Hierarchical Data-Base Management: A Survey

D. C. TSICHRITZIS

and

F. H. LOCHOVSKY

Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A7

This survey paper discusses the facilities provided by hierarchical data-base management systems. The systems are based on the hierarchical data model which is defined as a special case of the network data model. Different methods used to access hierarchically organized data are outlined. Constructs and examples of programming languages are presented to illustrate the features of hierarchical systems. This is followed by a discussion of techniques for implementing such systems. Finally, a brief comparison is made between the hierarchical, the network, and the relational systems.

Keywords and Phrases: data base, data-base management, data manipulation, data model, data independence, generalized processing, hierarchical data model, hierarchical languages, hierarchical systems, network systems, record retrieval, relational systems

CR Categories: 3.51, 4.33, 4.34

INTRODUCTION

In the real world, there seems to be a great deal of order and structure. It may be that this structure is implicit as a basic property of the physical world. On the other hand, it may be that humans simply use this structure to understand the world better. Whether God-given or human created, structure is instrumental in helping men to understand the world. One of the most common and simplest structures is the hierarchical structure. In the natural world, among the animals, for instance, there is a hierarchical structure that is graded according to intelligence. Some human organizations are also arranged hierarchically, e.g., management structures. Because hierarchies are so familiar in nature and in human society, it seems natural to us that our ideas about the world are often arranged in a hierarchical fashion.

Data bases portray and represent information about the world. Since the world appears to us to be arranged hierarchically, it seems natural that data base management systems (DBMSs) should provide the tools needed to represent and manipulate hierarchical structures. There are actually a number of very successful DBMSs which provide the user with a hierarchical view of the world [9, 14, 17]. In this paper, we discuss the facilities offered by these DBMSs. Before considering these facilities, however, we first have to define, exactly, the nature of a hierarchical structure.

HIERARCHICAL DATA MODEL

Data represent ideas about the real world which people conceive in terms of entities

CONTENTS

and their attributes. A *data model* is a method of logically organizing such data according to the relationships found among the data. In general, two types of relationships can be distinguished [19]. The first type is a relationship among attributes of the same entity; the second type is a relationship between entities. The difference between the two types can be understood by examining the following examples.

- The *attribute relationship* becomes clear if we consider presidential elections. A presidential candidate has a name and is affiliated with a political party. The candidate's name and his political party both describe the candidate, and these two descriptions are of interest to us only as long as the election is of interest—perhaps only until the election is held.
- The *entity relationship* between two entities becomes clear if we consider the relationship between a president and an election. Both the president and the election exist independently of one another, whether or not the president still holds office as a result of an election.

In the sequel, attribute relationships of an entity will be represented by a *record type*. A *record type* is a named collection of *data items*. A *data item* is the smallest named logical unit of data; it represents an attribute of an entity. Since relationships between attributes are represented within record types, we will concentrate mainly on relationships among different entities.

The relationships between entities can be represented by a graph [1]. In the graph, each node indicates a record type that represents an entity and each undirected arc specifies a relationship between the entities. Many relationships are possible between any two nodes. Each relationship is named explicitly to distinguish it from other relationships. Figure 1 shows some of the relationships between presidents, congresses, etc.

Each relationship depicted in Figure 1 can potentially be a general $N:M$ relationship. Taylor and Frank have shown, for instance, in their paper on CODASYL data base management systems in this special issue [see page 67] that the relationship between PRESIDENT and CONGRESS is $N:M$. However, most of the relationships are $1:N$, e.g., ADMINISTRATIONS HEADED, ADMITTED DURING, etc. In addition, as it was shown in the same companion paper, there is a way of transforming a general $N:M$ relationship into two $1:N$ relationships by introducing an intermediate record type. By applying this transformation repeatedly, we can obtain a relationship graph, called a *data structure diagram*, where all the relationships are $1:N$, as seen in Figure 2 [2].

The relationships between the entities can be viewed as relationships between the record types. All of the relationships can be assumed to be $1:N$ without loss of generality. The $1:N$ relationships have a certain direction that is represented by introducing directed arcs in the graphs, as we have done in Figure 2. Each arc points from one to N in the $1:N$ relationship. For instance, between the record types STATE and PRESIDENT, the arc representing the relationship NATIVE SONS points from STATE to PRESIDENT, since each state may have many native son presidents, but each president was born in only one state.

Consider the special case where the data structure diagram represents a tree in which the direction of the arcs points away from

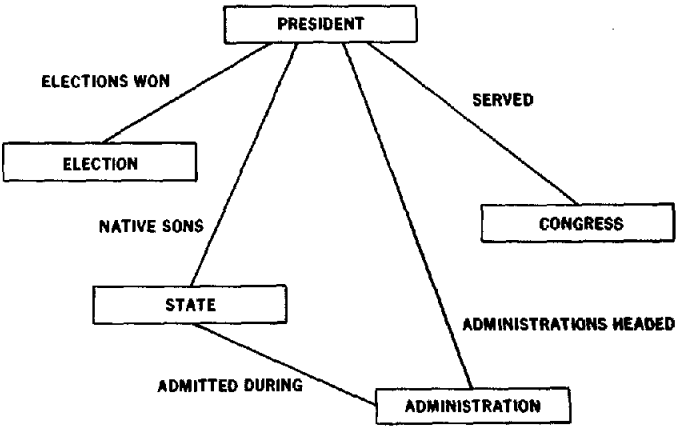


FIGURE 1. A general relationship graph.

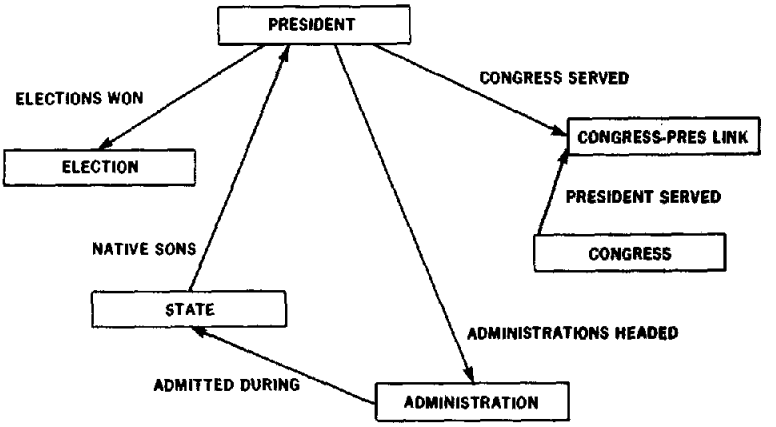


FIGURE 2. A data structure diagram.

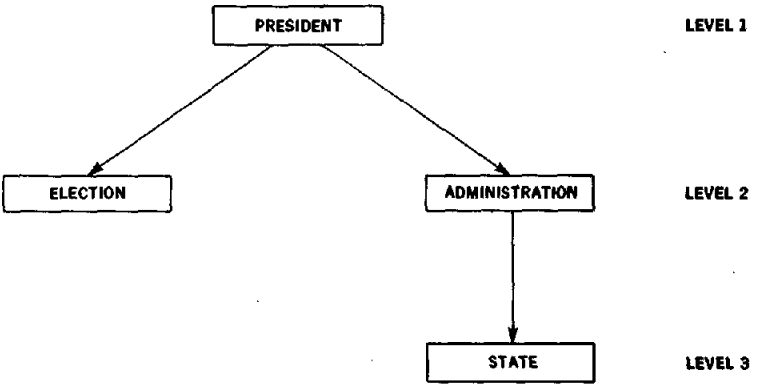


FIGURE 3. Hierarchical definition tree.

the root, as in Figure 3. Because there can be at most one arc between any two record types the arcs do not need to be labeled. Such a restricted data structure diagram is called a *hierarchical definition tree*. A hierarchical definition tree specifies both what record types are allowed to be included in the data base and the permissible relationships be-

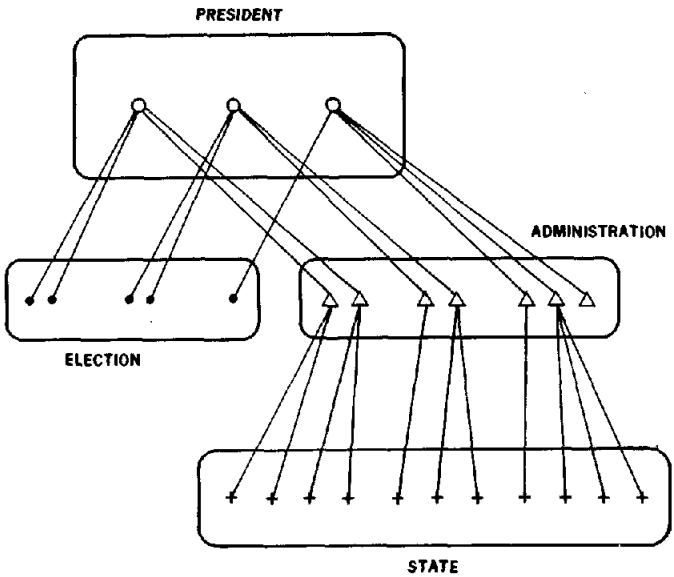


FIGURE 4. Data-base trees.

tween record types. Figure 3 represents a hierarchical definition tree—one that is a subset of the data structure diagram shown in Figure 2. The *hierarchical data model* is defined as the data model which organizes data logically according to the structural relationships of hierarchical definition trees.

The *level* of a record type in the hierarchical definition tree is a measure of its distance from the root of the tree. The *root record type* is the highest level record type in the tree (level 1). The PRESIDENT record type in Figure 3 is an example of a root record type. The other record types, called *dependent record types*, are at lower levels in the tree (levels 2, 3, etc.). In this instance the ELECTION and ADMINISTRATION record types are both at level 2.

A data base that corresponds to the hierarchical definition tree of Figure 3 is shown in Figure 4. The *hierarchical data base* is a collection or *forest* of trees called *data-base trees* whose record occurrences appear as nodes. There are, for instance, three data base trees in Figure 4. All of the trees are constructed according to the relationships permitted explicitly in the hierarchical definition tree. In a hierarchical data base, *parents* and *children*, or *ancestors* (parents, parents of parents, etc.) and *descendants* (children, children of children, etc.) among the record occurrences can be identified in a natural way. A *hierarchical path* is a sequence of records in which the records, starting at a root record, follow alternately in a parent-child relationship. For example, the sequence: PRESIDENT record, ADMINISTRATION record, STATE record, defines a hierarchical path.

In a hierarchical data base that is structured like Figure 3, each PRESIDENT record occurrence can have many ADMINISTRATION record occurrences connected to it. Each ADMINISTRATION record occurrence may (in turn) have several STATE record occurrences connected to it, as exemplified in Figure 4. Each STATE record occurrence, however, has exactly one parent record occurrence: ADMINISTRATION (during which the state was admitted), and one grandparent record occurrence: PRESIDENT (who served when the state was admitted).

Two things should be noted in Figure 4. First, there can be a varying number of occurrences of each record type at each level. Second, each record occurrence (except for a root record occurrence, PRESIDENT) must be connected to an occurrence of an ancestor record type as constrained by the hierarchical definition tree. There can be no "independent" record occurrences of record

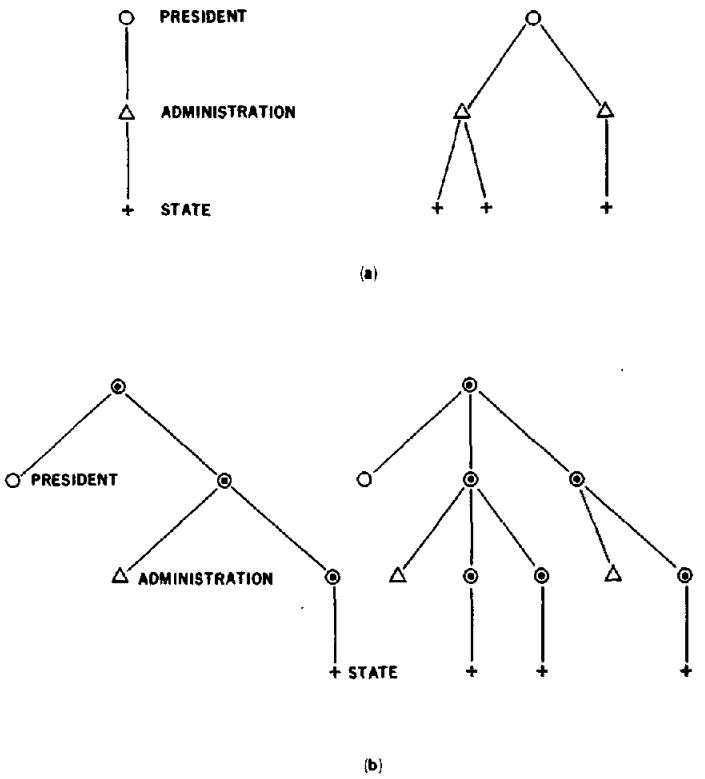


FIGURE 5. Two data-base hierarchies.

types ELECTION, ADMINISTRATION, or STATE. If we regard the relationships between parent and child record types as a data structure set, then membership in the set is always mandatory in terms of the DBTG network systems, as Taylor and Frank have discussed in the paper already mentioned [page 67].

The reader can find in the literature a similar, but not identical, notation for representing a hierarchical data base. This notation allows the data to reside only at the terminal nodes of the tree [5]. All intermediate nodes of the data-base trees are present only to maintain the hierarchical relationships. Only the terminal nodes have record occurrences with data-item values.

Figure 5(a) shows a hierarchical definition tree and a data-base tree of the type previously discussed; Figure 5(b) represents the same information in terms of a hierarchical definition tree where the record types are present only at the terminal nodes. The intermediate nodes (marked with a dot since they carry no data) serve only to maintain the structure and to qualify the data. Since, logically, the two notations are equivalent, for simplicity and uniformity only the former notation will be used.

All features of the example of a presidential data base shown in Figure 2 cannot be completely captured by one hierarchical definition tree. The data structure diagram is itself a network. However, we can represent the same information by using more than one hierarchical definition tree as shown in Figure 6.

The record types contain the following data items:

PRESIDENT—PRES NUMBER, PRES NAME, BIRTHDATE, DEATH DATE, PARTY, SPOUSE.
ELECTION—YEAR, PRES VOTES, LOSER VOTES, LOSER, LOSER PARTY.
ADMINISTRATION—ADMIN NUMBER, INAUG DATE, VICEPRESIDENT.

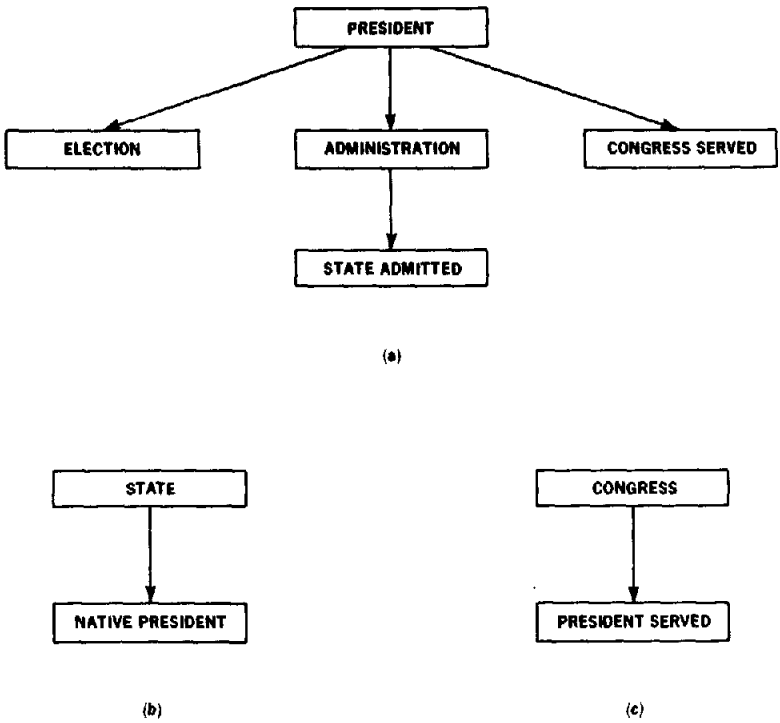


FIGURE 6. Presidential data-base hierarchical definition trees.

CONGRESS SERVED—CONGRESS NUMBER.
STATE ADMITTED—STATE NAME.
STATE—STATE NAME, POPULATION, STATE VOTES.
NATIVE PRESIDENT—PRES NUMBER.
CONGRESS—CONGRESS NUMBER, SENATE REP PERCENT, SENATE DEM PERCENT, HOUSE REP PERCENT, HOUSE DEM PERCENT.
PRESIDENT SERVED—PRES NUMBER.

HIERARCHICAL LANGUAGES

A *hierarchical system* is a DBMS which provides facilities for inserting, modifying, deleting, and retrieving record occurrences in a hierarchical data base. Because of the nature of the hierarchical data model, each new record that is inserted (except for a root record occurrence) has to be connected to an occurrence of a parent record type. Usually this action is effected by selecting a parent record occurrence and inserting a child. For example, in Figure 5, before one can insert an occurrence of a STATE record type, an occurrence of an ADMINISTRATION record type would first have to be selected.

When a record occurrence is deleted, all of its descendant record occurrences are also deleted. The hierarchical data model does not allow non-root records to exist without ancestors. In Figure 5, if an ADMINISTRATION record occurrence is deleted, all occurrences of STATE record types connected to the occurrence of the ADMINISTRATION record type also have to be deleted. To retain the descendant records, but not a parent record, as is sometimes necessary, some systems provide commands which delete the data-item values, but not the record itself [6, 9, 10, 17]. Essentially, this facility permits null (empty) records to exist in the data base. In this way, the data associated with the descendant record is retained.

Records retrieved in a hierarchical data base may be both *selected* and *qualified* ac-

cording to the tree structure [12, 13, 18]. Records are usually selected by means of a *qualification* which expresses the criterion of selection. The qualification consists of conditions of the form

(⟨data item name⟩⟨conditional operator⟩⟨value⟩)

connected by Boolean operators AND, OR, and NOT. The usual conditional operators are $<$, $\leq$, $>$, $\geq$, $=$, and $\neq$ or their mnemonic equivalent. The qualification may select records of several different record types. In our example, a qualification such as ((PRES NAME=EISENHOWER) AND (YEAR=1956)) selects both PRESIDENT and ELECTION records, In general, the qualification can be associated with data items from any record type in the hierarchical definition tree. However, almost all systems permit the qualification to contain data items only from record types that lie in a hierarchical path. In this way, they avoid the ambiguity that arises when the Boolean negation operator (NOT) is specified in the qualification [12, 13].

After a record has been selected, other records may *qualify* for retrieval. Every record has, for example, a unique set of ancestors in the data base. All ancestors of the selected record may qualify for retrieval. Further, a record may have a set of descendants, all of which can also qualify for retrieval. For example, an ADMINISTRATION record may have several STATE records connected to it. Notice that each selected record always has no more than one ancestor record of each ancestor record type. That is, each record has, at most, one parent which (in turn) has, at most, one parent, and so on. However, in general, a selected record may have several descendant records of each descendant record type, since it may have several children which (in turn) may have several children, etc. When most systems qualify descendants, this qualification is performed along only one hierarchical path. This means that from a PRESIDENT record, one can qualify ELECTION records or ADMINISTRATION records, but not both. The qualifying of ancestors of a selected record is called *upward hierarchical normalization*, while the qualifying of descendants is called *downward hierarchical normalization* [16].

The retrieval operation in a hierarchical data base may be performed in one of two ways. According to the first method, a user explicitly uses the tree structure of the data base to *traverse* the data-base trees in a specified order. Traversal may be independent of record selection and qualification, in which case it resembles sequential processing. On the other hand, records may be selected and qualified, but in a very restricted manner only determined by the traversal order. According to the second retrieval method, the user selects and qualifies records based on the relationships between the data items of the record types. Although the user has to be aware of the tree structure of the data base, he does not explicitly use this structure to retrieve records. Instead, the system utilizes the hierarchical structure to determine which records qualify among those selected by the user. Each method will be discussed in detail in the following sections.

Tree Traversal

In systems that use a tree traversal language, record types and record occurrences are sometimes called *segment types* and *segment occurrences* or simply *segments* [14]. A single data-base tree consisting of one root record occurrence and all its associated descendants, is sometimes called a *data-base record*. The root record type of the hierarchical definition tree is referred to as the *root segment type*. The other segment types are called *dependent segment types*. Each segment type is further divided into data items called *fields*. In this section, the original terminology of the Hierarchical Data Model section, page 105, will be used.

One of the DBMSs that uses a tree traversal language is the Information Management System (IMS) [14]. IMS uses a preorder tree traversal to traverse a data-base tree. A preorder tree traversal is defined as follows [15]:

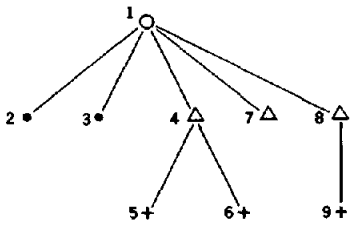
1) Visit the record if it has not already been visited.

2) Else, visit the leftmost child not previously visited.
3) Else, if no children, grandchildren, etc., remain to be visited, go back to the parent record.

These steps are applied to each record of the data-base tree when the record is reached. It is assumed that the children of each parent are ordered according to the appearance of the child record type in the hierarchical definition tree, i.e., all ELECTION records under a PRESIDENT record come before any ADMINISTRATION records.

The traversal begins at the root record. The traversal essentially visits all records in the tree going in top-to-bottom, left-to-right order. Taking as an example the data-base tree given in Figure 7, a preorder tree traversal would visit the records in the order indicated. If one imagines individual occurrences of data-base trees to be connected to an imaginary head record, a single data-base tree is formed; this procedure makes it possible to visit all the records in the data base. During the traversal, the selection of record occurrences can take place. In addition, the records may be qualified.

Requests to IMS are specified through procedure calls to Data Language/One (DL/I) from application programs written in PL/I, COBOL, or Assembler Language. A position pointer marks the progress of an application program through the data base according to a preorder traversal of the data-base tree. The calls to DL/I require several parameters which are passed through a subroutine call in the host language. These parameters identify communication buffers, I/O buffers, the type of call, and qualifications. However, only the last two parameters are relevant to the following discussion.



○ PRESIDENT

• ELECTION

△ ADMINISTRATION

\+ STATE

FIGURE 7. Preorder tree traversal.

During the operation of retrieving records, several records may be selected by means of one or more qualifications. After selection, other records may be qualified. No more than one record of each record type, located in the hierarchical path, may be qualified. A qualification on a record is specified by means of a segment search argument (SSA). An SSA specifies a qualification that applies to only one record type. The form of an SSA is:

⟨RECORD NAME⟩⟨COMMAND CODE⟩⟨QUALIFICATION⟩

The ⟨RECORD NAME⟩ is the name of a record type in the hierarchical definition tree. A ⟨QUALIFICATION⟩ is optional in the SSA and is expressed in the form specified previously. The only Boolean operators permitted are AND and OR.

The ⟨COMMAND CODE⟩ is optional and specifies the various options of the call. Some of the more important options permit:

- retrieval or insertion of some or all of the records from the root to a specified record type in a single DL/I call (path call);
- backing up to the first child under a record at any or all levels (except at the root record level);
- retrieval of the last occurrence of a record that meets all specified conditions under a parent;
- setting of the parentage to a specific record (see the *GET NEXT WITHIN PARENT* call below).

The set of SSAs in a DL/I call specify a path that passes from a root record down the data-base tree to a specified record. There may be no more than one SSA for each level in the hierarchical path. If an SSA, located at a particular level, does not uniquely identify a record, or if no SSA is specified, then IMS selects the hierarchically next

(preorder) record, except when modified by command codes.

Every DL/I call results in the setting of a status code parameter in the communication buffer used by IMS to communicate with an application program. The status code may be checked after every DL/I call to determine whether the result of the call is successful, unsuccessful, warning, or some other condition. It is then up to the application program to determine the appropriate action to be taken. The various status codes will not be discussed here. The interested reader should refer to the appropriate manuals [14].

Data-base calls are categorized into three general types: *GET* calls; *INSERT* calls; and, *DELETE* and *REPLACE* calls. All calls may optionally include some form of SSAs.

The *GET UNIQUE* call retrieves a record, as selected by the SSAs, independently of its current position in the data base. This call is used for nonsequential processing or to establish a start position for sequential processing of the data base.

The *GET NEXT* call processes in a forward direction only, starting from the current position in the data base. Records (optionally of a particular type) are retrieved in the order established by a preorder traversal of the data-base trees. *GET NEXT* can be used without SSAs to retrieve the records in the data base sequentially. It can also be used to search for a particular record if an SSA is included in the call.

A *GET NEXT WITHIN PARENT* call obtains records (optionally of a particular type) within the family of a parent record. The parent record is established by the last *GET NEXT* or *GET UNIQUE* call, or by the (COMMAND CODE) option in an SSA of a preceding *GET NEXT* or *GET UNIQUE* call. The only difference between a *GET NEXT* and a *GET NEXT WITHIN PARENT* call is the result obtained after the last child (optionally of a given type) under a given parent is reached. A *GET NEXT* call continues to retrieve records until IMS sets the status code to signal that the end of the data base has been reached. A *GET NEXT WITHIN PARENT* call sets the status code to signal that there are no more children under the parent.

A *GET HOLD* call is used to retrieve and hold a record for a *DELETE* or *REPLACE* call. A *GET HOLD* call allows only one application at a time to alter a record by serializing access to the record.

An *INSERT* call is used to load a data base or to add new records to a data base. SSAa are used to select the position (in the data-base tree) where the record is to be inserted. Notice that an *INSERT* call performs two operations: it stores the new record, and connects it to its parent. This dual operation is necessary, since every record, except a root record, must have a parent.

A *DELETE* call deletes a record and all of its descendant records from the data base. The *DELETE* call is a "triggered" delete—that is, the record selected and all of its descendants are deleted.

A *REPLACE* call updates records in the data base. Any nonkey data item may be changed in the record. Attempting to change a key data item results in an error.

To demonstrate the use of the DL/I calls, the following query will be implemented using PL/I: Print the names of all the states that were admitted during a Democratic administration. The query is first presented in its entirety, and then an explanation of its various parts is given. It is assumed that there is a hierarchical data-base structure that conforms to the hierarchical definition tree shown in Figure 6(a). Only the PRESIDENT and STATE ADMITTED record types are required for this query. To conform to the naming conventions of IMS, these two record types are renamed PRES and SADMIT, respectively.

```
DLITPLI:PROCEDURE (QUERY_PCB) OPTIONS (MAIN);
    DECLARE QUERY_PCB POINTER;
    /* Communication Buffer */
    DECLARE 1 PCB BASED (QUERY_PCB),
      2 DATA_BASE_NAME CHAR (8),
```

```
    2 SEGMENT_LEVEL CHAR (2),
    2 STATUS_CODE CHAR (2),
    2 PROCESSING_OPTIONS CHAR (4),
    2 RESERVED_FOR_DLI FIXED BINARY (31,0),
    2 SEGMENT_NAME_FEEDBACK CHAR (8),
    2 LENGTH_OF_KEY_FEEDBACK_AREA FIXED BINARY (31,0),
    2 NUMBER_OF_SENSITIVE_SEGMENTS FIXED BINARY (31,0),
    2 KEY_FEEDBACK_AREA CHAR (28);
  /* I/O Buffers */
  DECLARE PRES_IO_AREA CHAR (65),
    1 PRESIDENT DEFINED PRES_IO_AREA,
    2 PRES_NUMBER CHAR (4),
    2 PRES_NAME CHAR (20),
    2 BIRTHDATE CHAR (8),
    2 DEATH_DATE CHAR (8),
    2 PARTY CHAR (10),
    2 SPOUSE CHAR (15);
  DECLARE SADMIT_IO_AREA CHAR (20),
    1 STATE_ADMITTED DEFINED SADMIT_IO_AREA,
    2 STATE_NAME CHAR (20);
  /* Segment Search Arguments */
  DECLARE 1 PRESIDENT_SSA STATIC UNALIGNED,
    2 SEGMENT_NAME CHAR (8) INIT ('PRES    '),
    2 LEFT_PARENTHESIS CHAR (1) INIT ('('),
    2 FIELD_NAME CHAR (8) INIT ('PARTY   '),
    2 CONDITIONAL_OPERATOR CHAR (2) INIT ('='),
    2 SEARCH_VALUE CHAR (10) INIT ('DEMOCRAT  '),
    2 RIGHT_PARENTHESIS CHAR (1) INIT (')');
  DECLARE 1 STATE_ADMITTED_SSA STATIC UNALIGNED,
    2 SEGMENT_NAME CHAR (8) INIT ('SADMIT  ');
  /* Some necessary variables */
  DECLARE GU CHAR (4) INIT ('GU  '),
    GN CHAR (4) INIT ('GN  '),
    GNP CHAR (4) INIT ('GNP  '),
    FOUR FIXED BINARY (31) INIT (4),
    SUCCESSFUL CHAR (2) INIT ('  '),
    RECORD_NOT_FOUND CHAR (2) INIT ('GE');
  /*This procedure handles IMS error conditions */
  ERROR:PROCEDURE (ERROR_CODE);
      .
      .
      .
  END ERROR;
  /* Main Procedure */
  CALL PLITDLI (FOUR,GU,QUERY_PCB,PRES_IO_AREA,PRESIDENT_SSA);
  DO WHILE (PCB.STATUS_CODE=SUCCESSFUL);
    CALL PLITDLI (FOUR,GNP,QUERY_PCB,SADMIT_IO_AREA, STATE_ADMITTED_SSA);
    DO WHILE (PCB.STATUS_CODE=SUCCESSFUL);
      PUT EDIT (STATE_NAME) (A);
  CALL PLITDLI (FOUR,GNP,QUERY_PCB,SADMIT_IO_AREA, STATE_ADMITTED_SSA);
    END;
```

```
      IF PCB.STATUS_CODE¬ = RECORD_NOT_FOUND
        THEN DO;
          CALL ERROR (PCB.STATUS_CODE);
          RETURN;
          END;
      CALL PLITDLI (FOUR,GN,QUERY_PCB,PRES_IO_AREA, PRESIDENT_SSA);
    END;
    IF PCB.STATUS_CODE¬ = RECORD_NOT_FOUND
      THEN DO;
        CALL ERROR (PCB.STATUS_CODE);
        RETURN;
        END;
END DLITPLI;
```

To run this query, the user would invoke IMS. After IMS has performed the necessary initializations, it passes control to the procedure called DLITPLI. Within his program, a user must declare a mask for a communication buffer, allocate I/O buffers, and set up the formats of the various SSAs that will be needed.

The declaration of the PL/I structure named PCB is an outline of the structure for the communication buffer, established by IMS, that is needed for communication with the user's program. The buffer is allocated by IMS in the initialization phase, and a pointer to it (QUERY_PCB) is passed as a parameter, to the application program. In general, there may be several communication buffers, one for each data base accessed by the application. The declaration of the PL/I structure, PCB, only serves to facilitate access to the communication buffer via the structure entry names. The buffer is used by DL/I to advise an application program of the results of its DL/I calls. All entries in the buffer are read-only, and most are self-explanatory. The only entry of interest in this example is the STATUS_CODE. The STATUS_CODE indicates whether the result of a DL/I call is successful, or an error, a warning, or other condition.

Next, the I/O buffers for the record types are declared. There is usually one I/O buffer for each record type in the hierarchical definition tree. The I/O buffers are used to hold the record, corresponding to the appropriate record type, that is retrieved by IMS, and to make the record available in the application program. The use of PL/I structures, applied, for example, to PRESIDENT, facilitates access to the data items of the record type. All records are assumed to be stored as character data. In this example, only buffers for the PRESIDENT and STATE ADMITTED record types are required.

Finally, the SSAs are declared. There may be one SSA for each record type in the hierarchical definition tree. In the example given, only the PRESIDENT and STATE ADMITTED record types require SSAs. Although the SSAs in this example do not change after they have been declared and initialized, it is possible to change the search value and/or the conditional operator by means of the host languages facilities.

The ERROR procedure will not be described in detail here. It is used to print error messages and to take appropriate action when an error condition arises.

As mentioned earlier, a program accesses an IMS data base by making a subroutine call to DL/I. In PL/I, such DL/I calls are characterized by the starting sequence CALL PLITDLI. These calls may have a varying number of parameters. However, in the present example each of the calls uses five parameters, which are in order of appearance within the call:

- the number of parameters to follow (in this example, four);
- the type of call, e.g., *GET NEXT*, *INSERT*, etc.;
- the pointer to the communication buffer for the call;

- the location of the I/O buffer; and
- the SSA(s) for the call.

The processing of the query is performed in several steps as follows:

1) Retrieve the first occurrence of a PRESIDENT record where the PARTY data item is equal to DEMOCRAT. This action is performed by means of a *GET UNIQUE* (GU) call using the PRESIDENT_SSA. If no record is selected, then the processing is complete.
2) Get the first STATE ADMITTED record that is a descendant of the PRESIDENT record. This action is performed by means of a *GET NEXT WITHIN PARENT* (GNP) call using the STATE_ADMITTED_SSA. Notice that it is not necessary to specify an SSA for ADMINISTRATION records. We are not really interested under which administration the state was admitted, but only under which president. IMS, therefore, selects each ADMINISTRATION record in turn as required and processes its associated STATE ADMITTED records. If no states were admitted during a president's tenure, then we go to step 4.
3) Get all the other STATE ADMITTED records, in turn, by means of a *GET NEXT WITHIN PARENT* (GNP) call until there are no more STATE ADMITTED records for this president. Print the name of each state.
4) Get the next PRESIDENT record where the PARTY data item is equal to DEMOCRAT. This action is effected by a *GET NEXT* (GN) call using the same SSA as for the *GET UNIQUE* call in step 1. If no PRESIDENT record is selected, then the processing is complete. Otherwise go to step 2.

Tree traversal languages usually operate on one record at a time. They essentially perform a linear tree traversal. As a result, although they operate on a tree, their processing looks sequential. The nature of the commands they use can also influence their implementation. It would be nice if the record in the hierarchy that is logically next would also be the next physical record. If performed in this way, sequential processing of the data-base tree would be very efficient. In the Implementation section, [see page 118], we will examine some implementations of a hierarchical data base.

General Selection

The terminology employed in systems that use general selection languages differs from that of systems using tree traversal languages [5, 6, 7, 9, 10, 11, 17, 20]. Among the general selection languages, a record type is sometimes referred to as a *repeating group*. A record occurrence is called a *repeating group occurrence* or *data set*. A data-base tree consisting of one root record and all its descendants is called a *logical entry*. Finally, a data item is called a *data element*. However, to avoid terminological confusion, repeating groups will be called record types, and the other terms will also be named as before. In the following discussion, a hierarchy structured according to the hierarchical definition trees given in Figure 6 will be used.

General selection languages treat the record occurrences of a record type as sets of data-item values. Records are selected and qualified according to the relationships found among the data items of the record types of the hierarchical definition tree. A qualification in the general selection languages is usually specified by a WHERE clause [12, 13, 17]. A WHERE clause consists of the keyword WHERE and a Boolean combination of the conditions discussed earlier. The WHERE clause specifies the records to be selected. After these specifications have been effected, upward and/or downward hierarchical normalization can be performed. Downward hierarchical normalization is usually restricted to one hierarchical path.

The common characteristic features of general selection languages will be illustrated by examples using the "Natural Lan-

guage" feature of the SYSTEM 2000 [9, 17]. This feature is typified by an English-like, interactive query language. The commands of this language consist of two parts: an action part and a WHERE clause. The action part specifies the operation to be performed, e.g., retrieval, update, etc., and the data items on which to operate. The basic retrieval command is the *PRINT* command. The following query illustrates upward hierarchical normalization.

PRINT VICEPRESIDENT *WHERE* STATE NAME *EQ* ALASKA:

This query specifies that we want VICEPRESIDENT data-item values in ADMINISTRATION records. In addition, the ADMINISTRATION records must have a STATE ADMITTED descendant that satisfies the WHERE clause. To provide this, we select all STATE ADMITTED records satisfying the WHERE clause and then perform an upward hierarchical normalization to qualify ADMINISTRATION records. In general, several ADMINISTRATION records may qualify. However, according to the semantics of the example, only one ADMINISTRATION record would qualify.

The next query illustrates downward hierarchical normalization.

PRINT STATE NAME *WHERE* VICEPRESIDENT *EQ* NIXON:

We examine the ADMINISTRATION records to select those records that satisfy the WHERE clause. Although, in general, several records may be selected, we are constrained in this example to select only one record. A downward hierarchical normalization is then performed to qualify STATE ADMITTED records. Many STATE ADMITTED records may qualify. Note that this and the preceding query are conceptually symmetric queries. However, they are answered in quite different ways.

PRINT ADMIN NUMBER *WHERE* VICEPRESIDENT *EQ* NIXON:

This query involves neither upward nor downward hierarchical normalization. The query is answered by selecting and qualifying only ADMINISTRATION records. In this case again, several records may be selected and qualified.

Now consider the query:

PRINT PRES NAME *WHERE* VICEPRESIDENT *EQ* AGNEW *AND* VICEPRESIDENT *EQ* FORD:

The casual user would perhaps expect the response to this query to be Nixon as the president who had both Agnew and Ford as vice-presidents. However, the semantics of the WHERE clause are such that the answer to this query is null, i.e., no PRESIDENT record qualifies. This problem arises because all Boolean operations must be performed at the same level at which records are selected, that is, on ADMINISTRATION records. The result is that the intersection (AND) of two conditions which specify the same data item—but different values—mus be null. The same record cannot have two different values for the same data item.

If the Boolean operations could be performed on qualified records at a higher level, then the problem would be solved. The purpose of the HAS clause is to raise the level at which the Boolean operations are performed by carrying out an upward hierarchical normalization on the selected records. For instance, the query:

PRINT PRES NAME *WHERE* PRES NAME *HAS* VICEPRESIDENT *EQ* AGNEW *AND* PRES NAME *HAS* VICEPRESIDENT *EQ* FORD:

will produce the answer "Nixon," as expected. In the previous query, using only the WHERE clause, the intersection (AND) was performed on ADMINISTRATION records. In this query, the intersection is performed on PRESIDENT records, since the level of Boolean operations is raised to the PRESIDENT record by means of the HAS clause. (The level to which upward hierarchical normalization is performed is specified by the data-item name preceding the keyword HAS.) In the first query, no ADMINISTRATION record satisfies both conditions of the WHERE clause. In the second

query, there is at least one PRESIDENT record which has ADMINISTRATION records that satisfy the HAS clauses.

Approaching the problem from a different angle, one can visualize the HAS clause as selecting records of a certain type and then determining whether there is at least one parent record connected to *all* the selected records. On the other hand, the WHERE clause starts at a parent record and determines whether for a given parent, *one* descendant exists that satisfies the WHERE clause.

The HAS clause is also useful in other situations. For example, sometimes it is necessary to retrieve values from a record of one type based on the selection of a record of a different type at the same level. For instance, suppose we want to know all the election years for the president with administration number 48. A WHERE clause does not have the capability to retrieve these values. However, a HAS clause does, as follows:

PRINT YEAR *WHERE* PRES NAME *HAS* ADMIN NUMBER *EQ* 48:

We select the appropriate ADMINISTRATION record(s) and then do an upward hierarchical normalization to qualify a PRESIDENT record. We then do a downward hierarchical normalization to qualify all ELECTION records connected to the PRESIDENT record. It is also possible to retrieve a year based on the selection of a STATE ADMITTED record.

PRINT YEAR *WHERE* PRES NAME *HAS* STATE NAME *EQ* ALASKA:

In general, the insertion, update, and deletion commands for general selection languages are similar in their operational effect to the commands outlined in the Tree Traversal section, [See page 111]. However, systems that use general selection languages usually distinguish between a record and its position in a data-base tree. This means a position in the data-base tree may exist, but either there may not be any record, or only a part of a record may be associated with it. For example, in SYSTEM 2000, the *REMOVE* command deletes data-item values from a record occurrence [9, 17]. The effect is to make the data-item value null. But the record is not deleted and associated descendant records are therefore retained.

General selection languages are truly hierarchical. They use tree operations to select and qualify records in the data base. Records are qualified because they are ancestors or descendants of particular selected records. Queries that require movement up and down the data-base tree can usually be handled in one command. Because of these features, general selection languages are more versatile than tree traversal languages which traverse the data-base trees in a specific linear manner.

IMPLEMENTATION

To represent and then implement a hierarchical data base, we must first be able to specify the connections between the records, i.e., how are the records related to one another. One way of implementing a hierarchical data base is by the use of pointers. Essentially every relationship between two record types in the hierarchical definition tree has a set of pointers associated with it. The pointers implement the connections between the parent and child record occurrences. For instance, for the data base tree outlined in Figure 4, both a forward and a backward pointer can be associated with every branch of the tree.

While this organization is simple to visualize it has many drawbacks. A great deal of space is taken up by pointers. The amount of space required for pointers that are associated with a parent record varies as children are inserted and deleted. It would facilitate storage allocation if this space requirement were fixed. To this end, some systems limit the number of children of a given type that a parent may have [8]. If upward or downward hierarchical normalization of more than one level is adopted, much pointer chasing becomes necessary.

To reduce the space occupied by pointers, the connections can be implemented in other, slightly different ways. As an example of a way of avoiding many forward

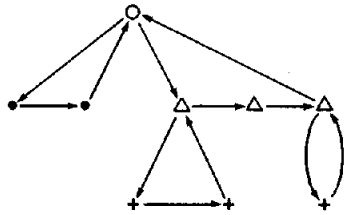


FIGURE 8. Hierarchical pointer implementation.

and backward pointers between parent and children, only one forward pointer plus one brother pointer can be used (Figure 8) [16]. In such a case, each record has a fixed amount of pointer overhead. This organization saves space, but now even more pointer chasing is involved. Whereas in the previous organization any child of a parent can be accessed by following one pointer, this organization may demand that several pointers be followed.

Instead of deploying explicit pointers, the physical contiguity of records can imply a connection. For example, the *GET NEXT* command in IMS processes the data-base tree in a sequential manner. The records can be allocated to reflect this sequential nature. Such allocation saves space and increases access speed for the *GET NEXT* type of processing, for, depending on the buffering characteristics of the operating system, the next record occurrence is often already in the buffer. Hence, no additional secondary storage access is necessary in this case of sequential allocation. However, data base reorganization problems arise for cases where a record is inserted or deleted. Certain overhead costs are also incurred by skipping groups of records when only a certain type of record is wanted. In addition to these problems, changes to the schema, which are difficult in any organization, are particularly difficult in this approach.

Pointers can be combined with physical contiguity to implement a hierarchical data base. This approach results in many special purpose access methods for storing hierarchical data bases. We will outline, as an example, the approach taken by IMS.

IMS has two basic storage organizations: hierarchical sequential and hierarchical direct [14]. Within each of these organizations, the access method may optionally use an indexed organization. The indexed organization is similar to B-trees [3]. In the simplest form of indexed organization, the records are ordered by a key data item value and divided into blocks. Each block contains a fixed number of records. A directory indicates the highest key value stored in each block. In a more general organization, there are several levels of directories. In this manner, the block in which a record is stored can be quickly determined from its key value. The block is then scanned sequentially to find the desired record.

In the hierarchical sequential organization, records are related by physical adjacency. The *Hierarchical Sequential Access Method* (*HSAM*) organization stores all records in physically adjacent storage locations in hierarchical (preorder) sequence. All records must be of fixed length. To modify a HSAM data base, the entire data base must be reloaded.

The *Hierarchical Sequential Indexed Access Method* (*HISAM*) organization is used for indexed access to a data-base tree. Each data-base tree is stored in physically contiguous locations in hierarchical sequence. The root record type must contain a key data item. The key data item is used to index each data-base tree. The storage area is divided into a primary and an overflow area. The primary area stores, in hierarchical sequence, a fixed size part of a data-base tree consisting of the root record and as many records of the data-base tree as can be accommodated. Additional records of the data-base tree are stored, in hierarchical sequence, in the overflow area. A direct address pointer relates a data-base tree in the primary area to its extension in the overflow area.

In the hierarchical direct organization, records are related by pointers. There are two pointer organizations possible: hierarchical and physical child/physical twin. Hierarchical pointers relate records in hier-

archical sequence. Physical child/physical twin pointers relate all records of a given type under a parent record to each other, and the parent to its first child. Both pointer organizations may optionally have backward pointers. It is possible to specify any combination of pointer organizations within a data base and different organizations for different record types.

The pointers in the hierarchical direct organization are stored with each record. The record consists of two parts in this case: a prefix and a data part. The data part contains the record data as supplied by the user. The prefix, which is system controled and not available to an application, contains system data and the pointers. The system data consists of a record code, a delete flag, and a counter. The record code identifies the record type, and the delete flag indicates whether the record has been deleted. The counter is optional and is only present if the record type participates in a logical relationship [14]. A simple prefix consisting of the record code and the delete flag is stored with every record in a multiple record type HISAM data base.

Within the hierarchical direct organization, the *Hierarchical Direct Access Method* (*HDAM*) organization is used to access root records via a hash algorithm. Records are hashed into a primary storage area called the *root segment addressable area.* The root record type must contain a key data item. The hash is performed on the key data item. A fixed portion of a data-base tree, including the root record, is stored in the root segment addressable area. Additional records of a data-base tree are stored in an overflow area. A direct address pointer relates records of a data-base tree in the root segment addressable area with its extension in the overflow area.

The *Hierarchical Indexed Direct Access Method* (*HIDAM*) organization provides indexed direct access to records in a data base. The index in this organization is a sequential file called INDEX. Each record in the INDEX file contains the key data item value of a root record and a pointer to the root record in the data base. Root records are accessed by searching for the key data-item value in the INDEX and then following the pointer to the data base. Records in the data base are related by pointers, as discussed previously.

To summarize, a successful hierarchical implementation can utilize contiguity and pointers with the following objectives:

- it is important to have efficient retrieval by eliminating costly pointer chasing and resulting secondary storage accesses;
- the space required both for the data and the pointers should be minimized; and
- costly reorganizations should be avoided by providing a flexible environment that allows easy expansion of the data base.

There are other ways of implementing a hierarchical data base. For example, a method can be used to assign a logical address to a record in a data-base tree. This logical address can then be mapped into a physical one. A logical address to a record in a data-base tree is called a *trace* [16]. An example of assigning traces to records in a data-base tree will be outlined.

Each record type in a hierarchical definition tree can be identified by a *type-number* as indicated in Figure 9(a). Any record occurrence in a data-base tree can then be identified by its type-number and a *generation tuple.* The generation tuple defines a path, in the data-base tree, which leads to the record occurrence.

For example, suppose that Figure 9(b) corresponds to the third data-base tree in a hierarchical data base. The root record type of this tree is identified by the trace 1 (3). The number 1 is the type-number of the record type. The number 3 indicates that this is the third record of type-1. The first type-2 child of this root is assigned the trace 2 (3, 1)—That is, it is the first type-2 child under the third root record. The first type-4 descendant under the first type-3 child of the third root record has as its trace 4 (3, 1, 1). The first number identifies the record type. The numbers in parentheses define a path to a particular record occurrence. Using a type-number and a generation tuple, any record occurrence in the data base can be uniquely addressed by the path to it. In this rep-

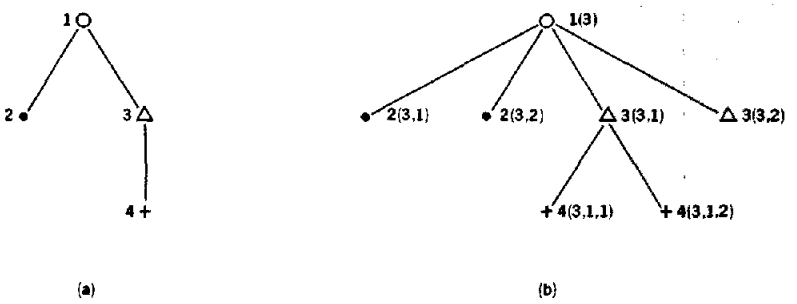


FIGURE 9. Assigning traces to records.

resentation, all traces are valid provided the bounds of the hierarchical definition tree are adhered to, i.e., type-numbers and levels are respected. However, traces may correspond to records which have not yet been inserted.

Given a trace, there is a straightforward algorithm to obtain the traces of ancestors, descendants, and brothers. The rules that govern our example representation of traces are:

1) *ancestor trace*—drop some (greater than level of ancestor) digits at the end of the generation tuple and change the type-number.
2) *Descendant traces*—add (descendant level minus current level) digits in next generation tuple places and change the type-number.
3) *Next brother trace*—add one to the last digit in the generation tuple.
4) *Previous brother trace*—subtract one from the last digit in the generation tuple (if digit $\neq 1$).

Sometimes, valid traces are restricted by specifying some bounds such as maximum number of levels in the hierarchical definition tree, maximum number of children at each level, etc.

Traces for record occurrences already in the data base are kept in a trace table. A *trace table* gives a mapping between a trace and the location where the record occurrence is stored. This mechanism captures all the structure of a hierarchical data base. Given the trace of a record occurrence, one can find its ancestors, the record occurrence itself, its brothers, and its descendants. If we can efficiently implement the trace table, then we have a nice implementation of the hierarchical data structure [4].

CONCLUDING REMARKS

To summarize the essential points made by this survey, a hierarchical system is a DBMS which presents to the users of the system certain explicit views of the data base that are characteristic of the hierarchical data model. The hierarchical data model has the following characteristics:

1) There is a set of record types $\{R1, R2, \ldots, Rn\}$.
2) There is a set of relationships connecting all record types in one data structure diagram.
3) There is no more than one relationship between any two record types Ri and Rj. Hence, relationships need not be labeled.
4) The relationships expressed in the data structure diagram form a tree with all arcs pointing toward the leaves.
5) Each relationship is $1{:}N$, and it is total—that is, for every Rj record occurrence there is *exactly one* Ri record occurrence connected to it, if Ri is the parent of Rj in the definition tree.

Hierarchical systems deal with relationships among attributes of the same entity in a manner similar to relational and network systems discussed in the companion papers in this issue. All of the systems organize the attributes as items in groups. Their main difference is found in the way they treat re-

lationships between entities. Relational systems provide operations on relations which construct new relations representing relationships among entities. As a result, relational systems do not use any additional concepts to the relationships among entities. The system is homogenous in that all relationships are represented by relations. Both hierarchical and network systems use the idea of a link or connection between record types to handle relationships between entities. In network systems, the data structure sets serve as the logical links between record types in a network. In the case of hierarchical systems, the parent-child relationships represented by the hierarchical definition tree are the links.

Hierarchical systems have been available and well accepted for a long time [5, 6, 7, 9, 10, 11, 14, 17, 20]. It is difficult to relate the success of a particular system to its data model. There are many other parameters which influence the quality of a commercial system. However, for some applications a hierarchical data model seems very natural, e.g., a corporate management structure is truly hierarchical. In addition, most applications can be modeled by a hierarchical organization of the data, although some applications produce more difficulty and redundancy than others.

A hierarchical data model provides no means for implementing direct $N:M$ relationships between record types. Such a relationship can only be effected within record types. However, most hierarchical systems do provide the ability to handle many hierarchical definition trees. By using data duplication, one can represent an $N:M$ relationship by two hierarchical definition trees, each representing a $1:N$ relationship. For instance, consider the record types STATE and COMPANY and the $N:M$ relationship "registration" between them. That is, a company may be registered in many states and a state may have many companies registered in it. The $N:M$ relationship beween STATE and COMPANY can be handled by using two hierarchical definition trees, one with root record type STATE and child record type REGISTRATION, the other with root record type COMPANY and child record type REGISTRATION.

Note that the $N:M$ relationship could be represented in a single hierarchical definition tree with one record type as the root and the other as the child. However, data duplication is again required: for example, if the root is STATE, then the COMPANY record occurrences would have to be repeated under each state in which the companies are registered. If the amount of data associated with each company is quite large, then a great deal of storage space is required for duplication. It should be noted, however, that some existing hierarchical systems have facilities which essentially eliminate this problem: they implement $N:M$ relationships by using logical pointers among hierarchies. They allow logical hierarchical views that are different from the physically implemented hierarchical structures [9, 14, 17].

Hierarchical systems also handle conceptually symmetric queries in a very different manner. For example, consider the relationship SERVED in Figure 1. If PRESIDENT is the root, then a query such as: "Find all congresses in which President P served," is simple to answer. For President P, all CONGRESS descendants are found. However, the symmetric query: "Find all presidents who served in Congress C," involves a data base search of CONGRESS records. For every President, it is necessary to determine if he served in Congress C. Sometimes, content addressibility, e.g., inverted files, may be used to speed the search. In addition, if two definition trees, one with the root CONGRESS and one with the root PRESIDENT, are used, then the problem disappears.

Some specific advantages are widely accepted for the hierarchical approach:

- It is a simple data model which provides the user with relatively few, easy to master, commands.
- Because of the constraints on the types of relationships allowed, it can allow an easier implementation than other, more complex structures.

Some specific disadvantages are also associated with hierarchical systems:

- The restrictions imposed force a sometimes unnatural organization of the data. For instance, as shown previously, $N:M$ relationships can sometimes only be represented in a clumsy way.
- Because of the strict hierarchical ordering, operations such as insertion and deletion become quite complex.
- A delete operation can lead to the loss of information present in the descendants if null records are not permitted. Consequently, users have to be careful when performing a delete operation.
- It is sometimes not possible to answer symmetrical queries easily in a hierarchical system. Therefore, the structure of the data base may tend to reflect the needs of the application.

Sometimes criticism is concentrated on the nature of the hierarchical commands which are claimed to be too procedural. However this criticism can be answered, as we have seen in the section, General Selection, see page 116, by showing that higher level interfaces can be implemented. It is in this way, that even a casual user interface can be easily accommodated.

REFERENCES

[1] ABRIAL, J. R., "Data semantics," *Data base management*, Klimbie, J. W., and Koffeman, K. L., [Eds.], North-Holland Publ. Co., Amsterdam, The Netherlands, 1974, pp 1–59.

[2] BACHMAN, C. W., "Data structure diagrams," *Data Base* **1**, 2 (1969), 4–10.

[3] BAYER, R.; AND MCCREIGHT, E., "Organization and maintenance of large ordered indexes," *Acta Informatica* **1**, 3 (1972), 173–189.

[4] BERNSTEIN, P. A.; AND TSICHRITZIS, D. C., "Allocating storage in hierarchical data bases," Technical Report CSRG-34, Computer Systems Research Group, Univ. of Toronto, May 1974 (to appear in *Information Systems Journal*).

[5] BLEIER, R. E., "Treating hierarchical data structures in the SDC time-shared data management system (TDMS)," in *Proc. ACM National Conf.* 1967, ACM, New York, 1967, pp 41–49.

[6] BLEIER, R. E.; AND VORHAUS, A. H., "File organization in the SDC time-shared data management system (TDMS)," in *Proc. IFIP Congress* 1968, Vol. 2, North-Holland Publ. Co., Amsterdam, The Netherlands, 1968, pp 1245–1252.

[7] CONTROL DATA CORP., "MARS VI multiaccess retrieval system reference manual," 44625500, 1970.

[8] CII, "SOCRATE manuel de presentation," Reference document 4337 P/FR, Louveciennes, France.

[9] DATAPRO RESEARCH CORP., "SYSTEM 2000—MRI Systems Corporation," *Datapro* 70, (April 1972).

[10] EVERETT, G. D.; DISSLY, C. W.; AND HARDGRAVE, W. T., "Remote file management system (RFMS) users manual," TRM-16, Computation Center, Univ. of Texas at Austin, Texas, August 1971.

[11] FRANKS, E. W.; "A data management system for time-shared file processing using a cross-index file and self-defining entries," in *Proc. AFIPS, Spring Jt. Computer Conf.*, 1966, Vol. 28, Spartan Books, New York, 1966, pp. 79–86.

[12] HARDGRAVE, W. T., "Theoretical aspects of Boolean operations on tree structures and implications for generalized data management," TSN-26, Computation Center, Univ. of Texas at Austin, Texas, August 1972.

[13] HARDGRAVE, W. T., "BOLTS: a retrieval language for tree-structured data base systems," *Information Systems*, COINS-IV, Plenum Press, New York, 1974.

[14] IBM, INFORMATION MANAGEMENT SYSTEM/VIRTUAL STORAGE (IMS/VS) PUBLICATIONS 1975:
General information manual, GH20-1260-3.
System/application design guide, SH20-9025-2.
Application programming reference manual, SH20-9026-2.
System programming reference manual, SH20-9027-2.
Operator's reference manual, SH20-9028-1.
Utilities reference manual, SH20-9029-2.
Messages and codes reference manual, SH20-9030-2.

[15] KNUTH, D. E., *The art of computer programming*, Vol. 1, Addison-Wesley Publ. Co., Reading, Mass., 1968, p. 316.

[16] LOWENTHAL, E. I., "A functional approach to the design of storage structures for generalized data management systems," PhD Thesis, Univ. of Texas at Austin, Texas, 1971.

[17] MRI SYSTEMS CORP., "SYSTEM *2000 general information manual*," Austin, Texas, 1972.

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

[19] SCHMID, H. A.; AND SWENSON, J. R., "On the semantics of the relational data model," in *Proc. ACM SIGMOD, Internatl. Conf. on Management of Data*, 1975, ACM, New York, 1975, pp. 211–223.

[20] UNITED COMPUTING SYSTEMS, INC., UCS-VI UNIDATA *data management system reference manual*, Kansas City, Missouri, 1970.