe these concepts, even though since 1971 the role of the original Task Group has been assumed by the CODASYL Data Description Language Committee and the Data Base Language Task Group of the CODASYL Programming Language Committee. In fact, this article uses the syntax of the more recent reports [S2, S3, S4]. However, because the fundamental system architecture remains essentially the same as that specified in the 1971 Report, we still use the abbreviation DBTG; though not strictly accurate, "DBTG" does reflect popular usage.

This article explains the specifications; it will not attempt to debate the merits and/or demerits of the specific approaches taken to implement them or of the features of the different approaches. Such debates have taken place and they will continue to be held. Michaels, Mittman and Carlson survey in their paper [see page 125] many of the points that have been debated. It should be remembered, however, that the initial role of the DBTG was to recommend *language and system specifications* for data-

CONTENTS

base processing in the COBOL programming language. Data-base applications written in a host programming language are often associated with both large data bases which contain 10^8 characters or more and a well known set of applications or transactions, perhaps run hundreds of times a day, triggered from individual terminals. Such extensive processing must be efficient, and the designers of the DBTG system took care that its applications could be tuned to ensure efficiency. Although the DBTG recognized the importance of supporting other language interfaces for a data base, especially "self-contained" languages for unanticipated queries, it did not directly address the problem of other interfaces.

This paper presents concepts and language statements that are characteristic of DBTG-like systems. Where possible, we point out how a particular feature or option might be used.

1. DESIGN OF A DATA BASE

Two of the most difficult areas of data-base management are the design of an information structure and the reduction of that structure to a data structure which is compatible with and managed by the DBMS. This section deals with both of these topics, though emphasis is placed on the data structure design decisions which must be made. Later on, we introduce and provide examples of *data structure diagrams*, a notation that is widely used to deal with data and information.

Concepts of Information Structure Design

Data-base management systems are tools to be applied by the users of these systems to build an accurate and useful model of their organization and its information needs. To accomplish this, the information structure must accurately define and characterize the items of data and the relations among them that are of interest to the users. This is no small task, for it demands a knowledge of the organization and the distribution of information among its various parts.

There is currently very little theory which can guide a designer in the construction of this model, though there are several guidelines that can be formalized [M9]. We present here a more intuitive formulation.

A data-base designer first has the problem of identifying all the relevant entities (person, place, thing or event) that are of interest to his organization. For each entity, the relevant attributes must be identified. This is not an easy task in practice. Different users may call the same entity or attribute by different names or have differing views of it. Some users may call different attributes by the same name. It seems that resolution of such problems is primarily a human activity, though some

automated help is available in the record-keeping phase through the use of *data dictionary* software to catalog various characteristics of the attributes—name, length, type, who generates it, who uses it, etc. Once the relevant attributes are identified, the data-base administrator has the problem of grouping attributes together into proper entities. Some possible guidelines for doing this are:

1) Determine those attributes (or concatenations of attributes), occurrences of which identify the entities being modeled. Call these attributes *identifiers* or *candidate keys*. For example, if students are the entity being modeled and each student has a unique student number as well as a social security number, then both student number and social security number are identifiers. Group together all identifiers for a particular entity.
2) Determine those other attributes of an entity that describe it, and there will be only one value of this attribute for a given entity, but the attribute is not part of an identifier. Consider grouping these nonidentifier attributes with the identifier or identifiers of the entity. For example, if students have a name and are admitted from a given high school, the items name and high school would be grouped with the student number and social security number.
3) If, for an identifier, there may be several values of an attribute, consider whether this "repeating item" may be better modeled as part of a separate entity. For example, if a student is enrolled in several courses, consider whether courses are not themselves separate entities worthy of being modeled. If they are, then see guideline 4 below. If not (for example, if educational degrees are considered to describe a student but are not entities in themselves), then either allocate a separate attribute for each of the finite number of repetitions (degree 1, degree 2, etc.), or associate a dependent, repeating structure with the entity.
4) If a one-to-many association exists between separate entities (for example, if there are student entities and dormitory entities, and many students reside at a dormitory but a given student does not reside at more than one dormitory), then place the identifier of the "one" with the "many" entity. In our example, we would place the dormitory identifier with the student entity. If there is a many-to-many association (for example, if a student is enrolled in many courses and a course has many students enrolled), then consider creating a new entity which describes this association. This entity will contain the pair of identifiers (namely, the student identifier and course identifier) along with any attributes that depend on *both* identifiers, for example, the grade received by the student in the course.

The three examples of one-to-one, one-to-many, and many-to-many associations do not exhaust all possibilities: a person has exactly two natural parents (a one-to-two association); Abrial [M1] has discussed such cases.

At the end of the information design process, the designer should have a full specification of those entities that are of interest, their necessary attributes, and the names of the entities and attributes; those attributes that are entity identifiers; and those attributes that identify other entities. Data structure design can then commence.

Concepts of Data Structure Design

Ideally, an information structure can be handled, as designed, by the DBMS. (Later we give examples of situations for which this is not the case). But there is still much to be done to complete the design. One task is to inform the system of the information structure. This is generally achieved by stating the design in a formal computer language (the data description language). The data-base definition or schema,[1] written

[1] The word schema is used because the definition is a "schematic" diagram of the data base.

in the data description language, is normally compiled into internal tables of the DBMS. Before this is possible, however, other design decisions must be made. Information structure design deals with entities and attributes. In contrast, data-base management systems manage records which are organized as indexed sequential files, hashed or direct access files, inversions, ring structures, or other structures [G1]. Thus it is necessary to reduce the entity and attribute level of data-base design to the world of computers, e.g., to choose among storage allocation strategies; to equate entities with records, perhaps representing associations with pointer structures; to decide whether some attributes should be indexed; and to choose between one-way and two-way lists. A DBMS offers a variety of options during the data definition stage, and the data-base administrator must choose a reasonable (if not optimal) alternative. Further, if any validity, integrity, or privacy constraints are to be enforced, these must also be stated.

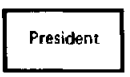
In later sections we detail some of the options for the design of data structures (and the accompanying data description language) as specified in the CODASYL *Data Description Language Journal of Development* [S2]. However, we must make the following point. Different DBMSs offer different options for the design of data structures. Usually options that require more human decisions offer increased control and thus the opportunity to tailor data structures to improve overall operation. On the other hand, some systems require less of the user, for example, those systems that make data structure choices "automatically" (that is, according to the algorithms that are in operation). These systems are generally "easier to use," but often operate inefficiently in cases when the structure chosen "automatically" is suboptimal.

The associations that the DBMS can make among records may not be sufficient to represent the associations at the information structure level. For example, the information structure may demand a many-to-many association between entities, while the system offers only hierarchical associations. In such cases, the data-base administrator may either have to adapt the information structure model so that it can be accommodated by the system, or find a way to simulate (by using a special procedure library) the capabilities not directly available in the DBMS.

Clearly, data structure design is a detailed, highly technical process which requires the expertise of a professional. During the design process, there is an immediate need for a notation that can be used to detail entities and their associations. One of the notations most widely used to model entities and their associations (and the one on which the DBTG system is based) is the *data structure diagram.*

Data Structure Diagrams

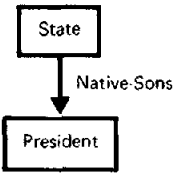
The data structure diagram was introduced by Bachman [N1]. This graphic notation uses two fundamental components—a rectangle and an arrow. A rectangle enclosing a name denotes an entity or record type that is dealt with in the data base. Thus:



indicates that there are occurrences of the record type named PRESIDENT. Each record type is composed of data items; but the particular item names are suppressed in this description, though they are defined in the full data-base description.

It is important to distinguish between a record type and an occurrence of a record of that type. For example, WASHINGTON and JEFFERSON denote two record occurrences within the record type PRESIDENT. We use the term *record* to denote a record occurrence and we use the term *record type* explicitly.

The second component in a data structure diagram is a directed arrow connecting two record types. The record type located at the tail of the arrow is called the owner-record type, and the record type located at the head is called the member-record type. This arrow directed from owner to member is called a *set type* and it is named. For example,



declares that a set type named NATIVE-SONS exists between the STATE (owner) and PRESIDENT (member) record types. The declaration that a set type exists specifies that there are associations between records of heterogeneous types in the data base. This allows the designer to interrelate diverse record types and thus to model associations between diverse entities in the real world.

It is also possible to have more than one member-record type in a set-type declaration. However, for simplicity of explanation, we do not treat that case in detail.

Just as the distinction was drawn between a record type and a record (occurrence), so a distinction is made between a set type and a set occurrence. The existence of a set type is declared by naming it, stating its owner-record type (exactly one) and its member-record type or types. A set occurrence is one occurrence of the owner-record type together with zero or more occurrences of each member-record type. Thus there is an occurrence of a set type whenever there is an occurrence of its owner-record type. A set occurrence is an association of one owner record with n (≥ 0) member records. It is this 1-to-n associating mechanism that is the basic building block for relating diverse records.

In every set occurrence, the following associations exist among the tenants (i.e., owner or member records) of that set occurrence:

- Given an owner record, it is possible to process the associated member records of that set occurrence.
- Given a member record, it is possible to process the associated owner record of that set occurrence.
- Given a member record, it is possible to process other member records in the same set occurrence.

Any implementation that satisfies these three rules is a valid implementation of the concept of a set type. Bachman [A1] has surveyed a number of possible implementation strategies.

Another rule of set occurrence is easily seen by examining the following example. Two occurrences of the NATIVE-SONS set type are shown in Figure 1, where each circle denotes a set occurrence. The owner record is denoted by its STATE-NAME item. Member records are denoted by their PRESIDENT-NAME item. In each set occurrence, there is one STATE record and the related PRESIDENT records. Since there are no Presidents who were born in Maine, the Maine set occurrence involves only the owner record (recall that there is a set occurrence whenever there is an owner-record occurrence). A set occurrence with no member-record occurrences is called an "empty set."

The reader should realize that sets, as the term is used here, bear very little relationship to the sets used in mathematics. For example, the empty set just defined has an element! To avoid possible confusion, some authors have preferred to call these structures "data structure sets" [D1] or "owner-coupled sets" [D2].

There is one other rule of set occurrence: a given member record may be associated with only one set occurrence of a given type. A member record cannot simultaneously belong to two owner records for the same set type. In terms of the circle diagrams of Figure 1, any overlap among two circles of the same type is illegal; it would be illegal to make EISENHOWER a native son of both KANSAS and NEBRASKA (see Figure 2). We give other consequences of this rule later, but we emphasize that this rule is fundamental to the understanding of sets. Because of this rule, it is possible

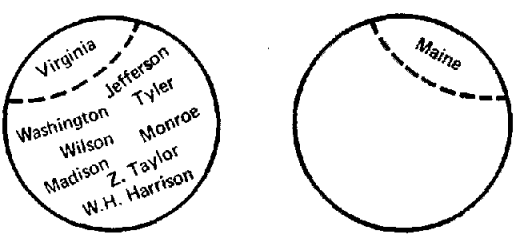


FIGURE 1. NATIVE-SONS set occurrences.

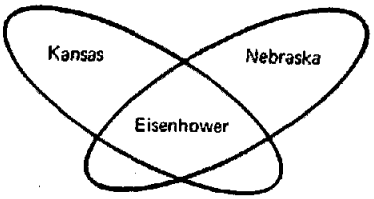


FIGURE 2. Illegal set occurrences.

to regard a set type as a function with a domain which is the occurrences of the member-record type or types and with a range which is the occurrences of the owner-record type.

We now present some examples of data structure diagrams. Set occurrences are denoted here in terms of one possible implementation—ring structures. However, it is important to understand that a set declaration does not imply a method of implementation. Any implementation that supports the rules just discussed is possible.

Hierarchies (1-to-n Relationships)

A hierarchy is a common data structure. In this structure, one record owns 0 to n occurrences of another record type; each of those occurrences in turn owns 0 to n occurrences of a third record type, etc.: no record owns any record that owns it, either directly or indirectly. For example, STATE-PRESIDENT-ADMINISTRATION, as shown in Figure 3, is a hierarchy. A possible occurrence of this hierarchy is shown in Figure 4.

In the ring structure shown in Figure 4, there is an access path directed from the owner record to its "first" member record, and from the first member record to the "next" member record. Each member record, except the last, has a "next" member record, and each member record, except the first, has a "prior" member record accessed by traversing the ring. In addition, there is an access path directed from each member record back to the "owner" record. (Note, however, that a traversal to the "next" member record from the last member record or to the "prior" member record from the "first" member record yields a diagnostic.) We have now satisfied all the requirements which were stated.

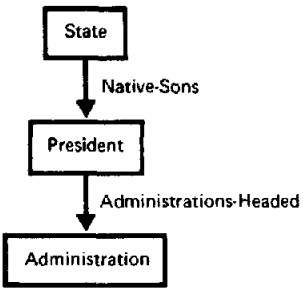


FIGURE 3. Data structure diagram of hierarchy.

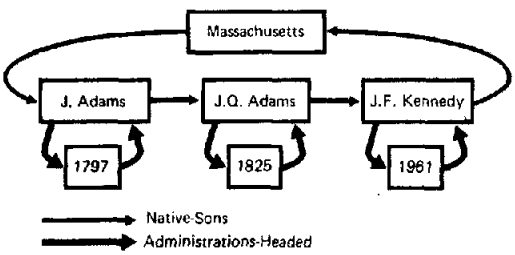
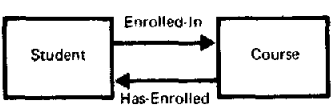


FIGURE 4. Occurrence of a hierarchy.

Many-to-Many Relationships

A second common structure relates two record types which stand in a many-to-many relationship to each other. Consider, for example, STUDENT and COURSE record types. A STUDENT may be taking many courses and a COURSE may have many students. To model this, it is not possible to construct the diagram:



since a COURSE with more than one STUDENT would simultaneously be a member in two set occurrences of ENROLLED-IN, violating the rule of unique ownership. This is illustrated in Figure 5. Similarly, when a student takes more than one course, the HAS-ENROLLED set condition is violated (see Figure 6).

The usual way of representing many-to-many relationships is to define a third record type, as shown in Figure 7. This new record type is used to relate the two other record types; it contains any information that pertains specifically to both STUDENT and COURSE, e.g., GRADE.

Figure 8 shows an occurrence of the many-

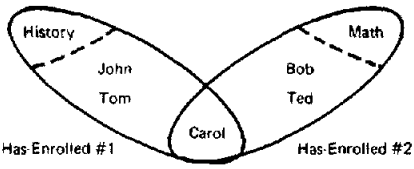


FIGURE 5. Course with more than one student. Two occurrences of ENROLLED-IN with a shared member.

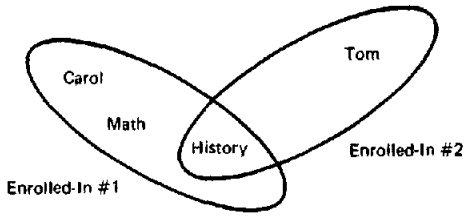


FIGURE 6. Student enrolled in more than one course. Two occurrences of HAS-ENROLLED with a shared member.

to-many relationship structure. It is possible to ascertain the classes a given student is enrolled in by following the ENROLLED-IN set; whenever a GRADE record is found, one switches to the HAS-ENROLLED set to find the owner record. Then one returns to the GRADE record, switches back to the ENROLLED-IN set, and continues to the next GRADE record, if any, etc. The retrieval of enrolled students given a class record is similar. When viewed in this way, it is clear that n-dimensional associations are possible when the "intersection record" in the array is a member of n set types. If the set types are implemented using ring structures, the structure is like a sparse array; this is an appropriate structure in many applications.

Another example, used in the presidential data-base example, is shown in Figure 9. A President may be associated with many Congresses and a Congress may serve with more than one President (as a result of a death in office, for example). In this case, a many-to-many relationship exists. Whether or not the intersection record contains data meaningful to the user, which is frequently the case, the introduction of the third record type is generally a necessity. In this example, the number and dates of speeches addressed to a joint session of a Congress by a President are examples of

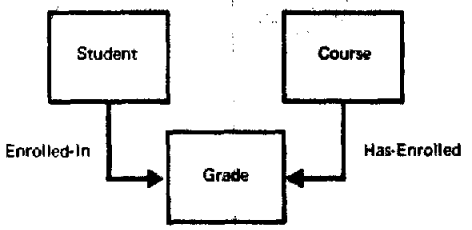


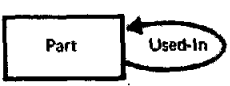
FIGURE 7. Representation of many-to-many relationships.

possible data items within the CONGRESS-PRES-LINK record type.

Complex Relationships Using Data Structure Diagrams

The preceding section presented examples of cases where a record type was a member in more than one set type. It is also possible for a given record type to be the owner of more than one set type. Figure 10 illustrates such a case. As shown in this figure, each President won a number of Elections and headed a number of Administrations. By using data structure diagram notation, a data-base administrator can define "networks," where a record type may serve as a member in one or more set types and as owner in one or more other set types.

It is also possible to represent "recursive" structures such as the parts explosion or bill of materials structure by using data structure diagram notation. In the parts explosion structure, a part is composed of other parts, which in turn are composed of other parts, etc. While data structure diagram notation does not forbid the same record type from being both owner and member in the same set type, this rule has been adopted by the *Data Description Language Journal of Development* [52]. For example,



is an illegal structure. However, it is still possible to represent this structure through a "relationship record type," as shown in Figure 11. The ASY record type represents an assembly of subparts. This structure has an occurrence diagram as shown in Figure 12 (where horizontal lines are HAS-SUBSTRUCTURE set occurrences and vertical

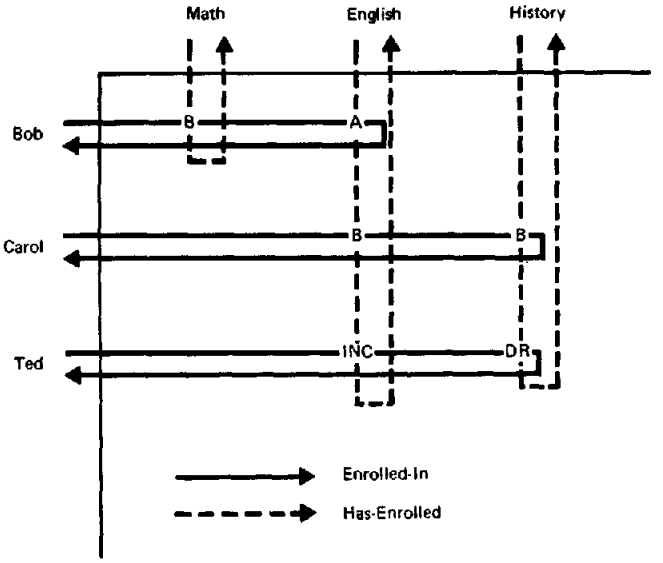


FIGURE 8. Occurrence of many-to-many relationship.

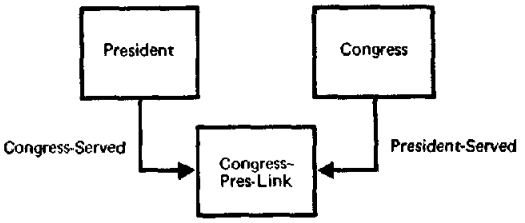


FIGURE 9. Many-to-many association in Presidential data base.

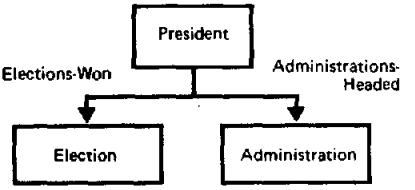


FIGURE 10. Multiple set ownership.

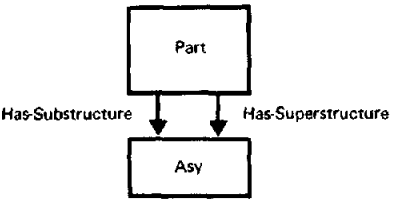
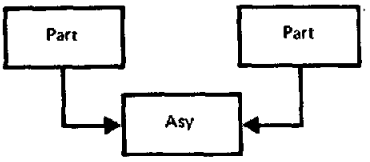


FIGURE 11. Parts explosion structure.

lines are HAS-SUPERSTRUCTURE set occurrences.)

Note that as a first approximation, we can consider Figure 11 a special case of the many-to-many relationship. That is, the diagram:



has been "merged" because both sides are of the same record type. By using the array notation for the occurrence diagram (see Figure 12), we can accomplish a "parts explosion traversal" using the "find next, switch sets, and find the owner in the other set" traversal. The reader should be able to determine how a "parts implosion" traversal could be accomplished.

Figure 12 does not illustrate one concept: that there is only one occurrence (not two) of a part; that is, there is one bike, one frame, etc. By "folding" the array, we obtain Figure 13, where there is only one occurrence of each part, with one part description, quantity on hand, etc., no matter where the part is used. The structure can also be regarded as an acyclic directed graph, as shown in Figure 14, where the

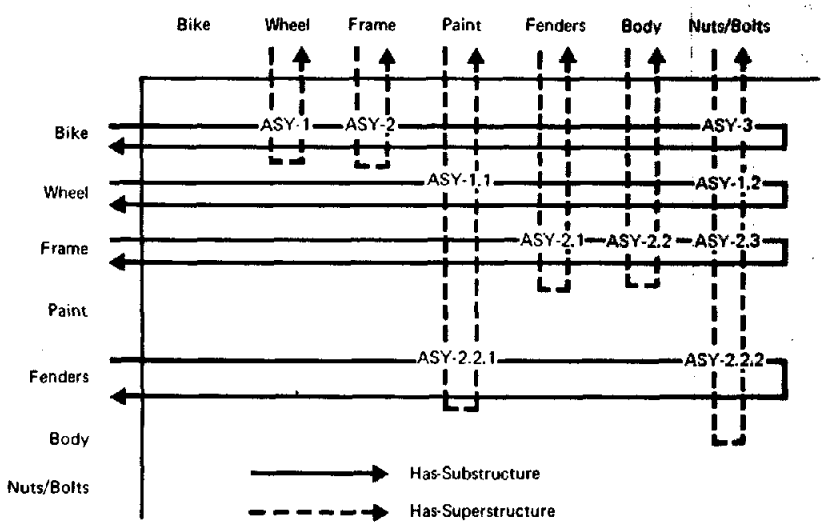


FIGURE 12. Occurrence of parts explosion.

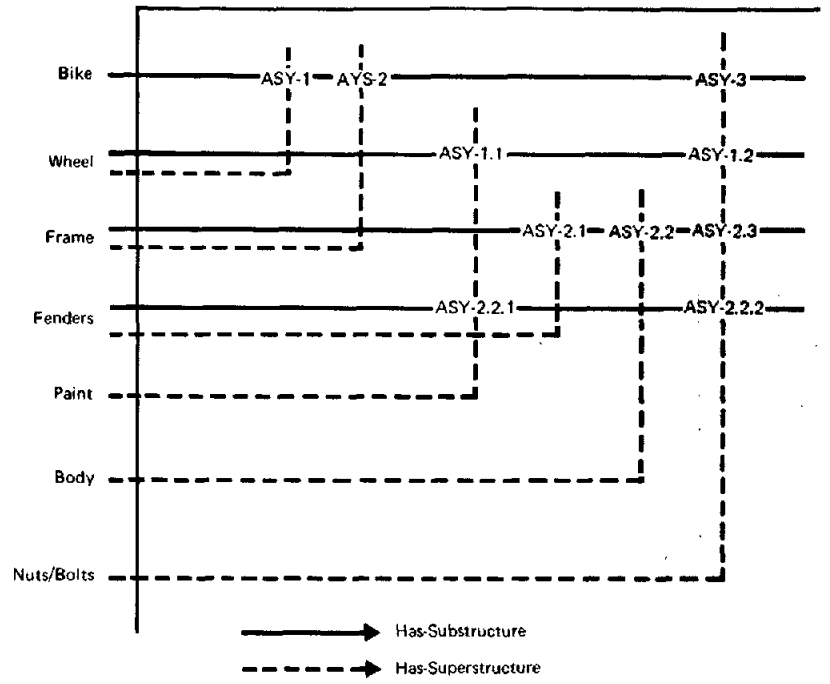


FIGURE 13. Parts explosion with records merged.

contents of the ASY records are shown as labels on the edges of the graph.

Presidential Data Base

Figure 15 shows our example, the Presidential data base, using data structure diagram notation. Record types are PRESIDENT, CONGRESS, ADMINISTRATION, STATE, ELECTION, AND CONGRESS-PRES-LINK. These record types represent the corresponding entities. In addition, the following associations are modeled using set types:

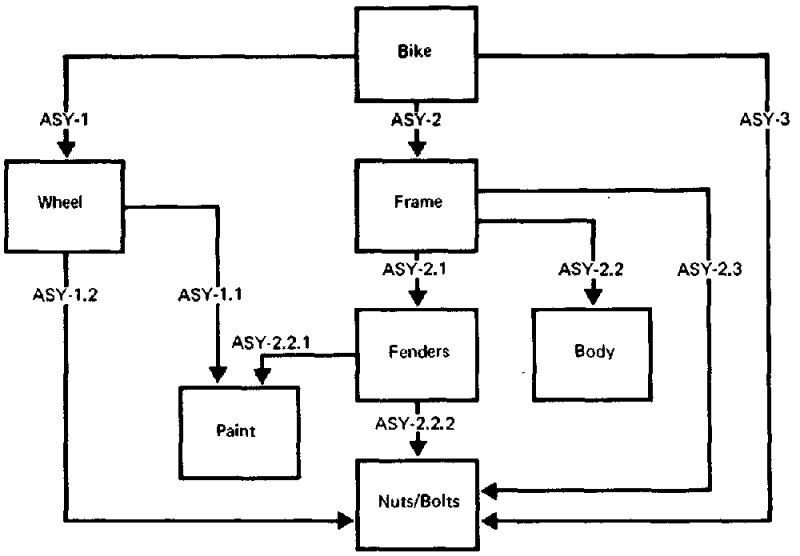


FIGURE 14. Parts explosion as a graph.

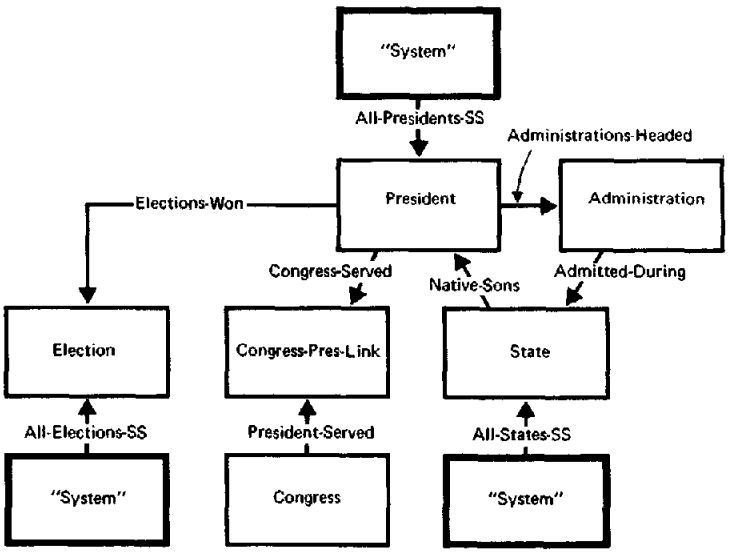


FIGURE 15. Data structure diagram of Presidential data base.

- Each President is associated with the Elections he won.
- Each President is associated with his Administrations in the ADMINISTRATIONS-HEADED set type.
- Each President is associated with a number of Congresses in a many-to-many relationship.
- Each State is associated with a number of Presidents who are its native sons.
- Each State (except the original thirteen) was admitted during an Administration.

Figure 15 also shows three set types where the owner is "the system." These are called singular sets and are discussed in more detail in Section 2, Sample Data-Base Application. Based on the rules regarding member-record accessibility, it is sufficient to note here that the singular set type implies that there are three access paths

(however encoded physically), an access path that passes through all occurrences of PRESIDENT (ALL-PRESIDENTS-SS), an access path that passes through all occurrences of ELECTION (ALL-ELECTIONS-SS), and an access path that passes through all STATE records (ALL-STATES-SS). These singular sets represent *entry points* into the data base in the sense that particular Presidents, Elections, or States may be located by values of their constituent items, with no need to have accessed other records in the data base.

2. SAMPLE DATA-BASE APPLICATION

The DBTG specifications include several languages which are to be used to describe and manipulate data. In this section we present an example of a data base which uses these languages. The subsection titled Presidential DataBase in the DDL discusses the use of the *Schema data description language* (*DDL*) to describe a data structure. The sub-schema of the Presidential Data Base subsection presents a *Sub-schema language* description of that part of the data base which is to be processed by an application program. Subsection Sample Retrieval Program through subsection Traversing an *m:n* Relation in the COBOL DML illustrate the use of the COBOL programming language, as augmented by the DBTG *data manipulation language* (DML), to access and update the stored data. Some of the more complex issues, particularly in the DML, are not discussed here, though some are presented in Section 3, Advanced Features.

Presidential Data-Base in the DDL

At the end of the preceding section we introduced the components of the Presidential data base, by using a data structure diagram. We use this sample data base to illustrate the concepts presented in the remainder of this paper.

Here we give a description of the Presidential data base in the Schema DDL. The syntax is that adopted by the 1973 *CODASYL Data Description Language* (*DDL*) *Journal of Development* [S2].

A description of a data base in the Schema DDL consists of four major sections:

- an introductory clause
- one or more AREA clauses
- one or more RECORD clauses
- one or more SET clauses.

The introductory clause is used to name the data base and to state certain global security and integrity constraints.

An *area* is a logical subdivision of the data base, which in many implementations corresponds to a file or data set in an operating system. While we usually think of a data base as being a single integrated collection of data, it is often desirable to subdivide such a data base into multiple logical subunits, in order to implement special security and integrity constraints and to provide a mechanism to control the performance and cost of implementation. Data-base security can be increased by placing highly sensitive data in logically separate areas and by placing special controls over those areas. Of course, physical separation of the areas may be used to increase security. Data-base integrity can be improved by placing critical data in areas that are safe from harm or are often duplicated, while high performance areas may need to reside on high speed devices. Similarly, costs can be minimized by placing infrequently used data in areas which reside on less costly devices. Any logical or physical reason for splitting the data can utilize the area concept.

The area description in the Schema DDL allows the data-base administrator to name these subdivisions of the data base and to specify which of the areas contain which record types. The actual mapping of areas to one or more physical storage volumes is under the control of a separate device-media control language (DMCL). The need for a DMCL was noted but has been left unspecified by the DBTG and its successor committees. In many implementations of the DBTG specifications, the functions of the DMCL are incorporated in the job control or command language of the operating system. Several advanced features of areas are described in detail in Section 3, Subsection Areas.

For every record type in a data base there exists a description in the Schema DDL. A schema record description consists of information about the record type, such as its storage and location mechanism, and information about the area or areas in which occurrences of the record type may be placed.

The record description contains a description of all data items that constitute the record type. A record occurrence in the stored data base consists of occurrences of each data-item type that constitute the record type. These record occurrences are the units of data transfer between the stored data base and an application program. Thus the application programmer interface uses a "record at a time" logic (one record occurrence is delivered or stored for each command) in accessing the data base.

For each set type in a data base, a separate set description is written in the Schema DDL. Each set description names the set type, specifies the owner-record type and member-record type or types, and states detailed information on how occurrences of the set are to be ordered and selected.

The introductory section of a schema description consists of a statement naming the schema, and certain security and integrity constraints. For our sample data base, this introductory section is:

```
SCHEMA NAME IS PRESIDENTIAL;
  PRIVACY LOCK FOR COPY IS 'COPY PASSWORD'
  PRIVACY LOCK FOR ALTER IS PROCEDURE CHECK-AUTHORIZATION;
```

This names the schema (PRESIDENTIAL) and specifies certain privacy criteria (LOCKS) that must be met when attempting to access the stored copy of this schema. In fact, the schema language provides four types of privacy associated with accessing the schema: ALTER, COPY, DISPLAY, and LOCKS. ALTER controls the conditions under which the contents of the stored schema may be altered. (Note that this is not the same as altering the data base. Here we are controling the ability to modify the schema itself, e.g., to add a new record type to the data-base description.) COPY controls who may copy the stored schema into a Sub-schema (see subsection Sub-schema of the Presidential Data Base). DISPLAY similarly controls the printing of the stored schema. LOCKS controls the altering of privacy locks in the schema, which is analogous to controling who may change the key to the key cabinet.

In the preceding example we specify that when the Sub-schema processor is requesting a copy of the schema for use in processing a sub-schema, it must supply the literal 'COPY PASSWORD' as a privacy key. Similarly, to modify the schema, a person must satisfy the privacy constraints imposed by the procedure named CHECK-AUTHORIZATION, (which is written by the data-base administrator). Such a procedure is automatically invoked every time anyone attempts to modify the stored schema. Such a procedure is termed a *data-base procedure* (see Section 3, subsection Data-Base Procedures).

Following the introductory section, we can intermix descriptions of the areas, records, and sets that make up the data base, subject to the constraint that an area description must precede the description of all records that may be placed in that area and the constraint that all records that make up a set must be described before the description of the set using them. For simplicity, we present all area descriptions, followed by all record descriptions, followed by all set descriptions.

In our example we restrict attention to a single area. All record occurrences are in this single area, which is described as follows:

```
AREA NAME IS PRESIDENTIAL-AREA;
ON OPEN FOR UPDATE CALL UPDATE-CHECK;
```

This names the area as PRESIDENTIAL-AREA and specifies that, if the area is opened for update, a data-base procedure (UPDATE-CHECK) will be invoked.

As shown in Figure 15 the Presidential

data base consists of six record types: PRESIDENT, ADMINISTRATION, STATE, CONGRESS, CONGRESS-PRES-LINK, and ELECTION. We illustrate several of these records in detail and then give a short description of the remaining records.

The PRESIDENT record may now be defined as:

```
RECORD NAME IS PRESIDENT
  LOCATION MODE IS CALC
    USING LAST-NAME, FIRST-NAME, DUPLICATES ARE NOT ALLOWED
  WITHIN PRESIDENTIAL-AREA

02  PRES-NAME
      03  LAST-NAME  PIC "A(10)"
      03  FIRST-NAME  PIC "A(10)"
02  PRES-DATE-OF-BIRTH
      03  MONTH-B  PIC "A(9)"
      03  DAY-B  PIC "99"
      03  YEAR-B  PIC "9999"
02  PRES-HEIGHT  PIC "X(10)"
02  PRES-PARTY  PIC "A(10)"
02  PRES-COLLEGE  PIC "A(10)"
02  PRES-ANCESTRY  PIC "A(10)"
02  PRES-RELIGION  PIC "A(10)"
02  PRES-DATE-OF-DEATH
      03  MONTH-D  PIC "A(9)"
      03  DAY-D  PIC "99"
      03  YEAR-D  PIC "9999"
02  PRES-CAUSE-DEATH  PIC "X(10)"
02  PRES-FATHER  PIC "A(10)"
02  PRES-MOTHER  PIC "A(10)"
```

After naming the record PRESIDENT, the LOCATION MODE clause specifies certain information about the way of placing and retrieving record occurrences. In this example, CALC is specified. CALC refers to address calculation or hashing. The clause specifies that a PRESIDENT record is positioned according to the values of its data items LAST-NAME and FIRST-NAME. Note the addition of the DUPLICATES ARE NOT ALLOWED clause, which specifies that if an attempt is made to store a new (but duplicate) occurrence of the PRESIDENT record the system should reject the request and notify the application program of the rejection. Of course, this means that the data base would not support two presidents with same first and last names!

The WITHIN clause specifies in which area or areas occurrences of the record may be placed: in our sample data base it is PRESIDENTIAL-AREA, the only area.

Next, the data items that constitute the record are specified. The names given for the items are self-explanatory. Associated with each data item name is a picture. The Schema language also provides for the inclusion of many additional data-item attributes, some of which are discussed in Section 3, Advanced Features. The PICTURE clause describes the number and type of character positions that make up the data item. For example, a picture of A(10) specifies in a way similar to COBOL or PL/I that the corresponding item is made up of 10 alphabetic positions. A picture code of X specifies an alphanumeric position, and a picture code of 9 specifies a numeric position. Data items can be collected into groups, which are collections of data items and (optionally) other groups that are named. For example, the group PRES-NAME consists of the data items LAST-NAME and FIRST-NAME. Such grouping provides for good documentation, as well as easing the programming task, since COBOL provides for primitives for manipulating groups as well as data items.

The ADMINISTRATION record is defined as follows:

```
RECORD NAME IS ADMINISTRATION
  LOCATION MODE IS VIA ADMINISTRATIONS-HEADED SET
  WITHIN PRESIDENTIAL-AREA
  02  ADMIN-KEY  PIC "XXX"
  02  ADMIN-INAUGURATION-DATE
    03  MONTH  PIC "99"
    03  DAY  PIC "99"
    03  YEAR  PIC "9999"
```

This record uses a location mode of VIA, which specifies that the record occurrences are, where possible, to be located near other record occurrences in the ADMINISTRATIONS-HEADED set to which it is linked. Thus all occurrences of the ADMINISTRATION record for a particular administration would be placed near each other and the owning PRESIDENT record. The VIA specification advises the DBMS of the desirability of clustering record occurrences on secondary storage. Of course, this is only a request to attempt to cluster the records; the system will follow an implementor-defined algorithm. The performance of this algorithm will depend, in part, on storage allocations made by the data-base administrator.

The remaining record descriptions are presented without further discussion; they

follow the same format as do the previous two.

```
RECORD NAME IS STATE
  LOCATION MODE IS CALC
      USING STATE-NAME DUPLICATES ARE NOT ALLOWED
  WITHIN PRESIDENTIAL-AREA
  02  STATE-NAME  PIC "X(10)"
  02  STATE-YEAR-ADMITTED  PIC "9999"
  02  STATE-CAPITAL  PIC "X(10)"

RECORD NAME IS ELECTION
  LOCATION MODE IS VIA ALL-ELECTIONS-SS
  WITHIN PRESIDENTIAL-AREA
  02  ELECTION-YEAR  PIC "9999"
  02  ELECTION-WON-ELECTORAL-VOTES  PIC "999"

RECORD NAME IS CONGRESS
  LOCATION MODE IS CALC
      USING CONGRESS-KEY DUPLICATES ARE NOT ALLOWED
  WITHIN PRESIDENTIAL-AREA
  02  CONGRESS-KEY  PIC "XXXX"
  02  CONGRESS-NUM-PARTY-SENATE  PIC "999"
  02  CONGRESS-NUM-PARTY-HOUSE  PIC "999"

RECORD NAME IS CONGRESS-PRES-LINK
  LOCATION MODE IS VIA CONGRESS-SERVED SET
  WITHIN PRESIDENTIAL-AREA
```

When the record types that make up the data base have been described, the process of relating record types through sets can begin. The simplest type of set description, termed a *singular set*, is one where the owner of the set is implicitly the "system." Because such a set has a unique owner, there can be only one occurrence of the set (thus the term singular). One way to use the singular set is to collect all records of a particular type for sequential access. In the Presidential data base the record types PRESIDENT, ELECTION, and STATE are to be members of three singular sets. The descriptions for these three sets are:

```
SET NAME IS ALL-PRESIDENTS-SS
  OWNER IS SYSTEM
  ORDER IS PERMANENT SORTED BY DEFINED KEYS
    DUPLICATES ARE LAST
  MEMBER IS PRESIDENT MANDATORY AUTOMATIC
    KEY IS ASCENDING LAST-NAME IN PRES-NAME
    SET SELECTION IS THRU ALL-PRESIDENTS-SS
      OWNER IDENTIFIED BY SYSTEM

SET NAME IS ALL-ELECTIONS-SS
  OWNER IS SYSTEM
  ORDER IS PERMANENT SORTED BY DEFINED KEYS
    DUPLICATES ARE NOT ALLOWED
  MEMBER IS ELECTION MANDATORY AUTOMATIC
    KEY IS ASCENDING ELECTION-YEAR
    SET SELECTION IS THRU ALL-ELECTIONS-SS
      OWNER IDENTIFIED BY SYSTEM

SET NAME IS ALL-STATES-SS
  OWNER IS SYSTEM
  ORDER IS PERMANENT SORTED BY DEFINED KEYS
    DUPLICATES ARE NOT ALLOWED
  MEMBER IS STATE MANDATORY AUTOMATIC
    KEY IS ASCENDING STATE-NAME
    SET SELECTION IS THRU ALL-STATES-SS
      OWNER IDENTIFIED BY SYSTEM
```

Since all three set descriptions follow the same format, only the first set is described. After naming the set type, the OWNER IS clause names the owner-record type. In these three examples, a declaration of SYSTEM as owner denotes singular sets. The ORDER IS clause specifies the order in which member-record occurrences may be presented sequentially to an application program. Here we specify the order of the set to be sorted based on keys stated as part of the description of the member record (the DEFINED KEYS option). The PERMANENT option (required for sorted sets) specifies that an application program may not make (permanent) alterations to the order of a set. Thus any ordering changes made by an application program are local to the execution of that program and do not permanently affect the data base.

The DUPLICATES clause specifies whether duplicates are permitted for the defined keys and, if they are, how they should be handled. The DUPLICATES ARE LAST option specifies that when an attempt is made to store a member record that has duplicated the keys of another member-record occurrence, the system should place the new record occurrence after all existing record occurrences that have the same key.

The MEMBER subentry names the member-record types that make up this set type. In all of the examples from the Presidential data base, a set type is composed of an owner-record type and a single member-record type, though the specifications allow for multiple member-record types in a single set type.

The record type which acts as a member of the set is named in the MEMBER IS entry. This is followed by a statement concerning the removal of member-record occurrences from set occurrences that is allowed and a statement specifying how member record occurrences are initially placed in set occurrences. The MANDATORY specification indicates that once an occurrence of PRESIDENT is placed in ALL-PRESIDENTS-SS it may not be removed from the set occurrence without actually deleting the record occurrence. The AUTOMATIC specification indicates that each time a new occurrence of the

PRESIDENT record is stored in the data base, it is automatically inserted in the ALL-PRESIDENTS-SS set. The combined effect of MANDATORY AUTOMATIC is that *all* occurrences of the PRESIDENT record will be a member of the ALL-PRESIDENTS-SS.

The SET SELECTION clause allows the system to support the AUTOMATIC insertion of member-record occurrences into the appropriate set occurrences. Since all sets described have only one occurrence, the system may (trivially) select the proper set occurrence. For singular sets, the SET SELECTION clause is a restatement of the fact that the set is singular. The other set descriptions present more complex cases of set selection.

We now describe the ELECTIONS-WON set:

```
SET NAME IS ELECTIONS-WON
  OWNER IS PRESIDENT
  ORDER IS SORTED PERMANENT BY DEFINED KEYS
    DUPLICATES ARE NOT ALLOWED
  MEMBER IS ELECTION MANDATORY AUTOMATIC
    KEY IS ELECTION-YEAR
  SET SELECTION IS THRU ELECTIONS-WON
    OWNER IDENTIFIED BY CALC-KEY
```

This set description basically follows the same format as do the previous descriptions, with the exception of the SET SELECTION clause. Since there now exists an occurrence of ELECTIONS-WON corresponding to each occurrence of PRESIDENT, we need to tell the system how to relate a new occurrence of ELECTION to the corresponding occurrence of PRESIDENT. This arises, for example, when a new ELECTION record is stored, since we have declared (with AUTOMATIC) that each new occurrence of ELECTION must be placed in an occurrence of the ELECTIONS-WON set. By stating OWNER IDENTIFIED BY CALC-KEY, the system is told that the application program will state the necessary CALC-KEY of the corresponding PRESIDENT. By referring to the description for a PRESIDENT record, we note that the CALC-KEY is LAST-NAME, FIRST-NAME. Therefore, before we attempt to STORE a new occurrence of ELECTION in the data base, we must present the system with values for the winning PRESIDENT LAST-NAME and FIRST-NAME. Since there is a one-to-one correspondence between owner-record occurrences and set occurrences, to identify a PRESIDENT-record occurrence uniquely identifies the corresponding ELECTIONS-WON set occurrence and allows insertion of the occurrence of the new ELECTION record into this set.

As discussed in Section 1, Design of a Data Base, there is an $m:n$ relationship between PRESIDENTS and CONGRESSES. We relate these two records as follows by use of an intermediate "link record," which in this case is CONGRESS-PRES-LINK, shown in Display 1 below.

The sole purpose for CONGRESS-PRES-LINK is to act as a link between PRESIDENT and CONGRESS. As such, it is not meaningful to define a sort order, and it is declared IMMATERIAL; the system may keep occurrences in an implementor-defined order.

For both sets, the SET SELECTION is determined by the CALC-KEY of the respective owner record. For example, when an application program determines that a particular PRESIDENT is to be linked to a particular CONGRESS, the program must provide a value for the CALC-KEY of PRESIDENT (LAST-NAME, FIRST-NAME) and CONGRESS (CONGRESS-KEY). When the program issues a STORE operation on CONGRESS-PRES-LINK

Display 1

```
SET NAME IS CONGRESS-SERVED
  OWNER IS PRESIDENT
  ORDER IS PERMANENT IMMATERIAL
  MEMBER IS CONGRESS-PRES-LINK MANDATORY AUTOMATIC
    SET SELECTION IS THRU CONGRESS-SERVED
      OWNER IDENTIFIED BY CALC-KEY

SET NAME IS PRESIDENT-SERVED
  OWNER IS CONGRESS
  ORDER IS PERMANENT IMMATERIAL
  MEMBER IS CONGRESS-PRES-LINK MANDATORY AUTOMATIC
    SET SELECTION IS THRU PRESIDENT-SERVED
      OWNER IDENTIFIED BY CALC-KEY
```

(CONGRESS-PRES-LINK is an AUTOMATIC member of both CONGRESS-SERVED and PRESIDENT-SERVED), the record is linked to both sets.

The other sets relate the records PRESIDENT, ADMINISTRATION, and STATE:

```
SET NAME IS ADMINISTRATIONS-HEADED
  OWNER IS PRESIDENT
  ORDER IS PERMANENT SORTED BY DEFINED KEYS
    DUPLICATES ARE NOT ALLOWED
  MEMBER IS ADMINISTRATION MANDATORY AUTOMATIC
    KEY IS ASCENDING ADMIN-KEY
    SET SELECTION IS THRU ADMINISTRATIONS-HEADED
      OWNER IDENTIFIED BY CALC-KEY
SET NAME IS ADMITTED-DURING
  OWNER IS ADMINISTRATION
  ORDER IS PERMANENT SORTED BY DEFINED KEYS
    DUPLICATES ARE NOT ALLOWED
  MEMBER IS STATE MANDATORY MANUAL
    KEY IS ASCENDING STATE-YEAR-ADMITTED
    SET SELECTION FOR ADMITTED-DURING
      IS THRU ADMINISTRATIONS-HEADED
        OWNER IDENTIFIED BY CALC-KEY
      THEN THRU ADMITTED-DURING WHERE
        OWNER IDENTIFIED BY ADMIN-KEY
SET NAME IS NATIVE-SON
  OWNER IS STATE
  ORDER IS PERMANENT SORTED BY DEFINED KEYS
    DUPLICATES ARE LAST
  MEMBER IS PRESIDENT MANDATORY AUTOMATIC
    KEY IS ASCENDING LAST-NAME IN PRES-NAME
    SET SELECTION IS THRU NATIVE-SON
      OWNER IDENTIFIED BY CALC-KEY
```

The format for the sets ADMINISTRATIONS-HEADED and NATIVE-SON follows that of the previous sets. In both cases the owners of the respective sets have CALC-KEYS defined for them, and these calc-keys are used to determine an occurrence of the respective sets when necessary.

The ADMITTED-DURING set introduces several new concepts. The first of these is the concept of MANUAL membership. In previous examples all member records have been declared AUTOMATIC, meaning that when a new occurrence of the member record is stored in the data base, it is to be automatically included in an occurrence of the set. There are two important reasons an AUTOMATIC membership attribute would be improper here. The first reason concerns the cyclic structure of the three sets, as shown by the data structure diagram of Figure 16.

If each of the records, PRESIDENT, ADMINISTRATION, STATE, shown in Figure 16, were defined as automatic members of their respective sets, problems would occur. Automatic membership implies that

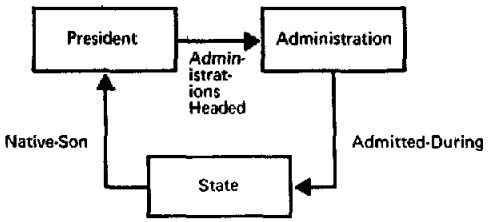


FIGURE 16. Part of Presidential data base.

whenever a new record occurrence is stored in the data base, it will be placed in an occurrence of each set in which it is an automatic member. However, there is a one-to-one correspondence between set occurrences and owner-record occurrences. In order for a set occurrence to exist, there must be an occurrence of its owner. In the context of Figure 16, if all three records are automatic members of their respective sets, it is impossible to store the first record occurrence of any of the record types (because storing an automatic member record *requires* an owner).

The solution to the dilemma, as required by the CODASYL specifications, is the declaration that one of the member records (in a cyclic structure) is MANUAL. In this example, we declare STATE to be a manual member of ADMITTED-DURING. The implication is that appropriate occurrences of STATE must be stored in the data base first; then store occurrences of PRESIDENT, and finally occurrences of ADMINISTRATION. After the record occurrences have been stored, the application program can traverse the structure and can manually link together STATE and ADMINISTRATION record occurrences.

The second and more fundamental reason for declaring STATE a manual member concerns the nature of the data being stored. The original thirteen states of the Union existed from the beginning of the Country, and were not "added" during any President's Administration. It would be improper to require that these States be placed in any occurrence of ADMITTED-DURING; hence STATE must be a manual member of ADMITTED-DURING. In general, when a member record participates

conditionally in a set, the MANUAL attribute is used.

The second major concept introduced by the ADMITTED-DURING set concerns its SET SELECTION clause. In all previous examples, SET SELECTION is either obvious (for singular sets), or apparent directly through the CALC-key of the owner record of the set. If we refer to the description of the ADMINISTRATION record, we note that the location mode for the record is VIA the ADMINISTRATIONS-HEADED set. Thus we may not identify the owner of ADMITTED-DURING by presenting a CALC-key, since ADMINISTRATION is not a "calced" record. In this case we chose an option which specifies that we first select an occurrence of ADMINISTRATIONS-HEADED by presenting the CALC-key of PRESIDENT. The result of this operation is the determination of a set occurrence of ADMINISTRATIONS-HEADED. Then, from among the occurrences of ADMINISTRATION that participate in the ADMINISTRATIONS-HEADED just selected, we select an ADMINISTRATION record by providing its ADMIN-KEY. Once we have identified an occurrence of ADMINISTRATION, we have also identified an occurrence of ADMITTED-DURING. It is important to note that the second phase of this process, that of selecting an occurrence of ADMINISTRATION based on a value for ADMIN-KEY, is performed only among those occurrences selected during the first phase. Thus there is no requirement that ADMIN-KEY be unique across the data base, but only that it be unique within individual occurrences of ADMINISTRATIONS-HEADED.

We have now described the Presidential data base. The next phase is the declaration of one or more sub-schemas.

A Sub-schema of the Presidential Data Base

The Schema defines the entire data base that is stored and available to all users. But an application program may need to view only some parts of the data base, as well as to make some simple changes. The Sub-schema DDL allows a data-base administrator to delimit which portions of a data base (as declared in the Schema) are to be made available to the application program or programs. It also enables the data-base administrator to make some changes in the way that the stored data is presented to an application program; for example, the internal representation of a data item might be changed from binary in the data base to decimal when passed to the application program. The particular sub-schema described here is defined for use with COBOL. CODASYL intends to develop sub-schema languages for other programming and self-contained languages.

The application programs discussed later need only the information in that part of the data base which is shown in Figure 16. We therefore define a sub-schema to be used for that information only. The first part of the definition gives names and privacy keys:

```
SS PRES-ADMIN-STATE-INFO WITHIN
   SCHEMA PRESIDENTIAL
   PRIVACY KEY IS 'COPY PASSWORD'.
```

This calls the sub-schema PRES-ADMIN-STATE-INFO, and names the privacy key which allows the sub-schema processor to access the stored schema; the schema definition has a COPY privacy lock.

Next, the necessary areas are defined; we must select those areas that are required by the application programs that access the data base through this sub-schema. Due to conflicts with COBOL reserved words, the term AREA in the Schema DDL is called a REALM in the sub-schema and COBOL DML. Since there is only one area in the schema, we specify that it is to be made available:

```
REALM DIVISION
RD PRESIDENTIAL-AREA.
```

Next we define the sets required in this sub-schema:

```
SET DIVISION.
SD ALL-PRESIDENTS-SS.
SD ALL-STATES-SS.
SD ADMINISTRATIONS-HEADED.
SD ADMITTED-DURING.
SD NATIVE-SON.
```

Finally we name the records and corresponding data items that are part of this

sub-schema:

```
RECORD DIVISION.
01  PRESIDENT.
    02  PRES-NAME.
        03  LAST-NAME   PIC A(10).
        03  FIRST-NAME  PIC A(10).
01  ADMINISTRATION.
    02  ADMIN-KEY   PIC XXX.
    02  ADMIN-INAUGURATION-DATE.
        03  MONTH  PIC 99.
        03  YEAR   PIC 9999.
01  STATE.
    02  STATE-NAME  PIC X(10).
    02  STATE-YEAR-ADMITTED  PIC 9999
```

In this record section we not only eliminate unnecessary records from the sub-schema, but include only those data items of interest to an application program using this sub-schema.

Sample Retrieval Program

Once the schema and sub-schema are defined, application programs can be written to store and access data. The DBTG specifications do not include a special data-base population function; therefore the first program written is normally a data-base load program. We shall, however, assume that a data base does exist for our examples.

The first program presented here finds all of the States that have *more* than one President as a native son. Then we print out the name of the State with its number of Presidents. Queries like this, which involve traversing almost the entire data base and performing counting are well suited for DBTG-type systems.

The DML presented in this section is designed to augment the COBOL programming language. To save space (and to aid those who are not COBOL programmers), the examples use English language descriptions for nondata-base functions such as input/output and computation. For those who know the COBOL programming language, the corresponding code should be obvious.

Our COBOL/DML program begins with the standard COBOL IDENTIFICATION and ENVIRONMENT DIVISIONS:

```
IDENTIFICATION DIVISION.
PROGRAM-NAME. SAMPLE-QUERY.
ENVIRONMENT DIVISION.
  identification of machine environment and
  declaration of non-data-base files
  (i.e., standard COBOL files)
```

The IDENTIFICATION and ENVIRONMENT DIVISIONS remain unchanged from those used by standard COBOL. Within the ENVIRONMENT DIVISION, we assign an internal COBOL file to the actual print file, for output of our query.

The COBOL DATA DIVISION incorporates the link to the data base:

```
DATA DIVISION.
FILE SECTION.
DB  PRES-ADMIN-STATE-INFO WITHIN PRESIDENTIAL.
FD  REPORT-FILE.
      remainder of data item entries which make
      up a standard COBOL file.
WORKING-STORAGE SECTION.
77  PRESIDENT-COUNT USAGE COMPUTATIONAL  PIC 999.
77  DONE  PIC 9(5) VALUE "04021".
77  NO-MORE-SONS  PIC A(5).
77  NO-MORE-STATES  PIC A(5).
```

The DB entry specifies which sub-schema and schema this program is referencing. While not required by the specifications, the effect of such a statement in most implementations is to cause the record descriptions from the sub-schema (augmented by schema information) to be copied into the COBOL application program, thus reserving space within the program for each data-base record type (and selected items) which this program may access. The record and data item names declared in the sub-schema are therefore referenceable from the COBOL program. DML verbs cause the DBMS to transfer data to/from the buffers reserved by copying the sub-schema record descriptions into the program.

The working storage section remains unchanged; here we define local variables to be used in the program. PRESIDENT-COUNT is used to keep a count of the native sons of a particular state. DONE is used as a mnemonic device for the status condition of a data-base. It is initialized to the correct status value. While using a DBTG-like system, it might be common practice to define a library of common status codes and to include them in the working storage section by using the COBOL COPY facility. NO-MORE-SONS and NO-MORE-STATES are status variables used within the program logic.

The procedures for accessing the data base

are specified in the COBOL PROCEDURE DIVISION:

```
PROCEDURE DIVISION.
DECLARATIVES.
EXPECTED-ERROR SECTION.
  USE FOR DATABASE-EXCEPTION ON "04021".
EXPECTED-ERROR-HANDLING.
  EXIT.
UNEXPECTED-ERROR SECTION.
  USE FOR DATABASE-EXCEPTION ON OTHER.
UNEXPECTED-ERROR-HANDLING.
  here we would process unexpected error
  conditions.
END DECLARATIVES.
```

The first part of the PROCEDURE DIVISION is the DECLARATIVES section. In the DECLARATIVES section we state that the processing is to take place when the DBMS determines that an error has occurred. The DBMS maintains a status location that can be referenced in the program by the name DATABASE-STATUS. DATABASE-STATUS, upon return from a DML command, will contain a value of "00000" if the command was successfully executed. A nonzero code represents an error condition, with the value for the code indicating the nature of the error.

In the event that an error code results from a DML operation, control is returned to that section within the DECLARATIVES that corresponds to the error code. For every error status code listed explicitly in a USE FOR DATABASE-EXCEPTION ON "error status code", control is passed to the paragraph that follows the USE statement. In the preceding example, if a DATABASE-STATUS of "04021" were returned, the paragraph labeled EXPECTED-ERROR-HANDLING would be invoked. Any DATABASE-STATUS codes that are not explicitly listed cause a transfer to the paragraph following the USE FOR DATABASE-EXCEPTION ON OTHER statement. In the preceding example, control would be returned to the paragraph UNEXPECTED-ERROR-HANDLING after an unlisted DATABASE-STATUS statement resulted.

As implied by the paragraph and section names in the preceding example, there is generally a distinction between error situations that we expect to happen as part of our normal processing, and error situations that are totally unexpected. An example of an expected error situation occurs when we sequentially traverse through a set occurrence and reach the end of the set occurrence. Such an error situation usually means that we should proceed to another part of our program to continue processing. An example of an unexpected error condition is an input/output error (for example, bad parity detected).

In this example the DATABASE-STATUS code of "04021" corresponds to the end-of-set occurrence condition just mentioned. When such an error occurs, the processing specifies a return to the statement following the DML command which caused the error (EXIT). The program would then examine DATABASE-STATUS and, if an end-of-set occurrence condition had occurred, could branch to another part of the program.

If any DATABASE-STATUS code other than "04021" occurs, control is passed to UNEXPECTED-ERROR-HANDLING. Here we would specify the processing to take place for these unexpected error situations. The code can examine DATABASE-STATUS as well as other status locations in an attempt to determine what caused the error and how the program should attempt to recover from it.

Following the DECLARATIVES section are the normal processing procedures:

```
INITIALIZATION.
  READY PRESIDENTIAL-AREA.
  OPEN non-database COBOL files.
  MOVE "FALSE" TO NO-MORE-STATES.
  FIND FIRST STATE IN ALL-STATES-SS.
  PERFORM PROCESS-STATE THRU FINISH-STATE
    UNTIL NO-MORE-STATES = "TRUE".
  GO TO FINISH-UP.
PROCESS-STATE.
  MOVE 0 TO PRESIDENT-COUNT.
  IF NATIVE-SON IS EMPTY
    MOVE "TRUE" TO NO-MORE-SONS,
    ELSE MOVE "FALSE" TO NO-MORE-SONS.
  PERFORM COUNT-NATIVE-SONS
    UNTIL NO-MORE-SONS = "TRUE".
  GO TO FINISH-STATE.
COUNT-NATIVE-SONS.
  FIND NEXT PRESIDENT IN NATIVE-SON.
  IF DATABASE-STATUS = DONE
    MOVE "TRUE" TO NO-MORE-SONS
    ELSE ADD 1 TO PRESIDENT-COUNT.
FINISH-STATE.
  IF PRESIDENT-COUNT IS GREATER THAN 1
    FIND STATE CURRENT,
    GET STATE,
    write out state name and president count.
  FIND NEXT STATE IN ALL-STATES-SS
  IF DATABASE-STATUS = DONE
    MOVE "TRUE" TO NO-MORE-STATES.
FINISH-UP.
  FINISH PRESIDENTIAL-AREA.
  CLOSE non-database COBOL files.
  STOP RUN.
```

The initialization of this program involves READYing the PRESIDENTIAL-AREA realm (area in the Schema language), which makes this realm available to the application program. Following this, any standard COBOL files are opened (for example, the report file). NO-MORE-STATES is used to indicate that we have finished processing all of the states, and is initialized to "FALSE".

Our algorithm used to solve this query is to traverse the STATE-record occurrences (by sequencing through the singular set ALL-STATES-SS). As we find each new STATE record, we check to see whether its NATIVE-SON set is empty (i.e., whether there are no native sons of that state), in which case we move to the next state. Otherwise, we traverse the occurrence of NATIVE-SON, counting the PRESIDENT records that participate in the set occurrence. If the count is equal to or greater than one, we write out the state name and number of native sons. If not, we continue by selecting the next STATE record until we have finished processing all states.

The FIND statement in the COBOL DML is used to locate a specified record occurrence in the data base. It does not cause the contents of the found record to be transferred to the working storage of the program. For example, the statement FIND FIRST STATE IN ALL-STATES-SS causes the system only to locate the record occurrence. The FIND statement changes the status of several *currency indicators* so that they point to the first record occurrence in ALL-STATES-SS. The member record which is "first" depends on the member-record order which was specified in the Schema description of the set.

By finding an occurrence of a STATE record, we have identified an occurrence of the NATIVE-SON record. We now PERFORM (execute) the COBOL paragraph labeled PROCESS-STATE through FINISH-STATE until the variable NO-MORE-STATES is set to the value of "TRUE", at which point we branch to FINISH-UP.

Within PROCESS-STATE we initialize PRESIDENT-COUNT to zero. If the current occurrence of NATIVE-SON is empty, we set NO-MORE-SONS to "TRUE"; otherwise, NO-MORE-SONS receives the value "FALSE". When the set is not empty, we PERFORM the paragraph COUNT-NATIVE-SONS until NO-MORE-SONS is set to "TRUE".

Within COUNT-NATIVE-SONS we FIND the NEXT occurrence of PRESIDENT. NEXT is relative to the current record occurrence of NATIVE-SON. If the current record of NATIVE-SON is the owner record (STATE), "next" is the first member record. When a PRESIDENT record is the current record of NATIVE-SON, then the "next" record is the PRESIDENT record which follows (unless the set is now exhausted).

If the program has traversed through all occurrences of PRESIDENT within the current occurrence of NATIVE-SON, the system sets DATABASE-STATUS to indicate this and passes control to the DECLARATIVES. Within the DECLARATIVES we have specified that for an end-of-set condition return is to be passed back to the statement after the FIND command. The program then sets NO-MORE-SONS to "TRUE", which causes the execution of COUNT-NATIVE-SONS to terminate.

If we have successfully found another occurrence of PRESIDENT, we increment PRESIDENT-COUNT and continue another iteration through COUNT-NATIVE-SONS.

The paragraph FINISH-STATE is entered when we have finished counting all of the native sons of the current STATE. We check PRESIDENT-COUNT to see whether its value is greater than one; if it is, we re-locate or again find the current STATE record (FIND STATE CURRENT) and GET it. The GET command causes the transfer of the record occurrence from secondary storage/system buffers to the internal work space of the program. After GETting the current STATE record, we write out the state name and the number of native sons.

We then proceed to select the next STATE record within ALL-STATES-SS. If we have finished processing all STATE records, NO-MORE-STATES receives the value "TRUE", which causes the termination of PERFORM PROCESS-STATE THRU FINISH-STATE. Otherwise, we continue with another iteration beginning at PROCESS-STATE.

When we have finished processing all of the STATE records, we enter the FINISH-UP paragraph; when we FINISH (release) the realm PRESIDENTIAL-AREA, CLOSE, and non-data-base (COBOL) files, we terminate the program.

Sample Update Program

While the Presidential data base is designed primarily for retrieval, its updating is still necessary, for example, at election time or when a new state is admitted to the Union. We now define a program to admit a new state. Referring to the discussion of selection criteria for the ADMITTED-DURING set, we recall that in order to enter a new occurrence of STATE in the data base, we must supply values for both LAST-NAME and FIRST-NAME in PRESIDENT, and for ADMIN-KEY in ADMINISTRATION.

In this program the IDENTIFICATION, ENVIRONMENT, and DATA DIVISIONs are basically the same as before, and therefore we specify only the PROCEDURE DIVISION:

```
PROCEDURE DIVISION.
DECLARATIVES.
UNEXPECTED-ERROR SECTION.
  USE FOR DATABASE-EXCEPTION.
UNEXPECTED-ERROR-HANDLING.
  here we process unexpected error conditions.
END DECLARATIVES.
INITIALIZATION.
  READY PRESIDENTIAL-AREA,
    USAGE-MODE IS UPDATE.
  OPEN COBOL files.
STORE-NEW-STATE.
  here we read in from standard COBOL input files
  the values for LAST-NAME and FIRST-NAME in PRESIDENT
  ADMIN-KEY in ADMINISTRATION, and values for the new
  STATE record.
  FIND ANY PRESIDENT.
  FIND ADMINISTRATION IN ADMINISTRATIONS-HEADED
    USING ADMIN-KEY.
  STORE STATE.
  CONNECT STATE TO ADMITTED-DURING.
FINISH-UP.
  FINISH PRESIDENTIAL-AREA.
  CLOSE COBOL files.
  STOP RUN.
```

Because STATE is a manual member of ADMITTED-DURING (see subsection Presidential Data Base in the DDL), we must explicitly tell the system which ADMINISTRATION record occurrence is to be the owner of the new STATE (in the ADMITTED-DURING set). We first locate the appropriate PRESIDENT. The FIND ANY PRESIDENT statement specifies that the system is to locate (using CALC) the occurrence of PRESIDENT based on its key value (LAST-NAME, FIRST-NAME). Then, within the occurrence of ADMINISTRATIONS-HEADED owned by the PRESIDENT thus selected, the system is to search for an occurrence of ADMINISTRATION with a value for ADMIN-KEY equal to that supplied to the program. We have now identified the necessary occurrence of ADMINISTRATION. When we request that the STATE information be STOREd in the data base, we complete the process by CONNECTing the newly stored State record to the current occurrence of ADMITTED-DURING. If the State had been an AUTOMATIC member of ADMITTED-DURING, the sequence of FIND statements would have been performed by the DBMS.

Traversing an m:n Relation in the COBOL DML

The relationship between CONGRESS and PRESIDENT is a many-to-many relationship (Section 1, subsection Many-to-Many-Relationships) and necessitated the introduction of a link record (CONGRESS-PRES-LINK). The introduction of such a link record complicates traversals from an occurrence of PRESIDENT to his CONGRESSes, and vice-versa. We present here a program segment designed to traverse such an $m:n$ relation.

We assume that LAST-NAME and FIRST-NAME in PRESIDENT have been set to a desired PRESIDENT in Dislay 2 below:

Display 2

```
    FIND ANY PRESIDENT.
FIND-NEXT-LINK.
    FIND NEXT CONGRESS-PRES-LINK IN CONGRESS-SERVED
    IF DATABASE-STATUS = DONE GO TO COMPLETE-RUN.
FIND-CONGRESS.
    FIND PRESIDENT-SERVED OWNER.
    here we have located a CONGRESS record occurrence over which the PRESIDENT
      served
    GO TO FIND-NEXT-LINK.
COMPLETE-RUN.
    continuation of program
```

This program segment is designed to FIND all CONGRESSes over which a specified PRESIDENT served. After FINDing the desired PRESIDENT, we traverse through each of the CONGRESS-PRES-LINK records owned by this PRESIDENT. When we reach a DATABASE-STATUS condition of DONE we have found all CONGRESSes over which this PRESIDENT served.

After locating a CONGRESS-PRES-LINK, we look for the owner of the record in the PRESIDENT-SERVED set. The owner of PRESIDENT-SERVED is the CONGRESS record. By FINDing each CONGRESS-PRES-LINK owned by a particular PRESIDENT within the CONGRESS-SERVED set and then by FINDing the owner of the link in the PRESIDENT-SERVED set, we can traverse from a PRESIDENT to all CONGRESSes over which he served. We can similarly traverse from a given CONGRESS record to all PRESIDENTs who served over that CONGRESS.

Other COBOL DML Facilities

The examples presented here illustrate some of the more important COBOL DML additions to the COBOL programming language. Primarily because of the static nature of the Presidential data base, it is difficult to create meaningful examples to illustrate several additional COBOL DML facilities. Several are briefly discussed here.

In addition to storing a new record occurrence in the data base, an existing record occurrence may need to have some of its values changed. The MODIFY verb is available for this. This operation (which is performed by the system) may be very complex because the effect of a key change may alter the position of the record or even affect the sets in which it resides (through set selection).

A record occurrence which exists in the data base may be deleted by use of the ERASE verb; once again, this may be a very complex operation, because the MANDATORY option may cause multiple deletions of many member records.

Finally, the second example showed the use of CONNECT to place a member record in a set. By use of the DISCONNECT verb an optional member record may be removed from a set occurrence. Note that DISCONNECT may only be used on optional member records.

3. ADVANCED FEATURES

Section 2, Sample Data-Base Applications introduced the basic facilities offered by the DBTG systems. The paper by Sibley and Fry [see page 7] pointed out, however, that there is more involved in the design and maintenance of a data base than the specification and manipulation of complex, interrelated data-base structures. In particular, any complete system must provide facilities that enable a data-base administration staff to write various utility routines for loading the data base, to check its validity, to collect statistics about record frequencies and clusterings, and to reallocate groups of record occurrences to improve performance. This list is by no means exhaustive. It should come as no surprise that the same logical structure can be realized on a computer in a variety of ways, each with varying efficiencies in different situations. Users occasionally need access to a "low" level of data (one close to bits on the storage media); there must be such a system interface.

An additional requirement in any system designed to serve a wide community of users concerns tuning and tailoring: means must be offered to allow various system services the option of either having their performance improved for a particular application mix, or having their services bypassed when such usage would lead to unacceptable performance. Much has already been said about data independence; certainly it is generally a major design goal. But data independence ultimately involves some execution time binding of decisions, and the extra computation involved may be intolerable. Yet it may still be desirable to allow certain programmers who knowingly sacrifice data independence to use the data-base system. The full recovery services of the system may be needed; its ability to manage multiple indexes may be needed; or

its ability to be used in a "customized" access method written in a procedure-oriented language may be needed.

The DBTG-like system architecture and language facilities makes such uses possible. The proposed facilities can be categorized in two parts. First, the DDL provides facilities for a data-base administrator to control various details of the storage strategies. For example, a user may declare that the system should create and maintain an index on specified items within a record type. The index would speed processing in appropriate situations. Examples of this and other DDL declarative facilities are given later in this section. In addition, the mechanism known as a data-base procedure allows the normal system facilities to be extended. Extremely detailed control can be attained, if desired, by the data-base administrator, and hence performance tuning is possible. Second, other facilities provide a set of data manipulation verbs (and a few more DDL options) which allow designated users to access data in a way that is less data independent. For example, there are facilities to obtain a data-base key (a logical "pointer" to a record occurrence) and subsequently to use this key to FIND a record. Examples of some uses of such a facility are included later. It is clear that use of these facilities should be controlled carefully, since data independence could suffer. A preprocessor or extended compiler could enforce such installation-defined standards.

The subsections that follow illustrate some of these advanced features. We augment the schema developed in Section 2, Sample Data-Base Application, and provide fragments of programs that use the advanced versions of various DML verbs.

Data-Base Procedures

The previous structural examples are largely fixed at the time schema definitions are made. For example, the record types, set types, and several of their attributes (e.g., AUTOMATIC) are all determined before data-base processing begins. One reason for this is, of course, that run-time interpretation generally reduces efficiency; hence record formats and accessing strategies are often fixed before processing begins. However, any flexible system provides a means whereby some parameters can be set (or certain extra processing can be triggered) during execution of a request. The DBTG system provides for this with data-base procedures.

A data-base procedure is logically a part of the data-base definition (the schema). It is a procedural augmentation of the (largely) declarative schema. The conditions under which a data-base procedure is to be invoked are given in the schema, either explicitly or implicitly. The procedure often operates as an ON condition functions in PL/I and may accomplish some change to the data-base state or control parameters. Data-base procedures can also be used to collect statistics and to enforce privacy by checking passwords. We use the concept of data-base procedures in many of the examples in the remaining subsections.

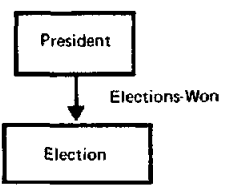
Data-base procedures are commonly used to derive data values. Frequently, certain data items are computable from other items in the data base. For example, the value of the total salaries paid to the people in a department is equal to the sum of the separate salaries of each of the employees of the department; or a person's age can be derived from his birth data and from today's date. Whenever such a functional relationship exists, the data-base administrator can provide a data-base procedure to compute the result. But two possibilities still remain. The functional value may be computed each time the record occurrence is retrieved by a program (VIRTUAL), or the functional value may be stored (ACTUAL) in the record occurrence and updated each time one of the values on which it depends is changed. Depending on the situation, performance can vary widely (e.g., where an ACTUAL result depends on time). The DBTG specifications provide the data-base administrator with the facilities for declaring items as either ACTUAL or VIRTUAL results of other items in the same record, or functions of items within selected members of sets owned by the given record. As an example, suppose we wished to include, in PRESI-

DENT, the maximum number of electoral votes ever obtained in any election of that president. This may be accomplished by including an item in PRESIDENT:

```
02 MAX-ELECT-VOTES; IS ACTUAL RESULT OF
     FIND-MAX-ELECT-VOTES
     ON MEMBERS OF ELECTIONS-WON.
```

where FIND-MAX-ELECT-VOTES is a data-base procedure name. ACTUAL was chosen here since updates should be infrequent. FIND-MAX-ELECT-VOTES would be called each time a new ELECTION record is added to the data base.

It is also possible to "propagate" items from an owner to a member record by using a SOURCE statement. This statement names the item within the owner-record type from which the propagated value should be drawn; once again, the data-base administrator can either save storage (virtual) or insure common copies of data values among the various records in the set (actual). For example, consider the structure:



where the data-base administrator has decided to include the name of the winning President as an item in the ELECTION record (one reason for doing this might be compatibility with a relational view of data). One way of implementing is:

```
RECORD NAME IS ELECTION
.
.
.
02 WINNING-PRESIDENT IS VIRTUAL AND
     SOURCE IS PRES-NAME
     OF OWNER OF ELECTIONS-WON.
```

where PRES-NAME is a qualified name of the item that will serve as the SOURCE item when ELECTION is fetched.

Data-base procedures provide a number of flexible facilities to control the behavior of the data base. Additional examples involving data-base procedures are given in the following subsections.

Areas

The various record occurrences that are the subject of the STORE command must, of course, be placed on actual secondary storage media. Naturally, if a data-base administrator is to have any influence in this placement strategy, there must be constructs in the DDL that allow this policy to be stated. Such control over physical placement seems to be necessary for reasonable performance, as demonstrated in many environments, including those not necessarily involving data bases. For example, Moler [G2] shows that numerical analysis programs which took advantage of the clustering of array values on the same page had significantly improved performance in a paged environment; for example, in FORTRAN implementations with column major order that scan by columns, not rows, when possible. The same phenomenon is as important in a data-base application.

The DBTG system provides basically two mechanisms for influencing record-occurrence placement: *areas* and *location mode* (which is treated in subsection Location Mode). "An area is a named subdivision of the data base" [S2, p. 2.23]. As was previously illustrated, each area is named in the schema, and there may be one or more areas declared. Many implementors have found it convenient to associate each area declared in the schema with a file (a catalogable entity) in the associated operating system. However, this need not be the case; hence our previous stress that the area is a logical rather than a physical concept.

Areas give the data-base administrator a mechanism for clustering or separating different record occurrences (possibly of diverse types). A given record occurrence of a given type may reside in only one area, but other occurrences of the same record type may reside in different areas, if desired. Also, occurrences of different record types can coexist in the same area. The area concept allows data-base designers flexibilities such as the following:

- If certain (or all) occurrences of a record type are known to be archival, they can reside in an area which is associated with a less expensive storage

medium. It may be that this area need not always be mounted.

- If records of diverse types are often used together, their occurrences can be clustered for high performance.
- By designating that all occurrences of a record type reside in one area, and by reserving that area for that record type, the effect of homogeneous files can be achieved.
- By omitting certain areas from a subschema, a measure of privacy is attained.
- Records may be processed consecutively within an area (using the FIND NEXT IN AREA verb), and processing can proceed at essentially sequential speeds, if the implementor allows areas to be allocated sequentially. This is because each "page" within an area will usually follow the previous one in terms of residing on the same cylinder on a disk. Such performance can be important, especially in utility and certain bulk "statistical" application programs in which record ordering is not important.
- An area may be used as a unit of recovery. It is then possible to vary the checkpoint policy for different portions of the data base. Some implementations may allow dual copies of designated areas (maintained by the system) both to reduce contention and to serve as added protection against catastrophe.

Since record occurrences are placed in areas, there must be a mechanism for specifying in which area a record occurrence of a given type must be stored. In Section 2, Sample Data-Base Application, we illustrated the constructs necessary to store all records in a single area. Since every record type had a unique area to contain it, no special action was needed. If, however, distinct occurrences of the same record type can potentially be stored in more than one area, the situation is more complex. For example, if the data-base administrator declares that STATE records may be stored in either of two areas by writing

```
RECORD STATE
WITHIN EASTERN-AREA, WESTERN-AREA
  AREA-ID IS STATE-AREA
```

then the variable STATE-AREA must be initialized with the proper alphanumeric area name at time of store by a command like:

```
MOVE 'EASTERN-AREA' TO STATE-AREA.
STORE STATE.
```

In this example, the applications programmer has control over area placement; the MOVE (COBOL) statement is used to initialize the AREA-ID variable. If the variable is left "null," then the area placement algorithm is left to the system implementor. Another option is to use a data-base procedure:

```
RECORD STATE
WITHIN EASTERN-AREA, WESTERN-AREA
  AREA-ID IS STATE-AREA
  USING PROCEDURE PICK-STATE-AREA
```

In this case, the procedure PICK-STATE-AREA is invoked to load the variable STATE-AREA, and the programmer need not know about the multiple areas.

Areas, then, give rather explicit control over major subdivisions of the data base and possibly their association with storage media. If this control is given to the applications programmer, there will probably be some loss in data independence. However, the decision concerning whether areas are or are not transparent to an application is in the hands of the data-base administrator.

Areas play an important role in data manipulation in the DBTG system. They are the basic unit (like files) which is OPENed and CLOSEd. As discussed in Section 2, Sample Data-Base Applications, areas are called REALMs in the COBOL subschema. The operations corresponding to OPEN and CLOSE are called READY and FINISH. The verb form

```
FIND NEXT record-name IN realm-name
```

provides a means whereby records can be scanned in ascending "physical" order within an area. For example, if the transaction that retires employees moves each retired employee record to a REALM called RETIRED-EMPLOYEES, then a "batch" program to scan sequentially for

all the recently retired employees would be as follows:

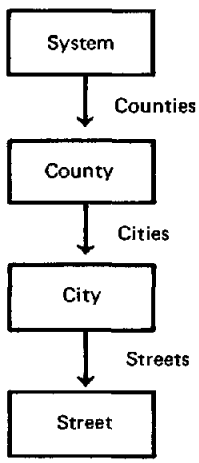
```
READY RETIRED-EMPLOYEES.
FIND FIRST EMPLOYEE IN RETIRED-EMPLOYEES.
check that at least one exists, then
PERFORM PROCESS-EMPLOYEES UNTIL (DATABASE-STATUS =
END-OF-REALM)
STOP RUN.
PROCESS-EMPLOYEES.
  Process an Employee record.
  FIND NEXT EMPLOYEE IN RETIRED-EMPLOYEES.
```

Location Mode

As illustrated in Section 2, Sample Data-Base Application, the DBTG-like systems allow a data-base administrator to have considerable control over the storage strategies and interrecord clusterings among records within a given area. This control is achieved through proper use of the LOCATION MODE clause, which appears in the declaration of each record type in the schema. It should be emphasized that the LOCATION MODE of a record type designates the strategy to be used for initial record placement when a new record occurrence is STOREd. Knowledge of how a record was initially stored can also be used in subsequently FINDing a record; but as the examples have shown, there are many ways of FINDing a record in the DBTG system. Thus, one should *not* associate LOCATION MODE with FINDing, but rather with "setting in a particular place" i.e., with the STORE command. As might be expected, there is a version of the FIND verb which depends on a knowledge of the location mode of the record being sought. Improper use of this form could, of course, lead to a loss in data independence.

There are four LOCATION MODEs defined. SYSTEM specifies that an implementor-defined algorithm be used in storing the record. Any area control specifications would, of course, be used by this algorithm.

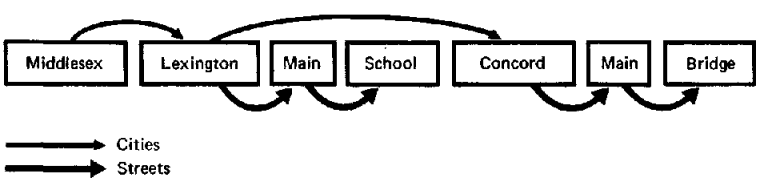
LOCATION MODE VIA set-name (where the record type is a set type member) specifies that the system place the new record occurrence as close as possible to its "appropriate" place in the set occurrence in which it (potentially) will become a member. This implies that the system will use the SET SELECTION clause of the appropriate set to find the proper owner-record occurrence. Having done so, the system will use the insert properties of the set (first, last, ordered, etc.) to place the new record. Using the LOCATION MODE VIA mechanism, it is possible to achieve a record clustering that is efficient for a "depth-first" search. For example, consider the following data structure diagram:



This diagram defines a hierarchical structure of a geographical region. By specifying the following statements in the record declaration section

```
RECORD COUNTY
LOCATION MODE IS SYSTEM
WITHIN GEO-AREA
    ⋮
RECORD CITY
LOCATION MODE IS VIA CITIES SET
WITHIN AREA OF OWNER
    ⋮
RECORD STREET
LOCATION MODE IS VIA STREETS SET
WITHIN AREA OF OWNER
    ⋮
```

and declaring the sets, it is possible to effect a storage structure:



Here, we introduce another option of the WITHIN clause. This option is meaningful only when the LOCATION MODE of the record is VIA some set-name. The area decision for a given record occurrence follows the area decision of the associated owner-record occurrence.

If a data-base administrator wishes more explicit control over record placement, then either of the two remaining location modes, CALC or DIRECT, may be appropriate. We illustrated, in Section 2, Sample Data-Base Application, how LOCATION MODE CALC may be used to place records according to an implementor-defined randomizing routine. The data-base administrator is also free to designate a data-base procedure that can serve as an algorithm for developing "addresses," based on the value of the identifier or identifiers. Whether such an algorithm attempts to behave pseudorandomly over the space of possible identifiers is, of course, determined by the algorithm developer (the data-base administrator). It is therefore possible to incorporate arbitrary record placement algorithms into the system.

It is important to understand the relationship between the location mode of a record type and the FIND verb. As illustrated in Section 2, there are many ways to FIND a record occurrence. These may be classified into two types. Either one FINDs a record occurrence based on its participation in a set (possibly a singular set), or one FINDs a record occurrence based on knowledge of how it was initially STOREd. These two methods provide a number of potential flexibilities. For example, by making every record type a member in a singular set and by having programmers FIND record occurrences using the form

```
FIND record-name IN singular-set-name USING identifier(s)
```

it is possible to provide access to all records by specifying their partial contents. A programmer need not know the location mode of a record type in order to use this verb. In addition, the data-base administrator can enhance performance in this case by defining indexes. The details are illustrated in subsection Search Keys. On the other hand, a knowledge of the method used in storing a record initially can provide excellent performance when applied again. For example, a search for a record with location mode VIA should follow the "clustering hierarchy" used in its initial storage. Similarly, knowledge of the identifier or identifiers used by the CALC routine can, in carefully designed applications, yield performance close to one secondary storage access per record retrieval [G3]. The form

```
FIND record-name IN set-name USING identifier(s)
```

can be used in a hierarchical search. This is illustrated by the example shown in Figure 16 which finds the first MAIN street in a city in MIDDLESEX county. This example differs from previous ones because there is a "don't care" condition on the CITY record occurrence.

As illustrated in Section 2, the specifications allow usage of the form

```
FIND ANY record-name
```

when a location mode of CALC has been declared for the sought record type. By properly initializing the identifier or identifiers which were used in initially storing the record, that record will be found. This procedure requires, of course, that programmers know which record types in the data base have location mode CALC and what item or items constitute their respective CALC-KEYS. It may therefore be difficult to change location modes (or the declaration of items forming the key) without affecting some program and hence introducing a lack of data independence. There have been suggestions [E4] for avoiding this potential problem by using a precompiler or data-base procedures.

The fourth location mode for a record

```
MOVE 'MIDDLESEX' TO COUNTY-NAME.
MOVE 'MAIN' TO STREET-NAME.
MOVE 'FALSE' TO SUCCESS.

FIND COUNTY IN COUNTIES USING COUNTY-NAME.
IF DATABASE-STATUS = NOT-FOUND
  Process for county not found

FIND FIRST CITY IN CITIES.
PERFORM SEARCH-FOR-STREET UNTIL
  (SUCCESS = 'TRUE') OR (DATABASE-STATUS = DONE)
IF SUCCESS = 'TRUE'
  Process for city found
  ELSE Process for city not found.

SEARCH-FOR-STREET.

FIND STREET IN CURRENT OF STREETS USING STREET-NAME.
IF DATABASE-STATUS = FOUND
  MOVE 'TRUE' TO SUCCESS
                    ELSE

  FIND NEXT CITY IN CITIES.
```

FIGURE 17. Hierarchical search.

type is DIRECT. In order to present this mode properly, we must introduce the notion of a data-base key. A *data-base key* is a unique identifier which is associated with every record occurrence in the data base. This data-base key is associated with the record occurrence when it is initially stored, and remains the unique identifier of the record occurrence throughout its lifetime in the data base. Although the specifications leave the detailed structure of a data-base key to be decided by the implementor, it is common for a data-base key to have some physical implications; data-base keys are often used in the implementation strategies of a given set. For set traversal to be efficient, the mechanism used must have some physical implications. For example, in many implementations, a data-base key will designate the area, the page within the area, and the record number within the page of the record occurrence. Areas (subdivisions of the data base) are often composed of fixed length pages, and it is easy to note the similarity to a segment/page addressing scheme in a virtual memory environment. However, in contrast with most virtual memory schemes, the actual placement of records within a page is often governed by a local "on-page" index, which maps the record number portion of the data-base key to a displacement within the page. In this way, space within a page can be garbage collected, and records, whose size can in general vary with time, can be moved within the page, all without disturbing pointers pointing at the object being moved. Of course, if a record grows to the extent that it must be moved to an overflow page, then a small "overflow pointer" must remain on the original page. As long as the number of records on overflow pages is not a great fraction of the total records, this scheme can behave almost like direct address pointers while still allowing record movement. If too many records are moved to overflow pages, then the corresponding area can be expanded, and either the page size or the number of pages can be increased.

We now return to the discussion of LOCATION MODE DIRECT. In the DIRECT mode, the program which STOREs occurrences of the given record type may specify the data-base key which the system will try to use. This is in contrast to other location modes where the data-base system determines the data-base key based on calculation keys or the proximity to a logical insertion point in a set. Assuming the data-base key is not already used, the system will use the program-designated key. If that data-base key is already used, the system chooses the next (higher) unused data-base key. With care, the program may have a great deal of control over record placement.

Clearly, DIRECT location mode may be much closer to the physical levels of data. If a high level of data independence is a goal, then the usage of this location mode must be carefully controled. However, such a facility can be quite useful, especially when writing data-base maintenance procedures. As an illustration, consider the method of loading the data base. One popular technique is to let the system, during loading, operate under a schema that is different from the run-schema. In this load-schema, the location mode of the records is DIRECT. If the run-schema specifies a location mode of CALC, the load program can then use the following algorithm:

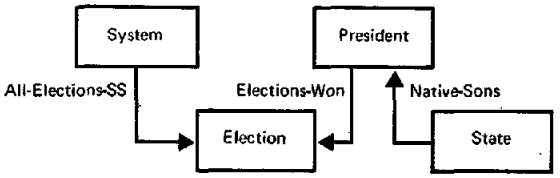
1) For each record to be loaded, invoke a local copy of the CALC subroutine to compute a data-base key using the designated CALC-key items within the record.
2) Sort all records by ascending computed data-base key.
3) Use the load-schema (which has location mode DIRECT) to store the records in one sequential pass over the area.

So-called batch-random operation [G4] in a data-base load program has been found to improve performance by as much as four times. Location mode DIRECT has allowed such a program to be written in a high-level language.

Location mode DIRECT may also be used when direct accessing of records is desired or when building specialized access methods on top of the basic system facilities.

Search Keys

Consider the data structure diagram



Let us suppose that the data-base administrator has determined that the dominant usage of this portion of the data base is to answer the question: Print the elections won by President X. In this case, it would be appropriate to give ELECTION records a LOCATION MODE VIA ELECTIONS-WON set in order to cluster them close to the related President. On the other hand, if we consider the question: Find the state whose native son was the winner of the election of date X, it is clear that the speed of response depends primarily on how fast the election record of a given date can be found (after that, the answer involves two FIND OWNER statements). The question may then be posed:

```
FIND ELECTION VIA ALL-ELECTION-SS USING ELECTION-YEAR.
```

Since an exhaustive search of all ELECTION record occurrences very likely would be slow, it may be appropriate, if query volume is sufficient, to define an index on occurrences of the ELECTION record type. Indexes in the DBTG-like systems are specified by using the SEARCH-KEY clause. By declaring:

```
SET NAME IS ALL-ELECTIONS-SS
  OWNER IS SYSTEM . . .
  MEMBER IS ELECTION
    MANDATORY AUTOMATIC . . .
  SEARCH KEY IS ELECTION-YEAR USING INDEX
    DUPLICATES ARE NOT ALLOWED
```

the system builds and maintains an index on the ELECTION-YEAR item. When the FIND statement is issued, the system uses this index to speed the search; it uses the ELECTION-YEAR item provided by the program to search the index to find the first (and in this case only) record having the given year. This ELECTION record is then accessed. The success of the FIND operation does not depend on the continued existence of the index; indexes can be added or deleted as appropriate.

In general, use of the SEARCH KEY clause implies that the DBMS will build an index on the member records (of a given type) within a set occurrence. Its use implies the existence of many indexes, one for each set occurrence. The use in a singular set is a special case.

Introduction of the SEARCH KEY clause allows simulation of the traditional indexed sequential file. By declaring:

```
SET NAME IS ALL-STATES
  OWNER IS SYSTEM
  ORDER IS PERMANENT INSERTION IS
    SORTED BY DEFINED KEYS
  MEMBER IS STATE
    MANDATORY AUTOMATIC
  KEY IS ASCENDING STATE-NAME
    DUPLICATES ARE NOT ALLOWED
    NULL IS NOT ALLOWED
  SEARCH KEY IS STATE-NAME USING INDEX
    DUPLICATES ARE NOT ALLOWED
```

the data-base administrator can: 1) specify an ability to scan sequentially through all record occurrences of a given type; plus 2) provide a direct path (through an index) to a particular occurrence, given a value of its primary key. To obtain a performance similar to the traditional indexed sequential file, the data-base administrator would have only one record type in the associated area. Thus logically adjacent records would tend to be physically adjacent, as in an indexed sequential file.

SEARCH KEYs can also be used to support an arbitrary number of inversions (secondary indexes) over the members of a set occurrence. The data-base administrator has the option of allowing or disallowing duplicates for items or concatenations of items among members of the set occurrence. Thus, in a singular set, there is a mechanism for guaranteeing unique values across all occurrences of a given record type.

Set Selection

As explained in Section 2, Sample Data-Base Application, each declared set in the

schema has an associated set (occurrence) selection clause. The basic purpose of this clause is to inform the system how to select a particular set occurrence of the particular set type. This is necessary, for example, when a record type is an AUTOMATIC member in a set type. As previously illustrated, when a new record of this type is stored, the system must automatically include it in a set occurrence; the set selection clause tells which is the appropriate one.

The other major use of set selection is in the FIND verb:

```
FIND record-name IN set-name USING id-1, id-2, ...
```

When this version of the FIND verb is executed, the set selection clause of the named set is invoked to find the set occurrence. Searching for a record of the designated type within the set occurrence then proceeds. If the optional USING clause is omitted, the first such record is selected. Otherwise, the system selects the first member of the set where all identifiers (in the record) are equal to the initialized identifiers. If no such record exists, the search fails and control is returned to the DECLARATIVES section of the program.

The set selection clause, while defined in the schema, may be changed or augmented in the subschema. That is, the particular set occurrence selection can vary from one subschema to the next, under control of the data-base administrator. The set selection clause in the schema is a default, which will be used unless overridden by the subschema.

As an illustration of these concepts, we search for the first MAIN street in LEXINGTON in MIDDLESEX county (see subsection Location Mode). The relevant piece of program code is:

```
MOVE 'MIDDLESEX' TO COUNTY.
MOVE 'LEXINGTON' TO TOWN.
MOVE 'MAIN' TO STREET-NAME.
FIND STREET IN STREETS USING STREET-NAME.
```

This contains considerably less code than the previous example (though the example is slightly different). Clearly, the extra searching is performed by the data-base system with only one FIND command. The specifications for doing this are declared in the associated subschema, as:

```
SD STREETS
  SET SELECTION FOR STREET IS
    VIA COUNTIES  OWNER SYSTEM
    VIA CITIES    OWNER VALUE OF
                    COUNTY-NAME IS COUNTY
    VIA STREETS   OWNER VALUE OF
                    TOWN-NAME IS TOWN.
```

In the set selection specification, the system is instructed to first locate a COUNTY by going to the singular set COUNTIES and to search forward until a matching county name (i.e., MIDDLESEX) is found; then the system is instructed to descend into the CITIES set to search for a matching town; finally the system is instructed to search for a matching street by the USING phrase on the FIND verb.

It should be clear that a set selection declaration is basically a specification for a data-base procedure that performs an effectively hierarchical search in order to establish a set occurrence. An entry point is established, either through singular sets, CALC entry points, or currency (see subsection Currency Indicators); then, if this is not the desired set occurrence, a hierarchical traversal can be carried out starting at this point until the proper set occurence is found.

One can also note that the only "match arguments" currently allowed are item equalities or concatenations of item equalities. The "don't care" condition and potential for backtracking, as expressed in subsection Location Mode, is not possible. This greatly eases the implementation, as opposed to making tree traversals against arbitrary Boolean expressions. To achieve the same effect produced by the example given in subsection Location Mode, a data-base procedure would be written. The system could then be told to use the procedure by the statement:

```
SET SELECTION IS BY PROCEDURE FIND-ANY-MAIN-STREET.
```

Support for the more complex traversals can be defined by installation using data-base procedures, though of course there is only one possible data-base procedure per set declaration per subschema. If the traversal is more specific to applications or transactions, it must be specially programmed.

Currency Indicators

The notion of a current record was discussed briefly in Section 2, Sample Data-Base Applications, but our examples have not emphasized currency. Normally, the system behaves in an expected way and the programmer does not need to take special note of currency. There are certain complex traversals, however, where the application programmer must be aware of the exact status of the currency indicators. In this section, we discuss currency and show when care must be exercised.

There are many currency indicators associated with a typical program that executes with a subschema (this is called a run-unit):

- one currency indicator designating the current record referenced by the run-unit;
- one currency indicator for each record type, indicating the current record of that type;
- one currency indicator for each set type, indicating the current set occurrence of that type by "pointing" at the referenced record occurrence, which is either the owner or a member of the given set; and
- one currency indicator for each realm, indicating the current record referenced in that realm.

Thus in the Presidential data base, if a program can access the entire data base, the system would maintain seventeen currency indicators—six for the six record types, nine for the nine sets (including each singular set), one for the single area, and one for the current record of the run-unit.

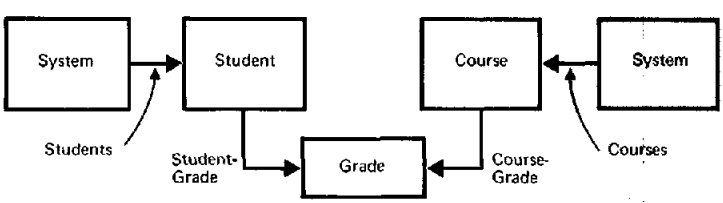
The currency indicators always make the concept of "next" (and prior) well defined, regardless of how the currency has been established; for example, next within a realm means the record in that realm with the next higher data-base key, and next within a set means the next record in the "forward" direction in that set.

It is important to understand when currency indicators change. Currency indicators always change on execution of some DML verb. Also, the object of certain actions must be the current record of the run-unit. The following rules apply:

- Only the current record of the run-unit may be the subject of the GET, ERASE, CONNECT, and DISCONNECT verbs.
- When a new record of any type is found or stored (except for special exceptions discussed later), it becomes the current record of the run-unit and realm in which it resides; it also becomes the current record of its type and of all sets in which it participates as either owner or member. Thus the currency of many sets may be affected.

Because the currency indicators of sets can potentially change when a new record of a given type is found, a programmer must be careful when doing traversals that involve "backtracking." By backtracking, we mean a situation which (for some reason) makes it necessary that a previously established position be reestablished. If, in the meantime, a new record has been stored, the set currency indicators may have potentially changed, making the reestablishment of position difficult or impossible. The programmer must anticipate such changes to the currency indicators and take steps to avoid such situations. The following example illustrates that, even during a pure retrieval, the currency indicators must be handled properly. The example has been adapted from Date [G5].

Given the STUDENT/COURSE data structure diagram below



answer the following question: For each student taking MATH, find all the other courses being taken by that student and print the student's name and the names of the other courses. Note the situation here. Given that we have found a student who takes MATH (found by locating the MATH record and by performing the "switch and find owner" traversal as illustrated in Section 2), we must reenter the same structure to find the courses which are not MATH being taken by that student and print them. But this retraversal of the STUDENT/COURSE structure will destroy the previous currency associated with the COURSE/GRADES set. When we try to find another student taking MATH, the results are potentially unpredictable.

To allow a currency saving facility, two approaches are provided. One is the ACCEPT statement.

$$\text{ACCEPT identifier FROM } \begin{Bmatrix} \text{realm-name} \\ \text{set-name} \\ \text{record-name} \end{Bmatrix} \text{ CURRENCY}$$

moves the designated currency indicator to a run-unit variable. Subsequent use of the FIND verb

FIND record-name; DATABASE-KEY IS identifier

would restore the relevant currency indicator.

Another method has also been provided. Both FIND and STORE have an optional RETAINING CURRENCY clause. By using this clause, the currency for designated record types, set types, or realms is not changed by the execution of the verb; that is, the normal currency updates, as explained previously, are not performed (except for the current record of the run-unit). As an illustration of this, the program shown in Figure 18 answers the STUDENT/COURSE question already discussed. We assume course name is (at least virtually) part of the GRADE record. The reader will note that a GO TO statement is used, since it is felt that a simpler program results.

Two other examples illustrating when special treatment of currency indicators is necessary are the parts explosion traversal (see section 1, Complex Relationships

```
    MOVE 'MATH' TO COURSE-NAME.
    FIND COURSE IN COURSES USING COURSE-NAME.
**  We now have the Math record. We assume it exists.
    FIND FIRST GRADE IN COURSE-GRADE.
    PERFORM PRINT-STUDENTS-OTHER-COURSES UNTIL
      (DATABASE-STATUS = DONE).
    STOP RUN.
  PRINT-STUDENTS-OTHER-COURSES.
    PERFORM FETCH-AND-PRINT-ONE-STUDENT.
    PERFORM PRINT-THAT-STUDENTS-COURSES.
**  Now check for more students by trying to
**  FIND more grades associated with the given course
**  (Math in our example).
    FIND NEXT GRADE IN COURSE-GRADE.
  END-OTHER-COURSES.
  FETCH-AND-PRINT-ONE-STUDENT.
    FIND STUDENT OWNER.
    GET NAME OF STUDENT.
    Print it.
  END-ONE-STUDENT.
  PRINT-THAT-STUDENTS-COURSES.
**  Note the use of the RETAINING CLAUSE Below
    FIND NEXT GRADE IN STUDENT GRADE
      RETAINING CURRENCY FOR COURSE-GRADE.
    IF DATABASE-STATUS ≠ DONE
      GET GRADE
      IF COURSE IN GRADE ≠ COURSE-NAME
        Print the other course name.
      GO TO PRINT-THAT-STUDENTS-COURSES.
  END-STUDENTS-COURSES.
```

FIGURE 18. Illustration of currency retention.

Using Data Structure Diagrams) and the relatively uncommon question: Find the employees who earn more than their managers. The reader is encouraged to think through these two examples with respect to currency indicators.

4. IMPLEMENTATIONS OF THE DBTG SPECIFICATIONS

As mentioned in the Introduction, the specifications initially published by the DBTG are under continuing development and refinement by groups within the CODASYL organization. It is difficult to state whether system "X" does, or does not, follow the specifications. There are, however, a number of commercially available systems that have used one or more versions of the specifications as a basis for implementation. While these system may employ syntax that is slightly different from our examples, they follow the same basic data model. Some of the commercially available systems which are generally deemed to be "DBTG type" systems are:

- DBMS/10 (Data Base Management System/10), marketed by Digital Equipment Corp. for use on DEC System 10 computers
- DMS/1100 (Data Management System/1100), marketed by UNIVAC

for use on UNIVAC Series 1100 computers
- EDMS (Extended Data Management System), marketed by XEROX Data Systems for use on XEROX SIGMA 6, 7 and 9 computers
- IDMS (Integrated Data Management System), marketed by Cullinane Corp. for use of IBM System/360 and System/370 computers
- IDS/II (Integrated Data Store/II), marketed by HONEYWELL Information Systems for use of the HONEYWELL 6000 series computers
- PHOLAS, marketed by PHILIPS ELECTROLOGICA.

Though this section is not meant to be detailed, it is worthwhile to consider what parts of the specifications are successfully implemented, for such an examination would indicate which features the implementors have found easy (or difficult) to implement, or which features the implementors thought would be of use to their customers.

As a general rule, the implementors have allowed full generality to be used in the data model presented in Section 1, Design of a Data Base, as well as in most of the DML functions presented in Section 2, Sample Data-Base Application, and Section 3, Advanced Features. The part of the data model most frequently omitted is that of singular sets. The rationale for this seems to be that singular sets can be very easily simulated by the user.

While the basic data model is maintained in all of the systems, various implementors have left out many of the more sophisticated features in the Schema DDL. The facility for privacy locks and keys is most frequently dropped, as are data-base procedures. In addition, none of the available systems provide for VIRTUAL items.

In approaches to the sub-schema facility, systems vary widely. The initial implementation of some systems did not provide for a separate sub-schema language (indeed, in 1969 the DBTG specifications did not include one); instead, the systems relied on the COBOL program. Such a facility provides for inclusion or exclusion of certain schema record types from the program. Even in cases where a separate sub-schema language is provided, many implementors have still allowed only for the inclusion or exclusion of entire record types or set types. In a minority of the systems the sub-schema is allowed to select only certain data items from a record, or to reorder or change the data-item type.

GUIDE TO FURTHER READING

The DBTG class of systems will continue to evolve, as will the state of various implementations. In an introductory paper such as this, it is impossible to cover all the options and characteristics of these DBTG-like systems. For these two reasons, it is necessary to read further in the literature for an in-depth understanding of the total system architecture, data model concepts, and data-base design techniques. This section presents an annotated guide to DBTG literature. We concentrate here only on literature whose principal focus is the DBTG specifications, or literature which compares, in-depth, the DBTG approach to some alternative. Discussions of data-base management in general, or tutorial treatments, or references to specific vendor manuals appear in the general bibliography in the companion paper in this issue by Fry and Sibley.

The most recent developments with respect to the Schema language appear in the CODASYL Data Description Language Journal of Development [S2]. Specifications for the Data Manipulation Language of COBOL appear in the CODASYL COBOL Journal of Development [S3]. These documents are issued periodically, and announcements of availability appear in various professional journals. There is also a CODASYL Committee that is developing Data Manipulation Language specifications for FORTRAN [S5, see also S6]. All three references are primarily language specification manuals; as such they are not written in tutorial fashion, but are intended primarily for implementors and as a final arbiter regarding details of program/data-base system semantics. However, [S2] does contain sections that describe concepts of the Schema language.

On a more general level, a discussion of the evolution of "navigational" systems, of which the DBTG systems are a prime example, is given in the ACM Turing Lecture by Charles W. Bachman [N3]. In [N2], Bachman describes how data structure diagram notation can be used to illustrate the organization of lower levels of data—specifically illustrating the access method and storage medium levels.

A collection of more advanced examples, based on the 1971 DBTG report has been published by Frank and Sibley [E1]. Another example by Sibley, using the 1973 syntax, is available as a National Bureau of Standards report [E3]. Additional examples appear in vendor manuals, especially [E2].

There has been a continuing debate concerning the merits and disadvantages of the DBTG architecture. Aspects of this debate are covered in the companion paper by Michaels, Mittman, and Carlson in this issue. Comparisons of the DBTG proposal relative to the relational model appear in many places; one of the most complete discussions is contained in the proceedings of a debate [D1, D2] in which C. W. Bachman and E. F. Codd are the principals. There have also been critiques and technical evaluations of various aspects of DBTG. One such critique [D5] was presented when the 1971 report was published. In 1975, an IFIP Working Conference was devoted specifically to an in-depth evaluation of various constructs in the Schema language. Proceedings of that conference have been published, and various revisions to the DDL are proposed [D8, R1-R5]. The volume also contains articles that illustrate how to use DBTG systems to support a relational view and discuss the use of concepts from the relational model (e.g., normalization) in the context of DBTG systems. Papers on the proper design of data bases by using data structure diagrams and on the proper use of the Data Manipulation Language appear in [A1-A7].

There have been discussions of the possibility of designing systems which could support any data model a user might wish—whether network, relational, hierarchic, or other types. Nijssen's article "Data Structuring in the DDL and Relational Data Model" [C1] outlines the possibility of the coexistence of data models. The article "On the Equivalences of Data Based Systems" by Sibley [C4] also explores this point.

A SHARE Working Conference held in Montreal, Canada, contains papers describing user experiences with various commercially available implementations of the DBTG systems [U3-U5].

Some aspects of implementation are discussed in [I1–I5].

There have also been a number of papers dealing with the features required by data-base management systems. References [M4–M6, M8] are of particular interest.

CLASSIFICATION OF REFERENCES

S Syntax and System Specifications
N Data Structure Diagram Notation, Navigational Systems
E Example Schemas, Sub-schemas, and Programs
D Critiques and Debate Position Papers
R Suggested Revisions to the Specifications
C Comparison of the DBTG Model to Other Data Models
A Designing Data Bases Using DBTG Systems
U User Experience with Commercial Implementations
I Implementations
M Other Modeling Papers
G Referenced Papers

REFERENCES

This bibliography collects and classifies references to various articles concerning the DBTG specifications. It also includes articles which debate the merits of various features, articles which discuss implementational aspects, and articles which discuss data-base design in the context of a DBTG system. The following abbreviations are used in the bibliography.

SIGMOD/SIGFIDET — The ACM Special Interest Group on the Management of Data (formerly named the Special Interest Group on File Description and Translation) holds an annual Conference. The Proceedings of these conferences are available from ACM, New York.

IFIP TC-2 — A Special Working Conference, "An In-Depth Evaluation of Codasyl DDL," was held in Belgium in January 1975. Proceedings of that conference are available in the book *Database Description*, B.C.M. Douque

and G. M. Nijssen, Eds. North-Holland Publ. Co., Amsterdam, The Netherlands, 1975.

(S) Syntax and System Specifications

[S1] CODASYL DATA BASE TASK GROUP, *April 1971 report*, ACM, New York.

[S2] CODASYL DATA DESCRIPTION LANGUAGE COMMITTEE, *Data Description Language Journal of Development*, Document C13.6/2:113, U.S. Government Printing Office, Washington, D.C.

[S3] CODASYL PROGRAMMING LANGUAGE COMMITTEE, *CODASYL COBOL Journal of Development*, Dept. of Supply and Services, Government of Canada, Technical Services Branch, Ottawa, Ontario, Canada.

[S4] CODASYL DATABASE LANGUAGE TASK GROUP, *CODASYL COBOL database facility proposal*, 1973, Dept. of Supply and Services, Government of Canada, Technical Services Branch, Ottawa, Canada. This document proposes revisions to CODASYL COBOL. The revisions, as accepted, appear in the latest version of the *CODASYL COBOL Journal of Development*, see [S3].

[S5] *CODASYL FORTRAN DML*, information on the activities of this committee is available from Chairman, CODASYL FORTRAN DML Committee, P.O. Box 124, 928 Garden City Drive, Monroeville, Pa., 15146.

[S6] STACEY, G. M., "A FORTRAN interface to the CODASYL Data Base Task Group specifications," *Computer J.* **17**, 2 (May 1974), 124–129.

(N) Data Structure Diagram Notation, Navigational Systems

[N1] BACHMAN, C. W., "Data structure diagrams," *Database* **1**, 2 (Summer 1969).

[N2] BACHMAN, C. W., "The evolution of storage structures," *Comm. ACM* **15**, 7 (July 1972), 628–634.

[N3] BACHMAN, C. W., "The programmer as navigator," *Comm. ACM* **16**, 11 (Nov. 1973), 653–658.

(E) Example Schemas, Subschemas, and Programs

[E1] FRANK, R. L.; AND SIBLEY, E. H., *The Data Base Task Group Report: an illustrative example*, Doc. No. AD-759-267, U.S. National Technical Information Service.

[E2] PHILIPS DATA SYSTEMS, *An application example of the CODASYL DBTG Proposal*, Pub. No. 5122-991-24151, Philips-Electrologica, Apeldoorn, The Netherlands, 1973.

[E3] SIBLEY, E. H., *The CODASYL database approach: a COBOL example of design and use of a personnel file*, NBSIR 74-500, Institute of Computer Sciences and Technology, National Bureau of Standards, Washington, D.C.

[E4] TAYLOR, R. W., "Data administration and the DBTG Report," *Proc. of ACM SIGMOD/SIGFIDET Conf.*, 1974, ACM, New York, pp. 431–444.

[E5] MANOLA, F. A., *Principles of the CODASYL approach to the description of data structures*, Report 3068, Naval Research Laboratory, 1975, Washington, D.C.

(D) Critiques and Debate Position Papers

[D1] BACHMAN, C. W., "The data structure set model," in *Proc. 1974 ACM-SIGMOD Debate, "Data Models: Data Structure Set versus Relational,"* R. Rustin, (Ed.), ACM, New York, pp. 1–10.

[D2] CODD, E. F.; AND DATE, C. J. "Interactive support for non-programmers: the relational and network approaches," in *Proc. 1974 ACM-SIGMOD Debate, "Data Models: Data Structure Set versus Relational,"* R. Rustin, (Ed.), ACM, New York, pp. 13–41.

[D3] DATE, C. J.; AND CODD, E. F., "The relational and network approaches: comparison of the application programming interfaces," in *Proc. 1974 ACM-SIGMOD Debate "Data Models: Data Structure Set versus Relational,"* R. Rustin (Ed.), ACM, New York, pp. 85–113.

[D4] EARNEST, C. P., *A comparison of the network and relational data structure models*, Technical Report Sciences Corp., El Segundo, California, 1974.

[D5] ENGELS, R. W., "An analysis of the April 1971 DBTG report," in *Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control*, ACM, New York, pp. 69–91.

[D6] METAXIDES, A., "Information bearing and non-information bearing sets," in *Proc. of IFIP TC-2 Special Working Conf., "An In-depth Technical Evaluation of the CODASYL DDL,"* pp. 363–368.

[D7] OLLE, T. W., "Current and future trends in database management systems," in *Proc. IFIP Congress, Information Processing 74*, North-Holland Publ. Co., Amsterdam, The Netherlands, pp. 998–1006.

[D8] WAGHORN, W. J., "The DDL as an industry standard?," in *Proc. IFIP TC-2 Special Working Conf., "An In-depth Technical Evaluation of the CODASYL DDL,"* pp. 121–167.

(R) Suggested Revisions to the Specifications

[R1] KAY, M. H., "An assessment of the CODASYL DDL for use with a relational sub-schema," in *Proc. IFIP TC-2 Special Working Conf., "An In-depth Technical Evaluation of CODASYL DDL,"* pp. 199–214.

[R2] NIJSSEN, G. M., "Set and CODASYL set or coset," in *Proc. IFIP TC-2 Special Working Conf., "An In-depth Technical Evaluation of CODASYL DDL,"* pp. 1–71.

[R3] OLLE, T. W., "An analysis of the flaws

in the Schema DDL and proposed improvements," in *Proc. IFIP TC-2 Special Working Conf.*, "*An In-depth Technical Evaluation of CODASYL DDL*," pp. 283–297.

[R4] Robinson, K. A., "An analysis of the uses of the CODASYL set concept," in *Proc. IFIP TC-2 Special Working Conf.*, "*An In-depth Technical Evaluation of CODASYL DDL*," pp. 169–181.

[R5] Taylor, R. W., "Observations on the attributes of database sets," in *Proc. IFIP TC-2 Special Working Conf.*, "*An In-depth Technical Evaluation of CODASYL DDL*," pp. 73–84.

[R6] Hawley, D. A.; Knowles, J. S.; and Tozer, E. E., "Database consistency and the CODASYL DBTG proposals," *Computer J.* **18**, 3 (1975), 206–212.

(C) Comparison of the DBTG Model to Other Data Models

[C1] Nijssen, G. M., "Data structuring in the DDL and relational model," in *Database Management*, J. W. Klimbie, and K. L. Koffeman, (Eds.), North-Holland Publ. Co., Amsterdam, The Netherlands, 1974, pp. 363–384.

[C2] McGee, W. C., "A contribution to the study of data equivalence," in *Database Management*, J. W. Klimbie, and K. L. Koffeman, (Eds.), North-Holland Publ. Co., Amsterdam, The Netherlands, 1974, pp. 123–148.

[C3] Senko, M. E., "Data description language in the context of a multi-level structured description: Diam II with Foral," in *Proc. IFIP TC-2 Special Working Conf.*, "*An In-depth Technical Evaluation of CODASYL DDL*," pp. 239–257.

[C4] Sibley, E. H., "On the equivalences of databased systems," in *Proc. 1974 ACM-SIGMOD Debate*, "*Data Models: Data Structure Set versus Relational*," R. Rustin, (Ed.), ACM, New York, pp. 45–76.

[C5] Stonebraker, M.; and Held, G., "Networks, hierarchies, and relations in database management systems," in *Proc. ACM Pacific 75 Regional Conf.*, ACM, New York, pp. 1–9.

(A) Designing Data Bases Using DBTG Systems

[A1] Bachman, C. W., "Implementation techniques for data structure sets," in *Database Management Systems*, D. A. Jardine, (Ed.), North-Holland Publ. Co., Amsterdam, The Netherlands, 1974, pp. 147–257.

[A2] Baker, G. J., "The correct use of CODASYL DBTG sets," *Database Journal* **6**, 2, pp. 19–21. A. P. Publications Ltd., London.

[A3] Brown, A. P. G., "Modeling a real world system and designing a schema to represent it," in *Proc. IFIP TC-2 Special Working Conf.*, "*An In-depth Technical Evaluation of CODASYL DDL*," pp. 339–347.

[A4] Bubenko, J. A., Jr., et al., "From information structures to DBTG data structures," in *Proc. Conf., on Data: Abstraction, Definition, and Structure, ACM SIGPLAN/SIGMOD* 1976, ACM, New York, pp. 73–84.

[A5] Gerritsen, R., "A preliminary system for the design of DBTG data structures," *Comm. ACM*, **18**, 10 (Oct. 1975), 551–557.

[A6] Mitoma, M. F.; and Irani, K. B., "Automatic database schema design and optimization," in *Proc. of the Internatl. Conf. on Very Large Databases*, 1975, ACM, New York, pp. 286–321.

[A7] Taylor, R. W., "When are pointer arrays better than chains," in *Proc. ACM National Conf.*, 1974, ACM, New York, p. 735.

(U) User Experience With Commercial Implementations

[U1] Bandurski, A. E.; and Jefferson, D. K., "Data description for computer-aided design," in *Proc. ACM SIGMOD Internatl. Conf. on Management of Data*, 1975, W. F. King, (Ed.), ACM, New York, pp. 193–202.

[U2] Canning, R. G., "What's happening with CODASYL-type DBMS," *EDP Analyzer*, (Oct. 1974).

[U3] Emerson, E. J., "DMS 1100 user experience," in *Database Management Systems*, D. A. Jardine, (Ed.), North-Holland Publ. Co., Amsterdam, The Netherlands, 1974, pp. 35–46.

[U4] Lavallee, P. A.; and Ohayon, S., "DMS applications and experience," in *Database Management Systems*, D. A. Jardine, (Ed.), North-Holland, Publ. Co., Amsterdam, The Netherlands, 1974, pp. 47–67.

[U5] Von Gohren, G. L., "User experience with integrated data store (IDS)," in *Database Management Systems*, D. A. Jardine, (Ed.), North-Holland Publ. Co., Amsterdam, The Netherlands, 1974, pp. 19–33.

(I) Implementations

[I1] Bachman, C. W., and Williams, S. B., "A general purpose programming system for random access memories," in *Proc. AFIPS 1964 Fall Jt. Computer Conf.*, Vol. 26, Spartan Books, Baltimore, Maryland, pp. 411–422.

[I2] Canaday, R. H., et al., "A back-end computer for database management," *Comm. ACM* **17**, 10 (Oct. 1974), 575–582.

[I3] Fossum, B. M., "Database integrity as provided for by a particular database management system," in *Database Management*, J. W. Klimbie and K. L. Koffeman, (Eds.), North-Holland Publ. Co., Amsterdam, The Netherlands, 1974, pp. 271–288.

[I4] Johnson, H. R., "A schema report facility for a CODASYL based data definition language," in *Database Description*, B. C. M. Douque and G. M. Nijssen, (Eds.), North-Holland Publ. Co., Amsterdam, The Netherlands, 1975, pp. 299–328.

[I5] SCHENK, H., "Implementational aspects of the CODASYL DBTG proposal," in *Database Management*, J. W. Klimbie and K. L. Koffeman (Eds.), North-Holland Publ. Co., Amsterdam, The Netherlands, 1974, pp. 399–412.

[I6] WARREN, THOMAS, *Feature analysis of CODASYL database management systems*, AD-A014 972/4WC, National Technical Information Service, Springfield, Virginia, 1975.

(M) Other Data Modeling Papers

[M1] ABRIAL, J. R., "Data semantics," in *Database Management*, J. W. Klimbie and K. L. Koffeman, (Eds.), North-Holland Publ. Co., Amsterdam, The Netherlands, 1974, pp. 1–60.

[M2] COLLMEYER, A. J., "Implications of data independence on the architecture of database management systems," in *Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control*, 1972, A. L. Dean, (Ed.), ACM, New York, pp. 307–321.

[M3] EARNEST, C. P., "Selection and higher level structues in networks," in *Database Description*, B. C. M. Douque and G. M. Nijssen, (Eds.), North-Holland Publ. Co., Amsterdam, The Netherlands, 1975, pp. 215–237.

[M4] EVEREST, G. C.; AND SIBLEY, E. H., "Critique of the GUIDE-SHARE DBMS requirements," in *Proc. ACM SIGFIDET Workshop on Data Description, Access and Control*, 1971, E. F. Codd and A. L. Dean, (Eds.), ACM, New York, pp. 93–112.

[M5] GUIDE-SHARE DATABASE REQUIREMENTS GROUP, *Database management system requirements*, 1970, SHARE, Inc., New York.

[M6] HUITS, M. H. H., "Requirements for languages in database systems," in *Database Description*, B. C. M. Douque and G. M. Nijssen, (Eds.), North-Holland Publ. Co., Amsterdam, The Netherlands, 1975, pp. 85–109.

[M7] MCGEE, W. C., "File level operations on network data structures," in *Proc. ACM SIGMOD Internatl. Conf. on Management of Data*, 1974, W. F. King, (Ed.), ACM, New York, pp. 32–47.

[M8] OLLE, T. W., *An assessment of how the CODASYL data base task group proposal meets the GUIDE-SHARE requirements*, Report 329 Norwegian Computing Center, 1972.

[M9] SENKO, M. E., et al., "Data structures and accessing in database systems," *IBM Systems J.* **12**, 1 (1973), 30–93.

[M10] STEEL, T. B., JR., "Database standardization: a status report," in *Database Description*, B. C. M. Douque and G. M. Nijssen, (Eds.), North-Holland Publ. Co., Amsterdam, The Netherlands, 1975, pp. 183–198. Also in *Proc. ACM SIGMOD Internatl. Conf. on Management of Data*, 1975, W. F. King, (Ed.), ACM, New York, pp. 149–156.

[M11] PARSONS, R. G.; DALE, A. G.; AND YURKANEN, C. V., "Data manipulation language requirements for database management systems," *Computer J.* **17**, 2 (May 1974), 99–103.

(G) Referenced Papers

[G1] DODD, G. G., "Elements of data management systems," *Computing Surveys* **1**, 2 (June 1969), 117–133.

[G2] MOLER, C., "Matrix computations with FORTRAN and paging," *Comm. ACM* **15**, 4 (April 1972), 268–270.

[G3] SEVERANCE, D. G.; AND DUHNE, R. A., "A practitioner's guide to addressing algorithms," *Comm. ACM*, (publication pending).

[G4] NIJSSEN, G.M., "Efficient batch updating of a random file," in *Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control*, 1971, E. F. Codd and A. L. Dean, (Eds.), ACM, New York, pp. 173–186.

[G5] DATE, C. J., *An introduction to database systems*, Addison-Wesley, Reading, Massachusetts, 1975.

ACKNOWLEDGMENTS

The authors are grateful to M. L. O'Connell of the CODASYL Data Base Language Task Group for clarifying several points. This research was supported, in part, by the National Science Foundation under Grants GJ-41829 and GJ-41830.