1984

The design and analysis of a complete hierarchical interface for the multi-backend database system.

Weishar, Doyle Joseph

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

THE DESIGN AND ANALYSIS OF A COMPLETE HIERARCHICAL INTERFACE FOR THE MULTI-BACKEND DATABASE SYSTEM

by

Doyle Joseph Weishar

June 1984

Thesis Advisor: David K. Hsiao

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Design and Analysis of a Complete Hierarchical Interface for the Multi-Backend Database System		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June 1984
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Doyle Joseph Weishar		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE June 1984
		13. NUMBER OF PAGES 90
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

database management, multi-backend database system (MDBS)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Typically, the design and implementation of a conventional database system begins with the choice of a data model, the specification of a model-based data language, and the design and implementation of a database system which controls and executes the transactions written in the data language. For example, we have the hierarchical model, the DL/I language and the IMS System. By using an unconventional approach (Cont)

ABSTRACT (Continued)

to the design and implementation of a basic database system, we can design a system to support multiple data models and several model-based languages as if the system is a heterogeneous collection of database systems.
In this thesis we present a methodology for supporting hierarchical database management on an attribute-based database system. Specifically, we construct an interface which translates Data Language/One (DL/I) calls into attribute-based data language (ABDL) requests. We describe the data structures, the control structures, and the functions required to implement this interface.

The Design and Analysis of a Complete Hierarchical Interface for the Multi-Backend Database System

by

Doyle Joseph Weishar
Captain, United States Army
B.S., United States Military Academy, 1975

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1984

ABSTRACT

Typically, the design and implementation of a conventional database system begins with the choice of a data model, the specification of a model-based data language, and the design and implementation of a database system which controls and executes the transactions written in the data language. For example, we have the hierarchical model, the DL/I language and the IMS System. By using an unconventional approach to the design and implementation of a basic database system, we can design a system to support multiple data models and several model-based languages as if the system is a heterogeneous collection of database systems.

In this thesis we present a methodology for supporting hierarchical database management on an attribute-based database system. Specifically, we construct an interface which translates Data Language/One (DL/I) calls into attribute-based data language (ABDL) requests. We descibe the data structures, the control structures, and the functions required to implement this interface.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Typically, the design and implementation of a conventional database system begins with the choice of a data model, the specification of a model-based data language, and the design and implementation of a database system which controls and executes the transactions written in the data language. Thus, we have the relational model, the SQL language and the SQL/Data System. Similarly, we have the hierarchical model, the DL/I language and the IMS system. We may also give an example in the case of the CODASYL model, language and system. The conventional approach to the design and implementation of a system is limited to a single data model, a specific data language and a homogeneous database system. By using an unconventional approach to the design and implementation of a basic database system, we can design a system to support multiple data models and several model-based languages as if the system is a heterogeneous collection of database systems.

This unconventional design and implementation approach reveals two important database concepts. First, there is an exceedingly simple and powerful data model such that many other data models may be realized easily by this data model. This is the attribute-based model. Second, the attribute-based database operations - being high-level and primary operations, are such that most of the other model-based language constructs can be mapped into this set of primary operations on a straightforward fashion. Furthermore, the system which implements these primary operations is relatively small in size and executable either in a single backend or in multiple backends. With these concepts, one can now use the attribute-based system to support many model-based interfaces. There could be an SQL interface so that the transactions written in SQL can be carried out. The execution of the transactions requires the SQL constructs to be transformed into the primary operations of the attribute-based system through the interface. Similarly, there could be a DL/I interface so that the transactions written in DL/I can also be executed. In this way, the multiple interfaces allow the system to support multiple data models and data languages as if it is a heterogeneous collection of database systems.

The attribute-based system supports the attribute-based data model, originally described in [Ref. 1] and extended in [Ref. 2]. Access to the attribute-based system is provided using the attribute-based data language known as ABDL. ABDL is a high-level data language which supports the primary database and aggregate operations, INSERT, DELETE, UPDATE, RETRIEVE, MIN, MAX, SUM, COUNT, and AVG. There are two distinct features in an attribute-based system. First, the system is easy to implement because the model and its operations are simple. Second, the directory information is well defined and easily structured in the model.

The attribute-based system supplies all the primary and aggregate operations required in a database system. With the specifications for another data language, we can construct an interface on top of the attribute-based system. In practice, we can construct a number of interfaces to support relational, hierarchical, and network operations with a minimal effort. Such an approach is clearly an attractive alternative to the approach where separate, stand-alone systems must be developed for specific models.

The procedure to construct a relational, hierarchical, or network interface is done at both the database and data language levels. At the database level, the series of papers [Ref. 3], [Ref. 4], and [Ref. 5] demonstrated that a relational, hierarchical, or network database can be converted into an attribute-based database. At the data language level, we focus on the development of language interfaces to the attribute-based system consistent with the user's chosen language. At this level, we address two issues. The first issue is to determine how the operations of the chosen language can be implemented using the operations of the attribute-based system. The second issue is the translation of the language of the interface to the attribute-based data language and the decisions regarding the interface mechanism. Although no implementation details are rendered, algorithms are provided to aid in the eventual implementation of the primitive mappings.

In this thesis, we investigate the design of a hierarchical interface for the multi-backend database system (MDBS). MDBS is an attribute-based database system, which is auto-configurable to either a single backend or to multiple backends. We are extending the work of [Ref. 5], which contains an initial design of a DL/I interface. In Chapter 2, the attribute-based model and hierarchical model are discussed. Also included in this chapter is an overview of the MDBS and the ABDL, and of IMS and DL/I. In Chapter 3, we illustrate a methodology for mapping a hierarchical database into an attribute-based database. In Chapter 4, the data structures used by the interface to translate DL/I calls to ABDL requests are examined. Chapter 5 shows the mappings of the DL/I calls to the ABDL requests. Although no implementation details are rendered, algorithms are provided to aid in the eventual implementation of the primitive mappings. In Chapter 6, we present interface implementation considerations, and a brief synopsis of additional considerations to reach the goal of a functional interface. And finally, in Chapter 7 we conclude the thesis.

II. AN OVERVIEW OF THE DATA MODELS

It is not our intent to describe in detail the data models of interest here, namely, the attribute-based data model and the hierarchical data model. Therefore, we only offer a brief overview of the pertinent aspects of these models.

A. THE ATTRIBUTE-BASED DATA MODEL

In this section we introduce the attribute-based data model. A conceptual view of the model is offered as well as a discussion of the data manipulation language that is associated with it. Finally, a system which is implemented upon the basis of the attribute-based model and language is discussed.

1. A Conceptual View

The attribute-based data model was originally described in [Ref. 1]. It is a basic model which incorporates a few simple concepts. As its name implies, it is built around the term attribute. Attributes and their associated values are represented by attribute-value pairs. An attribute-value pair is a member of the Cartesian product of the attribute name and the domain of values of the attribute. These pairs serve to represent all logical concepts within the attribute-based model. An attribute-value pair is otherwise known as a keyword. Keywords serve to form records, which are concatenations of keywords further concatenated with the record-body. Possibly empty, which is utilized for textual information. the record body is a string of characters which is utilized for textual information. An example of a which is utilized for textual information. record is as follows:

```
(<TYPE,COURSE>,<COURSE#,CS3112>,<TITLE,Operating Systems>
 {Operating Systems principles and techniques})
```

The angle brackets, <,>, enclose a keyword where the attribute and its value are separated by a comma. The curly brackets, {,}, enclose the record body. The entire record is enclosed with a pair of parentheses.

To access the database, we employ predicates. A keyword predicate, or simply predicate, is a triple of the form (attribute, relational operator, value). Combining keyword predicates in disjunctive normal form characterizes a query of the database. When the attribute of a keyword in a record is identical to the attribute in a predicate and the relation specified by the relational operator of the predicate holds between the value of the attribute and the value in the predicate, the keyword, and therefore the record, is said to satisfy the predicate. The query of two predicates

(TYPE = TEACHER) & (COURSE# = CS4112)

will be satisfied by all records of the teacher file whose teachers teach the course numbered CS4112.

2. The Multi-Backend Database System (MDBS)

The attribute-based model is implemented in an experimental database system called the multi-backend database system (MDBS). MDBS cannot be classified as either a distributed or nondistributed database system. One minicomputer functions as the controller, with multiple minicomputers and their disks configured in a parallel manner to serve as backends [Ref. 6], [Ref. 7], [Ref. 8], [Ref. 9], and [Ref. 10]. The database is distributed on the secondary storage across all of the backends. User access is accomplished through a host computer communicating with the controller. The MDBS structure can be classified as a centralized system.

As shown in Figure 1, the controller and the backends are connected by a broadcast bus. When a transaction is received from the host computer, the controller broadcasts the transaction to all the backends at the same time. Each backend has a number of dedicated disk drives. Since the data is distributed across the backends, a transaction can be executed by all backends in parallel. Each backend maintains a queue of transactions. When one transaction has been executed, the backend can begin execution on another transaction from its queue.

MDBS is implemented in several permanent processes. The process structure within the controller and each backend is shown in Figure 2. In addition to the processes listed, the controller and each backend have GET and PUT processes, which are used in the broadcast and reception of messages, respectively.

The controller is composed of three processes, Request Preparation, Insert Information Generation, and Post Processing. Request Preparation receives, parses and formats a request (transaction) before sending the formated request (transaction) to the Directory Management process in each backend. Insert Information Generation is used to provide additional information to the backends when an insert request is received. Since the data is distributed, the insert only occurs at one of the backends. Thus this process must determine the backend at which the insert will occur, along with the cluster and descriptor ids for the insert. Post Processing is used to collect all the results from a request (transaction) and forward the information back to the host computer.

Each backend is also composed of three processes, Directory Management, Concurrency Control, and Record Processing. Directory Management performs three functions, Descriptor Search, Cluster Search, and Address Generation.

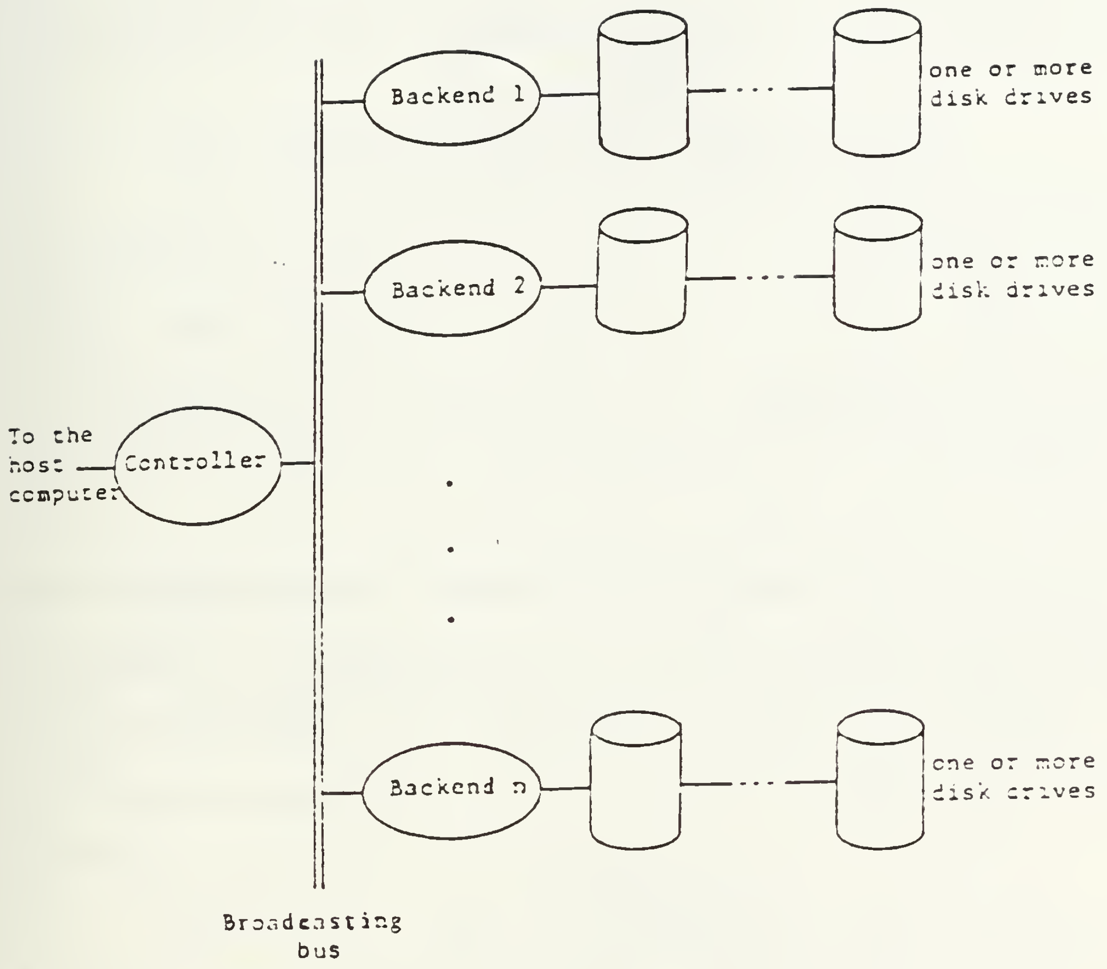

Figure 1. The MDBS Structure.

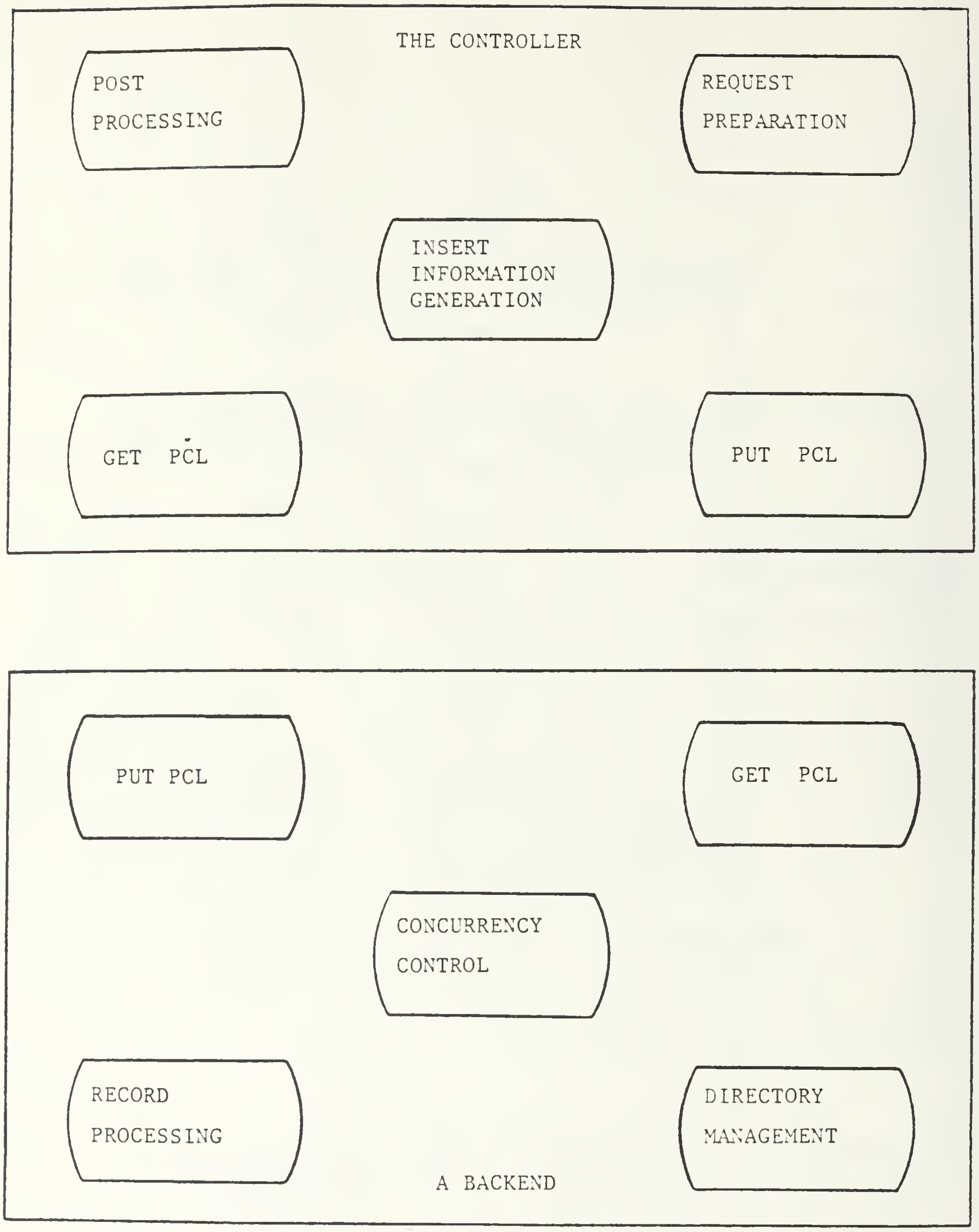

Figure 2. The Process Structure in the Controller and the Backends.

Descriptor Search determines the descriptor ids that are needed for a request. Cluster Search finds the cluster ids. Address Generation determines the secondary storage addresses necessary to process the request. Concurrency Control determines when the request can be executed. Record Processing performs the operation specified by the request.

3. The Attribute-Based Data Language (ABDL)

The Attribute-Based Data Language (ABDL) is designed to perform the primary database operations, INSERT, DELETE, UPDATE, and RETRIEVE. Through the host a user issues either a request or a transaction. A *request* is a primary operation along with a qualification. A *qualification* is used to specify the information of the database that is to be accessed by the request. It is defined in the next paragraph. A *transaction* is a list of two or more requests that are executed in a sequential order. There are four types of requests, corresponding to the four primary database operations.

Records are selected for retrieval by concatenating a query with a *target-list* and a *BY-clause*. A target-list is a list of elements. An element is either an attribute, e.g., Grade, or an aggregate operator to be performed on an attribute, e.g., AVG(Grade). ABDL supports five aggregate operators - AVG,SUM, COUNT,MAX, and MIN. The clause in the BY-clause is an attribute. The BY-clause is used to sort according to the values of the attribute. Records are inserted into the database by attribute-value pair. Records are deleted from the database by means of a query. Finally, records are updated by juxtaposing a query with a modifier. The query specifies which records of the database are to be changed and the modifier specifies how the records being changed are to be updated.

B. THE HIERARCHICAL DATA MODEL

In this section we introduce the hierarchical data model. We first offer a conceptual view of the model. Then, we discuss a database management system which incorporates the ideas inherent in the hierarchical model. And finally, we discuss a data manipulation language with which the database management system is implemented.

1. A Conceptual View

Hierarchies are a natural way to model a myriad of real-world applications. For example, businesses, baseball teams, political parties, and our elected representatives, all have units of information which can be organized using a hierarchical structure. The hierarchical structure is specified using hierarchical relationships, which represent a measure of precedence between units of information. Units of information, or data, are represented in a hierarchical model by entities. Entities have properties, called attributes, which uniquely identify each entity in an entity set. An entity set is simply a grouping of all similar entities. The relationships between entities can be represented by a graph called a <u>data structure diagram</u> (see Figure 3). In this diagram all entity and attribute relationships are one to many [Ref. 11]. These one to many relationships have a certain direction which is depicted by the directed arcs in the diagram. Each directed arc points from the one to the many relationship. For example, between record types COURSE and OFFERING, the arc representing the relationship PLANNED_FOR points from COURSE to OFFERING, since each course may have many offerings, but each offering is for only one course.

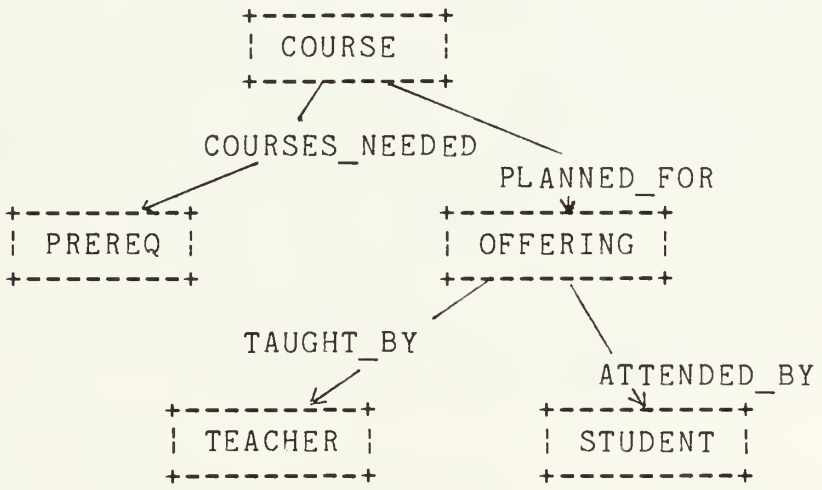

Figure 3. A Data Structure Diagram.

For the hierarchical model, the data structure diagram takes the form of a tree in which the direction of the arcs points away from the root. This tree has the restriction that there can be at most one arc between any two record types and is called a hierarchical definition tree (see Figure 4). The hierarchical definition tree specifies both what record types are allowed to be included in the database and the permissible relationships between record types. In this tree, the level of a record type is the measure of its distance from the root of the tree. The root record type is the highest level record type in the tree which, by convention, is referred to as level one. The other record types, called dependent record types, are at lower levels in the tree, i.e., at levels 2,3,4, and so on. Ancestor and dependent record occurrences can be identified by traversing the appropriate hierarchical path, which is simply a sequence of records in which the records, starting at the root record, follow alternately in a ancestor-dependent relationship. Referring to Figure 3, if one desired to find which teacher taught a Math course offered in Monterey, the hierarchical path would be from the COURSE record occurrence, to the OFFERING record occurrence, and then to the TEACHER record occurrence.

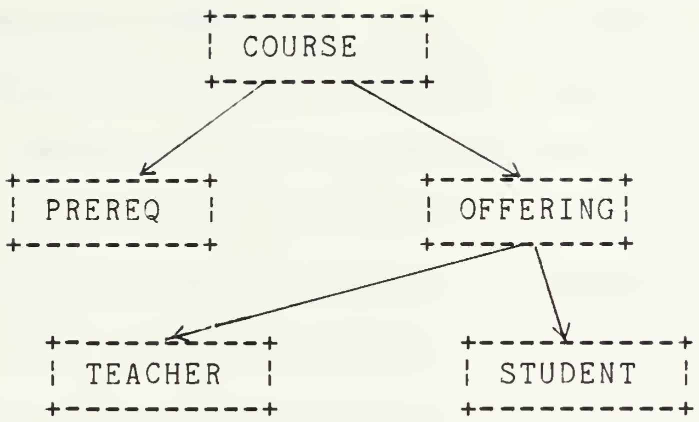

Figure 4. A Hierarchical Definition Tree.

A characteristic of the hierarchical conceptual model is that there can be a varying number of occurrences of each record type at each level. However, each record occurrence (except for the root record occurrence) must be connected to an occurrence of an ancestor record type. Because of this, each new record to be inserted (except for a root record occurrence) has to be connected to an occurrence of a parent type record. Deletions are also affected by this property. When a record occurrence is deleted, all of its descendent record occurrences are also deleted.

Records are retrieved according to a selection and qualification process. The qualification process expresses the selection criteria. A typical qualification takes the form:

<data item name><conditional operator><value>

connected by Boolean operators AND, OR, and NOT. The conditional operators are relational operators <, <=, >, >=, =, and < >. Qualification is performed along the hierarchical path of the selected record.

2. The Information Management System (IMS)

The Information Management System (IMS) is a product of International Business Machines (IBM) Corporation [Ref. 12], [Ref. 13], and [Ref. 14]. It uses the hierarchical data model. The smallest unit of logical data is called a *field* (data item). A *segment type* (record type) is a named collection of fields. Occurrences of segment types are called *segments* (records). An example of an IMS database is shown in Figure 5.

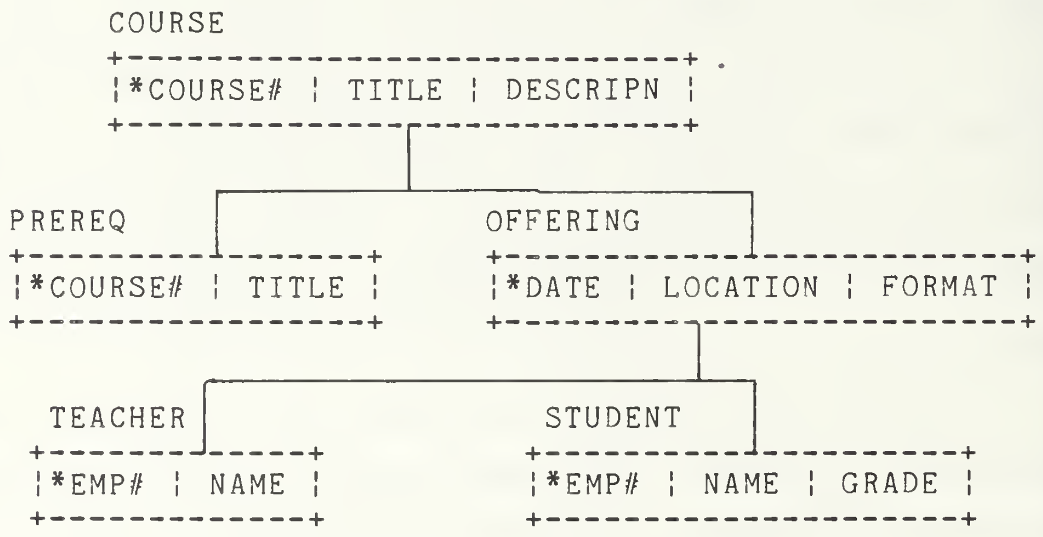

Figure 5. The Logical Data Structure of an IMS Database.

3. Data Language/One (DL/I)

The data manipulation language that IMS uses to respond to queries of this database is called Data Language/One (DL/I). Users issue calls using DL/I to access the database. The DL/I calls are used to traverse the database tree. DL/I performs a preorder tree traversal. This means that the traversal begins at the root record and then proceeds through the tree going in top-to-bottom, left-to-right order. Thus, in our previous example the hierarchical path IMS would take would be from COURSE, to PREREQ, to OFFERING, to TEACHER, and finally to STUDENT.

DL/I is designed to perform the primary database operations, GET, INSERT (ISRT), DELETE (DLET) and REPLACE (REPL). DL/I is invoked through procedure calls from applications programs written in PL/I, COBOL or Assembler Language. There are three types of calls, corresponding to the four primary database operations. Segments are selected for retrieval by means of one or more qualifications. DL/I qualifies segments by specifying a segment search argument (SSA). The form of the SSA is:

<SEGMENT NAME><COMMAND CODE><QUALIFICATION>

The SEGMENT NAME is the name of a segment type in the hierarchical definition tree i.e., COURSE, OFFERING STUDENT, etc. The QUALIFICATION is optional in the SSA and takes the form described above, with a minor exception. The only

Boolean operators allowed are AND and OR. The COMMAND CODE is also optional and delineates the various options of the call. Some of the more important options are:

- retrieval or insertion of some or all of the segments from the root to a specified segment type in a single DL/I call;
- backing up to the first child under a segment at any level;
- retrieval of the last occurrence of a segment that meets all specified conditions under a parent;
- setting of the parentage to a specific segment.

Segments are inserted into the database by segment search argument. SSAs are used to locate the position in the database tree in which the segment is to be inserted. Segments are deleted and modified in DL/I only after being retrieved. A DELETE call deletes a segment and all of its descendent segments from the database. A REPLACE call updates segments in the database.

III. MAPPING HIERARCHICAL DATA TO ATTRIBUTE-BASED DATA

Using a procedure originally outlined in [Ref. 5], we can map our sample hierarchical database into its ABDL counterpart. However, before doing so we must introduce and explain two notions whose existence are necessary to conduct the data conversion. These notions are that of the IMS _current position_ and that of the interface _symbolic identifier_.

A. THE NOTION OF CURRENT POSITION

IMS uses a pre-ordered traversal to navigate a database tree. Quite understandably, this traversal need not begin at the root each time a call is made to the database. The traversal could easily begin at a child segment. Indeed, the segment requested could be a twin of the segment just previously retrieved. Therefore, it is important to know the path of the traversal when conducting DL/I data manipulation operations. This is accomplished by designating the segment upon which the traversal has stopped as the _current position_. The current position of the IMS database is established after each retrieval or insertion operation. For a retrieval operation the current position is the segment just retrieved; for an insertion operation, the current position is the segment just inserted.

B. THE NOTION OF INTERFACE SYMBOLIC IDENTIFIER

In IMS it is necessary to indicate order among twin segments. This is achieved by designating a sequence field in the segment. As we convert our hierarchical data to attribute-based data, we also must be able to distinguish order among twin segments. Thus, in the conversion process we shall assign a symbolic identifier to each record. The symbolic identifier of a record R is a group of fields consisting of:

1) the symbolic identifier of the parent of R;

and

2) the sequence field of R.

C. THE CONVERSION OF THE IMS SEGMENTS

With the inclusion of the above notions, the database translation may now occur. An ABDL record may be created from an IMS segment using the following three step process:

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

Step 2: Form a keyword of the form <TYPE, SEGTYPE> where TYPE is a literal and SEGTYPE is the IMS segment type in consideration.

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

As an intermediary step it is helpful to utilize the above procedure to create attribute templates of the IMS database. Figure 6 illustrates these templates. These templates point out the attributes to be used in construction of the ABDL record and demonstrate the formation of the symbolic identifier, which in each template has been underlined. The final product of conversion is shown as Figure 7.

```
                         +-------------------+
                         | TYPE = COURSE     |
                         |*COURSE# =         |
                         | TITLE =           |
                         | DESCRIPN          |
                         +-------------------+

   +--------------------+       +-------------------+
   | TYPE = PREREQ      |       | TYPE = Offering   |
   |*COURSE# =          |       |*COURSE# =         |
   |*PREREQ.COURSE# =   |       |*DATE =            |
   | TITLE =            |       | LOCATION =        |
   +--------------------+       | FORMAT =          |
                                +-------------------+

      +------------------+        +--------------------+
      | TYPE = TEACHER   |        | TYPE = STUDENT     |
      |*COURSE# =        |        |*COURSE# =          |
      |*DATE =           |        |*DATE =             |
      |*TEACHER.EMP# =   |        |*STUDENT.EMP# =     |
      | NAME =           |        | NAME =             |
      +------------------+        | GRADE =            |
                                  +--------------------+
```

(the symbolic identifier is marked with an asterisk)

Figure 6. The attribute templates of MDBS records for the segments of Figure 5.

```
(<TYPE,COURSE>,<COURSE#,#_OF_COURSE>,<TITLE,COURSE_NAME>
               <DESCRIPN,DESCRIPTION>)

(<TYPE,PREREQ>,<COURSE#,#_OF_COURSE>,<PREREQ.COURSE#,
                #_OF_PREREQ>,<TITLE,COURSE_NAME>)

(<TYPE,OFFERING>,<COURSE#,#_OF_COURSE>,<DATE,WHEN>,
                 <LOCATION,LOCN>,<FORMAT,FORM>)

(<TYPE,TEACHER>,<COURSE#,#_OF_COURSE>,<DATE,WHEN>,
                <TEACHER.EMP#,TEACHER#>,<NAME,TEA_NAME>)

(<TYPE,STUDENT>,<COURSE#,#_OF_COURSE>,<DATE,WHEN>,
                <STUDENT.EMP#,STUDENT#>,<NAME,STU_NAME>,
                <GRADE,STU_GRADE>)
```

Figure 7. The Attribute-Based Representation of the Academic Database.

IV. DATA STRUCTURES NECESSARY TO EXECUTE DL/I CALLS

To effectively translate DL/I calls to ABDL requests the interface needs data structures to represent three tables and a series of buffers. These tables are the Status Information Table (SIT), the Hierarchy Table (HT) (see [Ref. 5]), and the Organization Table (OT). Each buffer in the series of buffers will be called an Interface Buffer (IB), or simply buffer. It should be noted that these are maintained for each user. That is, each user has his own set of tables and buffers.

A. THE STATUS INFORMATION TABLE AND THE HIERARCHY TABLE

The SIT and the HT are created in order to keep track of the current position of the database. These tables have as many entries as there are interface buffers. They are dynamic tables instantiated upon the first call to the database. Thereafter, they are updated in accordance with the various DL/I calls.

Each entry in the SIT consists of four fields: Seg(ment), Count, Addr(ess), and Qual(ification). The meaning of the i-th entry in the SIT is as follows:

SIT.Seg(i):	the name of the segment type of the i-th level of the hierarchical path;
SIT.Count(i):	the number of segments in the i-th buffer;
SIT.Addr(i):	the address of the segment within the i-th buffer;
SIT.Qual(i):	the SSA of the segment.

Each entry in the HT consists of two fields: F(ield) and V(alue). The meaning of the i-th entry of HT is as follows:

HT.F(i):	the sequence field name of the current position at level i;
HT.V(i):	the sequence field value of the current position at level i.

B. THE ORGANIZATION TABLE

The OT outlines the hierarchical structure of the entire database. This table lists all segment names contained in the database and stores the relationships among these segments. Although the hierarchical relationships of the database are maintained in the database translation, the complete descendent information is not available to the interface. When carrying out the translation of a DL/I DELETE, the ABDL system will need to know the names of all of the descendents of the segment identified for deletion. The OT provides this information.

Actual implementation of the OT can take several forms. However, we are suggesting that it be a list structure. A linear linked list will facilitate representation of a general tree and allow traversal of the OT, to extract the requisite descendent information. Each node in the list has five fields: Child, Seg, Sym_ID, SEQFLD, and Sibling. The meaning of the i-th entry in the OT is as follows:

OT.Seg(i):	the name of the segment type;
OT.Sym_ID(i):	the symbolic identifier of the segment;
OT.SEQFLD(i):	the sequence field of the segment;
OT.Child(i):	the pointer to the left-most child at level i+1;
OT.Sibling(i)	the pointer to the segment's sibling at level i.

Figure 8 illustrates the OT for our sample database.

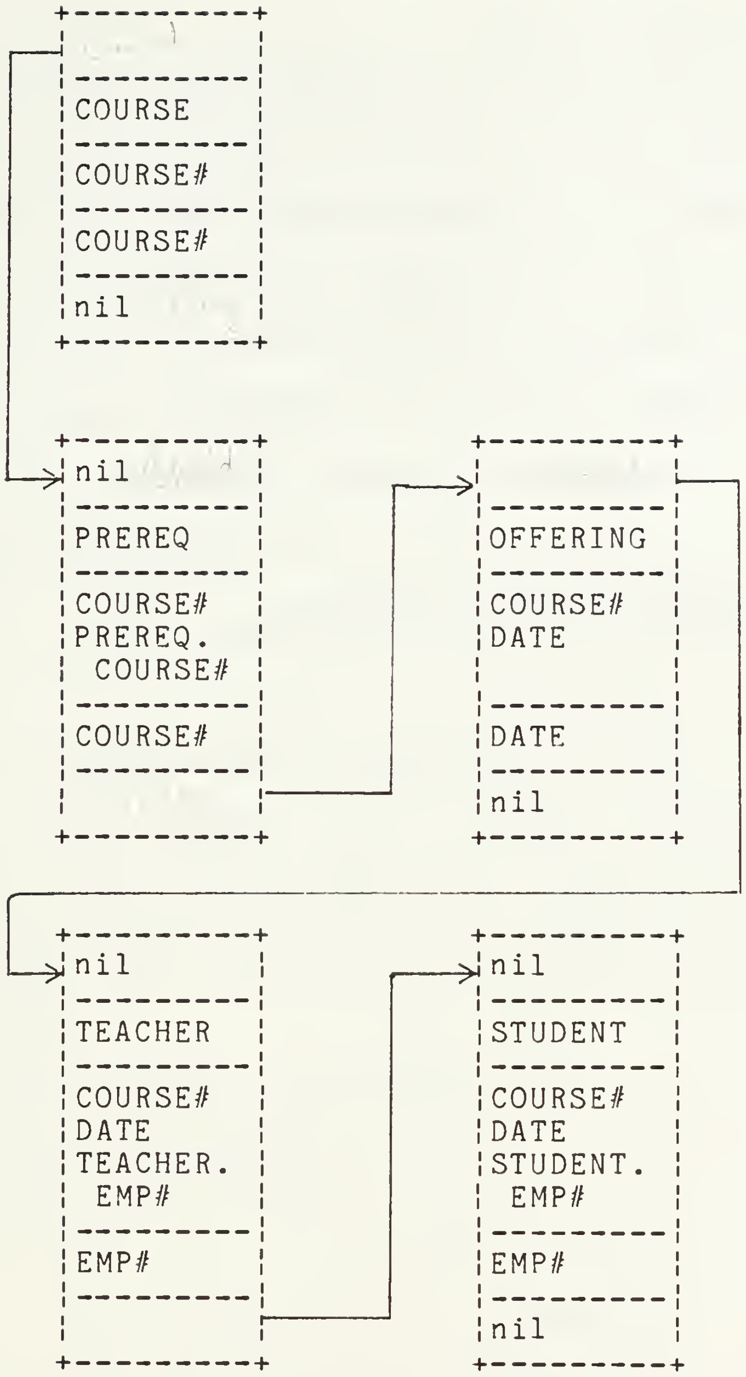

Figure 8. A List Representation of the OT.

C. THE INTERFACE BUFFER

The IB is simply a storage area utilized by the interface to store information needed to execute the translated DL/I calls. Although the exact role each buffer will play will be explained in the mapping of the DL/I calls, we can now say that a buffer is created for each operation which requires a retrieval. Upon a successful retrieval, all segment occurrences satisfying the query will be maintained in the buffer. This information will then be used for subsequent query execution.

V. MAPPING DL/I CALLS TO ABDL REQUESTS

We have demonstrated how hierarchical databases can be mapped into attribute-based databases. In this chapter we examine how calls in a hierarchical language, DL/I, can be mapped into the requests of the attribute-based data language, ABDL.

A. THE DL/I GET CALLS

The DL/I calls have been described earlier in our overview of DL/I. Each of these calls involves the retrieval of segment occurrences and, as such, are grouped together to be mapped to the ABDL RETRIEVE request. However, because each call is quite different in functionality, each must have an individual mapping to the ABDL RETRIEVE.

1. Mapping the DL/I Get Unique (GU) to the ABDL RETRIEVE

The general form of the DL/I GU is:

```
GU    Segment Search Argument(s)
```

The general form of the ABDL RETRIEVE is:

```
RETRIEVE Query Target-list [BY Attribute]
```

In order to successfully map the GU to the RETRIEVE, it is necessary to create an interface buffer (IB) as described earlier, which is used to store information retrieved from the database. The IB will be the mechanism through which movement up and down the hierarchical path is accomplished. Thus, at any one time it is likel-y that there will be multiple instances of the IB. The implementation, management, and placement of the IBs is discussed in Chapter 5.

To perform the mapping, the interface will first substitute the ABDL reserved word RETRIEVE for the DL/I reserved word -GU. Next, the interface takes the segment search argument (SSA) at level 1 and translates it into the ADBL query. This is a natural translation, since each segment occurrence is mapped into the ABDL database as a collection of keywords. Placing a relational operator between the attribute and value of these keywords results in a predicate, and a query is merely a collection of predicates. The final step in the mapping is the translation of the symbolic identifier into the target-list. Again, this is a natural translation since both the symbolic identifier and the target-list are collections of attributes. Having arrived at the end of the mapping, we can now explain the sequence of actions that will occur.

Basically, a DL/I GU call will result in a series of ABDL RETRIEVE operations; one RETRIEVE for each SSA in the

GU call. An example utilizing our sample database will help to illustrate the mechanics of the mapping where the requirement is as follows:

Get the first STUDENT occurrence who made an A in CS4900 at Monterey.

The DL/I call is:

```
GU COURSE    (TITLE = 'CS4900')
   OFFERING  (LOCATION = 'MONTEREY')
   STUDENT   (GRADE = 'A')
```

The interface would respond to this call by performing the following actions:

Step 1: The first RETRIEVE would be formed as such:

```
RETRIEVE ((TYPE = COURSE) & (TITLE = CS4900))
         (COURSE#)
```

The operation would result in having all COURSE# segments satisfying the query ((TYPE = COURSE) & (TITLE = CS4900)) placed into a buffer and sorted according to the values of their sequence field (see [Ref. 15]), which in this case is COURSE# (see Figure 9). The interface would then take the first segment in the buffer and form the call in step 2.

```
+--------------------------------+
|          <COURSE#1>            |
|          <COURSE#2>            |
|               .                |
|               .                |
|               .                |
|          <COURSE#n>            |
|                                |
|                                |
|                                |
|                                |
+--------------------------------+
```

Figure 9. The COURSE#s in Buf1.

```
    RETRIEVE ((TYPE = OFFERING) & (COURSE# = COURSE1) &
             (LOCATION = MONTEREY)
             (DATE)
```

As one can see, the RETRIEVE request is formed using the second SSA and the sequence field name and value. The sequence field names of the two segment occurrences serve as links along the hierarchical path. This action will bring all record occurrences satisfying the above query into a subsequent buffer (we shall call this buffer Buf2 and the aforementioned buffer Buf1). Again these records will be sorted according to the values of their sequence field, i.e., the attribute DATE (see Figure 10).

```
+--------------------------------------+
|               <DATE1>                |
|               <DATE2>                |
|                  .                   |
|                  .                   |
|                  .                   |
|               <DATEm>                |
|                                      |
|                                      |
|                                      |
|                                      |
|                                      |
|                                      |
+--------------------------------------+
```

Figure 10. The DATEs in Buf2.

We must mention here that if there are no records returned to Buf2 by the call, control is transferred back to step 1 where the next record in Buf1 will be retrieved using the operation called GET_NEXT_BUFREC(BUF#). This operation will move the pointer to the next record in the buffer. Upon completion of this operation, the action will proceed again to step 2.

Assuming that we have a record in Buf2, the interface shall again take the first record and form the call in step 3.

Step 3:

```
RETRIEVE (( TYPE = STUDENT) & (COURSE# = COURSE1) &
          ( DATE = DATE1)   & (GRADE = A))
          ( COURSE#,DATE,STUDENT.EMP#,NAME,GRADE)
```

The RETRIEVE request is formed as in previous steps. Likewise, the call will result in bringing all STUDENT segment occurrences satisfying the query in a subsequent buffer, Buf3 (see Figure 11).

```
+------------------------------------------------+
|     <COURSE1,DATE1,STUDENT#1,STU_NAME,A>       |
|                      .                         |
|                      .                         |
|                      .                         |
|     <COURSEn,DATEn,STUDENT#n,STU_NAME,A>       |
|                                                |
|                                                |
|                                                |
|                                                |
|                                                |
+------------------------------------------------+
```

Figure 11. The STUDENT records in Buf3.

Provided that segments were returned, these will be sorted by by their sequence field, i.e., by STUDENT.EMP#, and the first of these will be returned to the user. If no segments were retrieved, control will be returned to step 2 where the interface will choose the next record in Buf2.

The action will continue until the RETRIEVE query is satisfied or there are no more record occurrences in Buf1, at which time the user will be informed that the GU call was unsuccessful. The algorithm for the GU call is presented in Appendix A.

2. Mapping the DL/I Get Next (GN) to the ABDL RETRIEVE

The general form of the DL/I GN is:

GN Segment Search Argument

We map the Get Next to the ABDL RETRIEVE in a very similar fashion as we have mapped the GU. Upon encountering a GN, the interface will check the SIT for the current position of the database. This checking is performed because a DL/I GN retrieves the first occurrence of the specified segment following the current position. Thus, the interface must base upon this reference point to retrieve. Normally, a GN is preceded by a GU. Therefore, the segment we wish to retrieve is likely to be already available in the buffer which holds the current position. If this is the case, then the interface needs merely to return the next segment in that buffer; no additional retrieval is necessary. Of course, if this is not the case, the interface will perform the necessary retrieval(s) and bring the required segment into a buffer. Upon completion of the request, the SIT and the HT must be updated, as each instance of a GN re-establishes the current position in the database. The algorithm for the translation is presented The following example illustrates the mapping:

Retrieve the next segment who received an A in English.

The DL/I call is:

```
    GN COURSE     (TITLE = 'ENGLISH')
       OFFERING
       STUDENT    (GRADE = 'A')
```

Before proceeding, let us assume that the interface has just responded to the following DL/I call:

```
    GU COURSE      (TITLE = 'ENGLISH')
       OFFERING
       STUDENT     (GRADE = 'A')
```

Figures 12 through 15 represent the buffers that have been instantiated by the interface and the contents of the SIT. With this in mind, let us now proceed with the mapping.

```
+---------------------+
|     <COURSE#1>      |
|                     |
|                     |
|                     |
|                     |
|                     |
|                     |
|                     |
+---------------------+
```

Figure 12. Buf1.

```
+---------------------+
|      <DATE1>        |
|      <DATE2>        |
|      <DATE3>        |
|                     |
|                     |
|                     |
|                     |
|                     |
+---------------------+
```

Figure 13. Buf2.

```
+----------------------------------------+
| <COURSE#1,DATE1,STUDENT#1,STU_NAME,A>  |
|                                        |
|                                        |
|                                        |
|                                        |
|                                        |
|                                        |
|                                        |
+----------------------------------------+
```

Figure 14. Buf3.

level	Seg	Count	Addr	Qual
1	COURSE	1	xxx	TITLE = ENGLISH
2	OFFERING	3	yyy	
3	STUDENT	1	zzz	GRADE = A

Figure 15. The Status Information Table.

The interface would respond to this call by performing the following actions:

Step 1: The interface will first compare the hierarchical path stated in the call with the database currency information held within the SIT. Referring to Figure 15 we can see that indeed the segment search arguments match the Seg(ment) and Qual(ification) fields at all three levels. Having established this, we can now proceed to step 2.

Step 2: On the basis of the GU call we know that the first record in Buf3 is the current position of the database. Utilizing this information, the interface will check to see if there is a "next" segment in Buf3. If so, the interface will return the segment to the user in fulfillment of the request. However, in our example, there is no "next" segment. Therefore, the interface retracts one level and checks the contents of the corresponding buffer. The interface will determine if there is a subsequent segment in Buf2. If not, the interface will retract another level, and will continue to retract, until the request either fails or is completed. In our case, there is a subsequent record in Buf2, <DATE2>. With <DATE2> the interface will form an ABDL RETRIEVE. This retrieval is done in step 3.

Step 3:

```
    RETRIEVE ((TYPE = STUDENT) & (COURSE# = COURSE#1) &
              (DATE = DATE2) & (GRADE = A))
              (COURSE#,DATE,STUDENT.EMP#,NAME,GRADE)
```

Our request is satisfied as Figure 14. The first segment in this buffer is returned to the user. If there had been no segment returned, the interface would check Buf2 again for another subsequent segment. If successful, another retrieval would be formed. If not, the interface would retract another level as described in step 2.

3. Mapping the DL/I Get Next Within Parent (GNP) to the ABDL RETRIEVE

The general form of the DL/I GNP is:

GNP Segment Search Argument(s)

The mapping of the GNP to the ABDL RETRIEVE is identical to the mapping of the Get Next with one exception. When a GNP call is issued, the interface will return a segment occurrence that is either a sibling of a segment that has been previously retrieved which matches the SSA of the current segment, or is the first segment satisfying an ABDL RETRIEVE request for SSAn where n is greater than i in the SIT and HT. This difference can be visualized if we revert to our example for the DL/I Get Next. In that example we retrieved the "next" STUDENT segment that received an A in English. We could have achieved the exact same results with the following DL/I call:

```
    GNP COURSE    (TITLE = 'ENGLISH')
        OFFERING
        STUDENT   (GRADE = 'A')
```

However, for our GNP example, let us assume that the above call was made immediately after the GN call in the above example. Therefore, the situation is that the current segment is the segment in Buf3 (see Figure 14). Responding to the above call, the interface will proceed to check the existing buffers to see if the buffer information is useful. The interface will arrive in Buf3 and attempt to return the "next" segment occurrence in that buffer. However, Buf3 has only one occurrence. Therefore, the interface will retract to the next highest level ,Buf2, and check for subsequent segment occurrences. Since there is another segment occurrence in Buf2, i.e., <DATE3>, an ABDL RETRIEVE will be formed as follows:

```
    RETRIEVE ((TYPE = STUDENT) & (COURSE# = COURSE#1) &
              (DATE = DATE3) & (GRADE = A))
              (COURSE#,DATE,STUDENT.EMP#,NAME,GRADE)
```

We shall assume that the request is satisfied. Therefore, the first segment occurrence retrieved into our new Buf3 is returned to the user. Again, this is identical to the action for the Get Next call and does not show the subtle difference between the GN and the GNP calls. If we modify our example slightly, the difference becomes apparent. Let us assume that instead of the previously stated GNP call, the following Get Next Within Parent call occurred immediately after the aforementioned Get Unique call:

```
    GNP  STUDENT   (GRADE = 'A')
```

Notice that we now have the identical situation as described earlier, i.e., the current position of the database is the first segment in Buf3. With this in mind, we can now illustrate the difference between the algorithms (see Appendix A for the algorithm GNP).

Step 1: The interface first compares the segment search arguments (SSAs) of the GU call and the GNP call. The comparison is made by looping through the SIT entries, since the hierarchical path from the GU is stored in the SIT. While in most cases the GU call will provide the hierarchical path down to the level of retrieval (as it does here), there is no requirement for the GU to do so. As it relates to the GNP call, the main objective of the GU call is to establish the current position and to identify the "parent" segment. The "parent" segment for the GNP call is the lowest level SSA of the GU call. Since there may be any number of levels of the hierarchical path omitted from the last SSA of the GU call to the first SSA of the GNP call, the interface will need to discern this fact. If indeed there are missing SSAs, the interface must consult the OT in order to retrieve the appropriate segment occurrences into the buffers. This is accomplished in Step 3 of the algorithm (see Appendix A). Returning to our example, we find that the last SSA of the GU matches the first SSA of the GNP, i.e., STUDENT (GRADE = 'A'). The interface must now determine if there are any more SSAs in the GNP call. This is accomplished in step 2.

Step 2: In this step, the interface compares the SSAs of the GNP with the entries on the SIT. The reason that this is necessary is the essential difference between the Get Next and the Get Next Within Parent. Since the function of the GNP is to retrieve only segment occurrences within the parent, it is essential to know exactly who the parent is. As stated earlier, the "parent" is defined in the GU call by the last SSA. The segment type to be returned is the last SSA of the GNP call. These of course could be the same. In our example they are. This fact would be recognized by the interface, and since we have a buffer already in existence for this level, the interface would attempt to return the next record in that buffer. However, for our example there is no "next" record. Therefore, the interface would return a 'failure' to the user instead of returning a STUDENT occurrence for a different OFFERING, which would have been the result of a Get Next call. Thus, the essential difference between the GN and the GNP is clear. The Get Next in this case would start retracting to find the "next" segment, whereas the Get Next Within Parent just quits.

4. Mapping the DL/I Get Hold Calls to the ABDL RETRIEVE

DL/I has three Get Hold calls: the Get Hold Unique (GHU), the Get Hold Next (GHN), and the Get Hold Next Within Parent (GHNP). A Get Hold call is used in DL/I to retrieve into a work area and hold the record in that work area so that the record can be deleted or updated. ABDL does not have this requirement. Therefore, when the interface encounters a GHU call, a GHN call, or a GHNP call it will treat these calls as a GU call, a GN call, and a GNP call, respectively. With the exception of the "H", the general form of the Get Hold calls is identical to the forms of the non-hold counterparts. Therefore, the mappings described in the previous three sections are applicable to the Get Hold calls.

B. MAPPING THE DL/I ISRT TO THE ABDL INSERT

The general form of the DL/I ISRT is:

```
ISRT [Hierarchical Path SSAs]
     Unqualified Segment Type
```

A brief reminder, the DL/I ISRT traces SSAs along the hierarchical path in order to insert the unqualified segment type at a level we shall call n.

The general form of the ABDL INSERT is:

```
INSERT record
```

The mapping of the DL/I ISRT to the ABDL INSERT is facilitated by DL/I's rules. These rules mandate that:

1) With the exception of a root occurrence, the parent occurrence of the record to be inserted must already exist in the database;

2) The ISRT call must specify the complete hierarchical path to this parent;

3) The call must specify the type of the segment to be inserted.

With all of the information provided by the DL/I ISRT call, one might conclude that the ABDL mapping is simply a concatenation of transformed DL/I segments. However, for two reasons this is not the case. The first reason deals with the OT as introduced earlier. Although ancestor segment occurrences must already exist in the database, there is no such requirement for the segment type which is to be inserted. In order to perform its function, the OT must have complete knowledge of all parent-child relationships within the database. Thus, updating the OT is a required step for all insertions of new segment types; the second reason has to do with the current position. Recall that a DL/I ISRT call must establish the current position in the database in order to utilize DL/I Get Next and Get Next Within Parent calls. Our naive insertion would not have, nor could not have, this capability. We shall now proceed with the mapping.

To begin the mapping, the interface will utilize the ISRT SSAs specified to form ABDL RETRIEVE requests to retrieve segment ancestors into IBs in the same manner as the GU was conducted. However, instead of returning a "failure" if no segment is retrieved at level n, the interface will merely update the Organization Table and perform the insertion. The interface prepares for the insertion by getting the field names and values of the segment to be inserted from the DL/I work area. It then forms an ABDL INSERT statement of the form

```
INSERT (<Type,Sn>,<f1,v1>...<fi-1,vi-1>,<k1>...<km>
```

,where Sn is the segment name, f is a field name, v is the corresponding field value, and k1 to km are keywords formed from the DL/I qualifications for the segment to be inserted. With this request the mapping is complete (see Algorithm ISRT in Appendix B). The following example illustrates this mapping.

Requirement: Add a new STUDENT occurrence for the course entitled CS4112.

The DL/I call:

```
(build new segment in the I/O area)
ISRT COURSE   (TITLE = 'CS4112')
     OFFERING
     STUDENT
```

The interface would respond to this call by performing the following actions:

Step 1: The interface will respond to this call by forming an ABDL RETRIEVE request with the first SSA of the ISRT.

```
RETRIEVE ((TYPE = COURSE) & (TITLE = CS4112))
       (COURSE#)
```

This action will pull all COURSE# segments satisfying the request into Buf1 in the order of their sequence field values. The interface will then use the first of these to form the retrieval in step 2.

Step 2:

```
RETRIEVE ((TYPE = OFFERING) & (COURSE# = COURSE#1))
         (DATE)
```

This action will pull all DATE segments satisfying the request into Buf2 in order of their sequence field values. As in step 2, the interface will use the first of these to form the retrieval in step 3.

Step 3: Since this is the segment which is to be inserted the routine differs somewhat from the previous RETRIEVE requests, which were identical to those followed in carrying out the mapping for a GU. The request is of the following form:

```
RETRIEVE ((TYPE = STUDENT) & (COURSE# = COURSE#1) &
         (DATE = DATE1))    (STUDENT.EMP#)
```

Although the syntax is identical to the previous requests and the result is the same, i.e., all STUDENT.EMP# segments satisfying the request are sorted and placed in Buf3, the intent of the request is different. The purpose of this request is to check to see if there are any twin segment occurrences to the segment occurrence that is to be inserted. If there are occurrences, then the buffer will be utilized for the insertion. If not, then the interface must update the OT. The INSERT request is formed in step 4.

Step 4: Prior to forming the ABDL INSERT request, the interface will go to the DL/I I/O work area in order to retrieve the field names and values of the segment type to be inserted. Assuming that the STUDENT segment to be inserted has EMP# = 49, NAME = Zeke, and GRADE = A, the ABDL INSERT request is as follows:

```
INSERT (<TYPE,STUDENT>,<COURSE#,COURSE#1>,
        <DATE,DATE1>, <STUDENT.EMP#,49>,
        <NAME,ZEKE>,<GRADE,A>)
```

where the <COURSE#,COURSE#1> and <DATE,DATE1> represent fields and values of levels 1 and 2 respectively. With the successful completion of this request, the mapping comes to an end.

C. MAPPING THE DL/I DELETE TO THE ABDL DELETE.

The general form of the DL/I DELETE is:

DLET segment occurrence

The DLET call must be preceded by a GHU call, GHN call, or a GHNP call, which retrieves the segment occurrence and holds it in a work area so that the DLET can effectuate segment deletion. The general form of a DLET call will delete the specified segment occurrence and all of its children. The interface will use the OT to identify these segments for deletion.

The general form of the ABDL DELETE request is:

DELETE query

To perform the mapping we must first have the interface translate the GHU, GHN, or GHNP into a GU, GN, or a GNP. Once done, these commands will be translated as mentioned previously and the specified record occurrences will be held in the buffer. Next, the interface must make use of the OT in order to find all descendent segment occurrences of the segment earmarked for deletion. Having accomplished this, the mapping continues. The DL/I DLET will be translated into a number of ABDL DELETEs. This number will be determined based on the ancestry of the segment to be deleted. The number will be high if the deletion is of a root occurrence and low if the deletion is of a child. The reserved word DLET will be translated into ABDL's DELETE. The query part of the ABDL delete will be constructed from the symbolic identifier of the segment marked for deletion conjuncted with each descendent segment name.

As previously mentioned, these descendent segment names will determine the number of DELETE operations necessary in order to fully implement the DL/I DLET task. Beginning with the segment identified in the GHU, GHN, or GHNP, the OT will be traversed. Descendent segments will be alternatively RETRIEVEd and DELETEd by the interface. The action will stop once all dependent segments are deleted. The algorithm for the DLET call is presented as Appendix C. Note that a temporary SIT and HT have been established. These are necessary because a DLET does not alter the current position of the database. However, in order to form the ABDL RETRIEVEs and DELETEs, the interface must read the SIT and HT. If we do not update the SIT and the HT, there will be no entries for any levels below the last level in the SIT and HT. This, of course, will result in having segments left in the database that should have been deleted. On the other hand, if we update the SIT and the HT, we could re-establish the current position, which would be an unwanted side-effect. Therefore, we must have the temporary structures. An example call using our sample database will help to illustrate this mapping.

Delete the OFFERING occurrence for first Wine Tasting course offering in Monterey. The DL/I call:

```
    GHU COURSE   (TITLE = 'WINE TASTING')
        OFFERING (LOCATION = 'MONTEREY')
    DLET
```

The interface responds to this call by performing the following actions:

Step 1: The interface considers the DL/I GHU call to be the same as the DL/I GU call. Having done so, the first ABDL

RETRIEVE is formed:

```
    RETRIEVE ((TYPE = COURSE) & (TITLE = WINE TASTING))
              (COURSE#)
```

Step 2: As with the GU, a second retrieval is formed using the first record in Buf1 satisfying the request in step 1.

```
    RETRIEVE (( TYPE = OFFERING) & (COURSE# = COURSE#1)
             & (LOCATION = MONTEREY))            (DATE)
```

The result of this step is to retrieve all satisfying records into Buf2 and to designate the first of these for deletion. Deletion for this segment is carried out in step 3.

Step 3:

```
    DELETE ((TYPE = OFFERING) & (COURSE# = COURSE#1) &
            (DATE = DATE1)
```

This request will complete the task of deleting the segment occurrence but will not suffice for completion of the DL/I DLET. To do so, the interface must delete all descendent segment occurrences. In order to accomplish this, the interface enters the Organization Table with the pointer to the first child segment of the segment deleted in step 3. For our example let us say that the descendent segment occurrences are comprised of one TEACHER segment occurrence and 10 twin STUDENT occurrences. These segments will be alternately retrieved and deleted. The next two DELETEs illustrate the retrieval and deletion for the first two dependent segments. Note the absence of the accompanying RETRIEVEs. These requests were not necessary, since we are at a leaf in the traversal sequence.

```
    DELETE ((TYPE = TEACHER) & (COURSE# = COURSE#1)
             & (DATE = DATE1))

    DELETE ((TYPE = STUDENT) & (COURSE# = COURSE#1)
             & (DATE = DATE1))
```

Upon completion of all of the deletions for the dependent segments the action will be completed entirely.

D. MAPPING THE DL/I REPL (REPLACE) TO THE ABDL UPDATE

The general form of the DL/I REPL call is as follows:

```
REPL
```

Like the DL/I DLET call, the REPL call must be preceded by one of the Get Hold calls. The Get Hold call serves to retrieve the appropriate record into a work area so that the record may be modified. After the record is modified in the work area, the DL/I REPL call is issued which makes the modification permanent.

The general form of the ABDL UPDATE request is

```
UPDATE query modifier
```

where the query specifies which records of the database are to be updated, and the modifier specifies how the records are to be changed.

The mapping of the DL/I REPL call to the ABDL RETRIEVE proceeds initially with the interface translating the Get Hold call into the appropriate Get call. This action retrieves the record to be modified. Recalling our earlier discussion in the ISRT translation, we can apply the same logic as to not by-passing the translation of the Get Hold call in favor of the straightforward "one-step" translation, i.e., we must establish the current position. Therefore, once the Get call is translated, the interface will use the symbolic identifier of the segment to be modified as the query portion of the ABDL UPDATE. For the final step in the mapping, the interface will retrieve the update information from the DL/I work area and use this for the modifier. The algorithm for the mapping is presented as Appendix D. The following example illustrates the mapping:

Change the prerequisite of Course# 4 from Math to Discrete Math.

The DL/I call to accomplish this is as follows:

```
GHU    COURSE  (COURSE# = '4')
       PREREQ
change title to 'Discrete Math' in I/O work area
REPL
```

The interface would respond to this call by treating the Get Hold Unique call as a Get Unique call. Steps 1 and 2 show the formation of the appropriate ABDL RETRIEVE calls.

Step 1:

```
RETRIEVE ((TYPE = COURSE) & (COURSE# = 4))
          (COURSE#)
```

Step 2:

```
RETRIEVE ((TYPE = PREREQ) & (COURSE# = 4))
          (PREREQ.COURSE#)
```

Recalling the actions involved in the RETRIEVE request, we know that the first segment in Buf2 is the segment to be modified. Therefore, the interface will form the query portion of the ABDL UPDATE as follows:

```
(TYPE = PREREQ) & (COURSE# = 4) &
(PREREQ.COURSE# = COURSE#1)
```

Upon accomplishing this, the interface will proceed to the DL/I work area in order to get the update information. With this information, the modifier portion of the ABDL UPDATE request is formed, i.e., <TITLE = DISCRETE MATH>. The entire ABDL UPDATE call is formed by concatenating the "query" portion of above to the modifier as follows:

```
UPDATE ((TYPE = PREREQ) & (COURSE# = 4) &
        (PREREQ.COURSE# = COURSE#1))
        <TITLE = DISCRETE MATH>
```

Upon execution of this request, the call is completed.

VI. IMPLEMENTATION CONCERNS AND ADDITIONAL INTERFACE CONSIDERATIONS

In this chapter we present interface implementation concerns, and a brief synopsis of additional considerations to reach the goal of a functional interface. Specifically, we shall discuss the location of the interface, the combining of DL/I calls, and the implementation of DL/I segment search argument (SSA) command codes.

A. THE LOCATION OF THE INTERFACE

We have discussed the interface, thus far, in terms of functionality, without mention of the exact location of the interface within the overall database system. As we see it, there are four location possibilities. These are: 1) placing the interface in a separate location, i.e., within its own processor; 2) placing the interface within the host processor; 3) placing the interface within the MDBS controller; and 4) placing the interface in one of the MDBS backends. Additionally, there is a fifth option. That is, the interface can be distributed among one or more of the aforementioned locales. Of these possibilities, we recommend the adoption of option 2, i.e., placing the interface within the host. We make this recommendation for several reasons. First of all, if one situates the interface within a separate processor, or distributes it among parts of the system, one compounds the interprocess communication problems of the system. Secondly, if one places the interface within the controller, one jeopardizes the ability of the controller to perform its function, i.e., the controller would be in danger of being overloaded. Thirdly, if one places the interface within a backend, one undermines the intent of the MDBS system. The backends were specifically designed for data management functions only. And finally, it just makes sense to place the interface in the host. This is because the interface can make use of the resident database interface structures located within the host.

B. COMBINING DL/I CALLS

A preponderance of DL/I calls to the database can occur in combination. For example, it is standard to see a Get Unique followed by a Get Next, and a Get Hold Unique followed by a DLET. Therefore, the interface must be able to distinguish among these calls, and place combinations of calls in the correct sequence. In order to accomplish this, it is incumbent upon the interface to be able to update the individual user's SIT and HT throughout the user's session.

C. IMPLEMENTATION OF THE SEGMENT SEARCH ARGUMENT COMMAND CODES

The SSA command codes were discussed in Chapter 2. As described earlier, these are special codes which allow variations to the basic DL/I calls. In order to fully implement a functional interface, algorithms for these codes must be developed. In this section we discuss some of the details necessary for their eventual implementation. We shall limit our discussion to three command codes, D, F, and V, which are the most prevalent. For a discussion of the remaining codes (C,L,P,Q,U,N,-), see [Ref. 13] and [Ref. 14].

1. <u>The Command Code D</u>

The command code D permits retrieval, update, or insertion of some or all of the segments from the root to a specified segment type in a single DL/I call. For example,

```
GU COURSE * D
   OFFERING  (LOCATION = 'MONTEREY')
```

will retrieve not only the segment satisfying the OFFERING SSA, but will also retrieve the COURSE parent segment. The interface must be able to recognize this. This should not be a difficult modification to the basic GU algorithm. For example, there could be a conditional which would be activated upon recognizing the D in the SSA. This conditional would send the appropriate segment to the user.

Similar modifications can also be made to the ISRT and REPL algorithms.

2. The Command Code F

The command code F provides a means of stepping backwards under the current parent. This is important in situations where it is desired to retrieve a sibling that precedes the current segment. For example, suppose we desired to retrieve the name of the teacher of Jones attending Course# 1 in Monterey. Without the command code there is no way for us to do this, since TEACHER and STUDENT are siblings. We could possibly form two DL/I GU calls, but each of these would return segments that, when placed together, would not necessarily satisfy the original call. With this command code we can form the DL/I calls as follows:

```
GU  COURSE (COURSE# = '1')
GN  OFFERING (LOCATION = 'MONTEREY')
GNP STUDENT (NAME = 'JONES')
GNP TEACHER * F
```

The interface must be able to recognize that the current parent is the OFFERING segment satisfying (LOCATION = 'MONTEREY'), and must be able to backtrack in order to retrieve the correct TEACHER segment. The modification to the GNP algorithm necessary is, that upon recognizing the command code F, the interface must consult the SIT and HT in order to locate the buffer holding the current position, and use the current position as the basis for retrieval.

3. The Command Code V

One uses the command code V in a very similar fashion as the Get Next Within Parent. The subtle difference can only be understood, however, by first expanding our explanation of the notion of current position. As stated earlier, the current position is defined as the segment last accessed via a "get" or "insert" operation. However, this is not the entire story. Each segment along the hierarchical path to the current segment is considered as the current of that particular segment type. For example, if the segment last retrieved is a TEACHER, then that TEACHER is the current segment, the TEACHER's parent is the current OFFERING, and the OFFERING's parent is the current COURSE. Recall that a GNP retrieves segments only from the current parent (in our example, the OFFERING segment). By using the command code V, any ancestor can be designated as the "current parent", i.e., we can choose the COURSE segment instead of the OFFERING segment. Thus, the use of command code V directs IMS away from the current segment type named in the SSA to which it is appended in much the same fashion as the Get Next Within Parent.

The command code V is used with a Get Next call. By proposing a more specific example than our earlier one, we can illustrate the use of the command code V. Suppose that it is desired to get the next Teacher whose name is Smith. The code for our example would be as follows:

```
GU  TEACHER  (NAME = 'SMITH')
GN  COURSE*V
    OFFERING
    TEACHER  (NAME = 'SMITH)
```

Note that the use of the command code V does not require the presence of a preceding GU call in order to reposition the user to the start of the database. This will cause no problem with either the algorithm GU or the algorithm GN since we require a GN call to specify the entire hierarchical path. However, the GN algorithm must be modified in order to recognize the presence of the command code. The modification to the algorithm focuses upon recognizing the V, at which point the GU algorithm will call the Get Next Within Parent algorithm, sending the SSA with the V appendage as a parameter.

VII. RESULTS AND CONCLUSIONS

As stated earlier, by using an unconventional approach to the design and implementation of a basic database system, we can design the system to support multiple data models as if the system is a heterogeneous collection of database systems. Our unconventional approach is geared to flexibility, efficiency, and extensibility, which makes it an attractive alternative to conventional approaches. By developing multiple data language interfaces we offer users the alternative of our unconventional approach without incurring any retraining costs. In adopting our system, users appear to have their same old database system, but one that works faster.

In this thesis we have presented a methodology for supporting hierarchical database management on an attribute-based database system. Specifically, we have constructed an interface which translates DL/I calls into ABDL requests, and which maintains appropriate buffer and table contents. We have described the additional data structures, control structures, and functions required to implement this interface. Finally, we have shown that DL/I calls can be mapped to ABDL requests in a relatively straightforward manner. Based upon this information, the hierarchical interface can be implemented.

Although the hierarchical interface can be implemented based upon the work we have presented, we must caution that this work has addressed only the hierarchical model. Two other interfaces must also be completed to correspond with the other two popular data models, i.e., the relational and network models. [Ref. 16] and [Ref. 17] have designed an interface for the relational model. However, the network interface is still yet to be developed. Given the fact that two of the three interfaces have been designed, it is possible that implementation can proceed in these areas. However, the implementor(s) must proceed with caution and must pay particular attention to commonalities and overlapping of functions between the two interfaces. It is one thing to strive ahead, and yet another to strive ahead blindly.

APPENDIX A - THE GET ALGORITHMS

A. THE ALGORITHM GET UNIQUE (GU)

This algorithm executes the following DL/I call:

```
GU    S1 Q1
      S2 Q2
      .
      .
      .
      Sn Qn
```

where each Si is a segment type at level i, each Qi is a qualification (possibly null) and n >= 1. We assume that the sequence field name of segment type Si is Fi. The target list is defined as the sequence field up to level n-1. At level n, the target list is a list of all fields requested in the original DL/I call.

```
Step 1:    (Retrieve root segments into
            buffer and update SIT, HT)
           RETRIEVE (( TYPE = S1) & Q1)
                    (target list)
           sort attribute F1, buffer address a,
              count c
           SIT(1) <-- (S1,c,a,Q1)
           let(F1,V1) be the sequence field
              of the segment in address a
           HT(1) <-- (F1,V1)


Step2:     (All segments retrieved?)
           i <-- i + 1
           if 1>n then
               go to step 6
```

```
Step3:     (Retrieve segments at i-th level)
           RETRIEVE (( TYPE = S1) & (F1 = V1)
                    & ...
                    & (Fi-1 = Vi-1) & Q1)
                    (target list)
           sort attribute Fi, buffer address a,
              count c
           if c <> 0 then
                go to step 5


Step 4:    (Retract one level and try again)
           i <-- i-1
           if i = 0 then
              return ('failure', -)
           (Si,c,a,Qi) <-- SIT(i)
           c <-- c-1
           if c = 0 then
              go to step 4


Step 5:    (update SIT,HT)
           SIT(i) <-- (Si,c,a,Qi)
           let (Fi,Vi) be the sequence
              field of the segment
              in address a
           HT(i) <-- (Fi,Vi)


Step 6:    (Operation Successful)
           number of entries in SIT or HT <-- n
           current position of database <-- n
           parent position <-- n
           return ('success', buffer address a)
```

The Algorithm GU.

B. THE ALGORITHM GET NEXT (GN)

This algorithm executes the following DL/I call:

```
                    GN     S1 Q1
                           S2 Q2
                             .
                             .
                             .
                           Sn Qn
```

where each Si is a segment type in level i, each Qi is a qualification (possibly null) and n >= 1. In checking to see if at any time the SSA of the GN call precedes the corresponding SIT entry in the traversal sequence, we assume that the code for a segment name A is less than the code for a segment name B if A precedes B in the traversal sequence; m is the number of entries in the SIT or HT. The target list is defined as the sequence field up to level n-1. At level n, the target list is a list of all fields requested in the original DL/I call.

```
Step 1:   (Find t such that the condition
           ((Si = SIT.Seg(i)) &
           (Qi = SIT.Qual(i))) is satisfied
            for 1 <= i <= t
            but not for i = (t+1).)
          t <-- 0
```

```
Step 2:   (Compare the SIT with each SSA)
          t <-- t+1
          if t > n or t > m then
             go to step 3
          (Ft,Vt)  <-- HT(t)
          if (St = SIT.Seg(t)) &
             (Qt = SIT.Qual(t)) then
             go to step 2


Step 3:   (Get rid of any unnecessary buffers)
          t <-- t-1
          while t <= m do
             clear Buf(t)


Step 4:   (No buffer information is useful?)
          if t = 0 then
             go to step 10


Step 5:   (Perhaps the necessary segment
           is in the buffer?)
          1 <= t, but is t = n <= m?)
          if t = n then
             i <-- i+1
             go to step 14


Step 6:   (Entire buffer information is useful?
           1 <= t and m < n, but is t = m?)
          if t = m then
             i <-- i+1
             go to step 12


Step 7:   (S(t+1) precedes SIT.Seg(t+1))
          if S(t+1) < SIT.Seg(t+1) then
             return ('failure', -)
```

```
Step 8:   (S(t+1) does not
           precede SIT.Seg(t+1))
          if S(t+1) > SIT.Seg(t+1) then
             i <-- t+1
             go to step 12


Step 9:   (1 <= t < m < n, S(t+1) = SIT.Seg(t+1),
          Q(t+1) <> SIT.Qual(t+1))
          i <-- t+1
          RETRIEVE ((TYPE = Si) & (F1 = V1)
                   &...
                   & (Fi-1 = Vi-1) & Qi)
                   (target list)
          sort attribute Fi, buffer address a,
             count c
          go to step 13


Step 10:  (Retrieve root segments into buffer and
          update SIT,HT)
          RETRIEVE ((TYPE = S1) & (F1 = V1)
                    & Q1)
                   (target list)
          sort attribute F1, buffer address a,
             count c
          SIT(1) <-- (S1,c,a,Q1)
          HT(1)  <-- (F1,V1)


Step 11:  (All segments retrieved?)
          i <-- 1+1
          if i > n then
             go to step 16


Step 12:  (Retrieve segments at i-th level)
          RETRIEVE ((TYPE = Si) & (F1 = V1)
                    &...
                   & (Fi-1 = Vi-1) & Qi)
                   (target list)
          sort attribute Fi, buffer address a,
             count c
```

```
Step 13:  (Any segments retrieved?)
          if c <> 0 then
             go to step 15


Step 14:  (Retract one level and try again)
          i <-- i-1
          if i = 0 then
             return ('failure',-)
          (Si,c,a,Qi) <-- SIT(i)
          c <-- c-1
          if c = 0 then
             go to step 14


Step 15:  (Update SIT,HT)
          SIT(i) <-- (Si,c,a,Qi)
          let (Fi,Vi) be the sequence
              field of the segment
              in address a
          HT(i) <-- (Fi,Vi)
          go to step 11


Step 16:  (Operation successful)
          number of entries in SIT or HT <-- n
          current position of database <-- n
          parent position <-- n
          return('success',buffer address a)
```

The Algorithm GN.

C. THE ALGORITHM GET NEXT WITHIN PARENT (GNP)

This algorithm executes the following DL/I call:

```
GU      S1   Q1
        S2   Q2
        .
        .
        .
        Sn   Qn

GNP     Sb   Qb
        .
        .
        .
        Se   Qe
```

where the Get Unique call is as previously specified, each Sb through Se is a segment type in levels b through e, each Qb through Qe is a qualification (possibly null) in levels b through e, b >= e, and e >= 1. The target list is defined as the sequence field up to level n-1. At level n, the target list is a list of all fields requested in the original DL/I call.

```
Step 1:   (Compare the SIT and Sb)
          t <-- 0
          Repeat
             t <-- t+1
             (Ft,Vt) <-- HT(t)
          Until
             (Sb = SIT.Seg(t)) &
             (Qb = SIT.Qual(t))
              OR (t > n)
```

```
Step 2:    (Check to see if the first SSA
            of the GNP matches the SIT)
           if (Sb = SIT.Seg(t)) &
              (Qb = SIT.Qual(t))
              then go to step 3
           else if t > n then
              go to step 4


Step 3:    if b < e then
           (There is more than one SSA
            in the GNP)
           t <-- b
           LOOP:   t <--  t+1
                   if t > e or t > m then
                      go to Step 4
                   (Ft,Vt)  <-- HT(t)
                   if (St = SIT.Seg(t)) &
                      (Qt = SIT.Qual(t)) then
                          go to LOOP
           else      ( b = e)
              c <-- c-1
              if c = 0 then
                 return ('failure',-)
              else
                 go to Step 15


Step 4:    (We must retrieve further along the
           hierarchical path without establishing
           the current position)
           TEMPSIT <-- SIT
           TEMPHT  <-- HT
           i <-- t-1
           While i <= e do
              RETRIEVE ((TYPE = Si) & (Fi = Vi)
                       &...
                       & (Fi-1 = Vi-1) & Qi)
                       (target list)
              sort attribute Fi, buffer address a,
                   count c
              if c = 0 then
                 return ('failure',-)
              TEMPSIT(i) <-- (Si,c,a,Qi)
              TEMPHT(i)  <-- (Fi,Vi)
              i <-- i+1
           i <-- e
           go to step 18
```

```
Step 5:   (Clear any unnecessary buffer)
          t <-- t-1
          while t <= m do
             clear Buf(t)


Step 6:   (No buffer information is useful?)
          if t = 0 then
             go to step 12


Step 7:   (Perhaps the necessary segment
             is in the buffer?)
          1 <= t, but is t = e <= m?)
          if t = e then
             i <-- i+1
             go to step 16


Step 8:   (Entire buffer information is useful?
           1 <= t and m < n, but is t = m?)
          if t = m then
             i <-- i+1
             go to step 14


Step 9:   (Check to see if the desired segment
           precedes the current position in
           the traversal sequence)
          if S(t+1) < SIT.Seg(t+1) then
             return ('failure', -)


Step 10:  if S(t+1) > SIT.Seg(t+1) then
             i <-- t+1
             go to step 14
```

```
Step 11:  (1 <= t < m < n, S(t+1) = SIT.Seg(t+1),
          Q(t+1) <> SIT.Qual(t+1))
          i <-- t+1
          RETRIEVE ((TYPE = Si) & (F1 = V1) &...
                   & (Fi-1 = Vi-1) & Qi)
                   (target list)
          sort attribute Fi, buffer address a,
             count c
          go to step 15


Step 12:  (Retrieve root segments into buffer
          and update SIT,HT)
          RETRIEVE ((TYPE = S1) & (F1 = V1)
                   & Q1)
                   (target list)
          sort attribute F1, buffer address a,
             count c
          SIT(1) <-- (S1,c,a,Q1)
          HT(1)  <-- (F1,V1)


Step 13:  (All segments retrieved?)
          i <-- 1+1
          if i > e then
             go to step 18


Step 14:  (Retrieve segments at i-th level)
          RETRIEVE ((TYPE = Si) & (F1 = V1)
                   &...
                   & (Fi-1 = Vi-1) & Qi)
                   (target list)
          sort attribute Fi, buffer address a,
             count c


Step 15:  (Any segments retrieved?)
          if c <> 0 then
             go to step 17
```

```
Step 16:  (Retract one level up to level b and
           try again)
          i <-- i-1
          if i = b-1 then
             return ('failure',-)
          (Si,c,a,Qi) <-- SIT(i)
          c <-- c-1
          if c = 0 then
             go to step 16


Step 17:  (Update SIT,HT)
          SIT(i) <-- (Si,c,a,Qi)
          let (Fi,Vi) be the sequence field
              of the segment in address a
          HT(i) <-- (Fi,Vi)
          go to step 13


Step 18:  (Operation successful)
          number of entries in SIT or HT <-- e
          current position of database <-- e
          parent position <-- e
          return('success',buffer address a)
```

The Algorithm GNP.

APPENDIX B - THE ALGORITHM ISRT

This algorithm executes the following DL/I call:

```
ISRT     S1      Q1
         S2      Q2
         .
         .
         .
         Sn-1    Qn-1
         Sn
```

where each Si is a segment type in level i, each Qi is a qualification (possibly null) and n >= 1. We assume that the sequence field name of segment type Si is Fi. The target list is defined as the sequence field up to level n-1. At level n, the target list is a list of all fields requested in the original DL/I call.

```
Step 1:   (Retrieve root segments into Buf1
             and update SIT,HT)
          i <-- 1
          RETRIEVE ((TYPE = S1) & Q1)
                    (target list)
          sort attribute F1, buffer address a,
             count c
          if (c = 0) & (n > 1) then
             return ('failure',-)
          if (c = 0) & (n = 1) then
             update OT
             c <-- 1
             go to step 7
          SIT(1) <-- (S1,c,a,Q1)
          let (F1,V1) be the sequence field
              of the segment in address a
          HT(1) <-- (F1,V1)
```

```
Step 2:    (All ancestor segments retrieved?)
           i <-- i+1
           if i > (n-1) then
              go to step 6


Step 3:    (Retrieve segments at i-th level)
           RETRIEVE (( TYPE = Si) & (F1= V1)
                    & ...
                    & (Fi-1 = Si-1) & Qi)
                    (target list)
           sort attribute Fi, buffer address a,
                count c
           if c <> 0 then
              go to step 5


Step 4:    (Retract one level and try again)
           i <-- (i-1)
           if i = 0 then
              return ('failure',-)
           (Si,c,a,Q1) <-- SIT(i)
           c <-- (c-1)
           if c = 0 then
              go to step 4


Step 5:    (Update SIT,HT)
           SIT(i) <-- (Si,c,a,Qi)
           let (Fi,Vi) be the sequence
               field of the
               segment in address a
           HT(i) <-- (Fi,Vi)
           go to step 2


Step 6:    (Check to see if there are
              any twin segments)
           RETRIEVE ((TYPE = Sn) & (F1 = V1) & ...
                    & (Fi-1 = Vi-1)
                    (target list)
           if c = 0 then
              update OT
              c <-- 1
```

```
Step 7:   (Make the insertion)
          get field values of Sn from
              DL/I I/O work area
          INSERT (<TYPE,Sn>,<F1,V1>,...,
                  <Fi-1,Vi-1>,
                  <k1>...<km>)
          SIT(i) <-- (Si,c,a,Qi)
          HT(i) <-- (Fi,Vi)


Step 8:   (Operation successful)
          number of entries in SIT or HT <-- n
          current position of database <-- n
          parent position <-- n
          return ('success', buffer address a)
```

The Algorithm ISRT.

APPENDIX C - THE ALGORITHM DLET

This algorithm executes the following DL/I call:

```
GH[U][N][NP]        S1  Q1
                    S2  Q2
                    .
                    .
                    .
                    Sn  Qn

DLET
```

where each Si is a segment type at level i, each Qi is a qualification (possibly null) and n >= 1. We assume that the sequence field name of segment type Si is Fi. The target list is defined as the sequence field up to level n-1. At level n, the target list is a list of all fields requested in the original DL/I call.

```
Step 1:  Case Call =
              GHU  :  Translate GHU into GU
              GHN  :  Translate GHN into GN
              GHNP :  Translate GHNP into GNP
         Execute the GU, GN, or GNP
```

```
Step 2:  (Enter the OT with current_segment.child)
         nodeptr  <--  current_segment.child
         TEMPSIT  <--  SIT
         TEMPHT   <--  HT
         Procedure Pretrav (nodeptr)
            q  <--  nodeptr
            While q <> nil do
               Read node[q]
               If q.childptr < > nil then
                  RETRIEVE ((TYPE = SEG_NAME) &
                           (F1 = V1) &...
                           & (Fi-1 = Vi-1) & Qi)
                           (target list)
                  i <-- i+1
                  TEMPSIT(i) <-- (Si,c,a,Qi)
                  TEMPHT(i)  <-- (Fi,Vi)
               DELETE ((TYPE = Si) &
                     (F1 = V1) & ...
                     & (Fi-1 = Vi-1) & Qi
               Pretrav(child)
               Pretrav(sibling)
            end while
         end procedure Pretrav
```

The Algorithm DLET.

APPENDIX D - THE ALGORITHM REPL

This algorithm executes the following DL/I call:

```
GH[U][N][NP]    S1 Q1
                S2 Q2
                .
                .
                .
                Sn Qn

REPL
```

where each Si is a segment type at level i, each Qi is a qualification (possibly null) and n >= 1. We assume that the sequence field name of segment type Si is Fi. Aj is an attribute of field j, whose value will replace the old value of field j.

```
Step 1:  Case Call =
              GHU  :  Translate GHU into GU
              GHN  :  Translate GHN into GN
              GHNP :  Translate GHNP into GNP
         Execute the GU,GN, or GNP


Step 2:  (Form the "query")
         ((TYPE = Si) & (F1 = V1) & ...&
         (Fi-1 = Vi-1) & Qi)


Step 3:  (Form the "modifier")
         go to I/O work area to get
             update information
         <Aj = Vj>
```

```
Step 4:  (Perform the request)
         UPDATE ((TYPE = Si) & (F1 = V1) & ... &
                (Fi-1 = Vi-1) & Qi) <Aj = Vj>
```

The Algorithm REPL.

LIST OF REFERENCES

1. Hsiao, D. K., and Harary, F., "A Formal System for Information Retrieval from Files," Communications of the ACM, Vol. 13, No. 2, February 1970, Corrigenda, Vol 13., No. 3, March 1970.

2. Wong, E., and Chiang, T. C., "Canonical Structure in Attribute Based File Organization," Communications of the ACM, September 1971.

3. Banerjee, J. and Hsiao, D. K., "A Methodology for Supporting Existing CODASYL Databases with New Database Machines," Proceedings of National ACM Conference, 1978.

4. Banerjee, J., Buam, R. I. and Hsiao, D. K., "Concepts and Capabilities of a Database Computer," ACM Transactions on Database Systems, Vol. 4, No. 1, December 1978.

5. Banerjee, J., Hsiao, D. K., and Ng, F., "Database Transformation, Query Translation and Performance Analysis of a Database Computer in Supporting Hierarchical Database Management," IEEE Transactions on Software Engineering, March 1980.

6. Demurjian, S. A., et al., "The Implementation of a Multi-Backend Database System (MDBS): Part IV - The Revised Concurrency Control and Directory Management Processes and the Revised Definitions of Inter-Process and Inter-Computer Messages" Technical Report, NPS-52-84-005, Naval Postgraduate School, Monterey, California, March 1984.

7. He, X., et al., "The Implementation of a Multi-Backend Database System (MDBS): Part II - The Design of a Prototype MDBS," in Advanced Database Machine Architecture, Hsiao (ed), Prentice Hall, 1983.

8. Hsiao, D. K. and Menon, M. J., "Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part I)," Technical Report, OSU-CISRC-TR-81-7, The Ohio State University, Columbus, Ohio, July 1981.

9. Hsiao, D. K. and Menon, M. J., "Design and Analysis of a Multi-Backend Database System for performance Improvement, Functionality Expansion and Capacity Growth (Part II)," Technical Report, OSU-

CISRC-TR-81-8, The Ohio State University, Columbus, Ohio, August 1981.

10. Kerr, D. S., et al., "The Implementation of a Multi-Backend Database System (MDBS): Part I - Software Engineering Strategies and Efforts Towards a Prototype MDBS," Technical Report, OSU-CISRC-TR-82-1, The Ohio State University, Columbus, Ohio, January 1982.

11. Lochovsky, F. H., and Tsichritzis, D. C., "Hierarchical Data-Base Management: A Survey," Computing Surveys, Vol. 8, No. 1, March 1976.

12. IBM., "Information Management System IMS/360, Application Description Manual (Version 2)," IBM Corp., White Plains, New York, 1971.

13. IBM., "Information Management System/Virtual Storage, General Information Manual," IBM Corp., White Plains, New York, 1975.

14. Date, C. J., An Introduction to Database Systems, Addison Wesley, 1982.

15. Muldur, S., The Design and Analysis of Join and Ordering Operations for a Multi-Backend Database System, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1984.

16. Macy, G., Design and Analysis of an SQL Interface for a Multi-Backend Database System, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1984.

17. Rollins, R., Design and Analysis of a Complete Relational Interface for a Multi-Backend Database System, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1984.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
4.	Curriculum Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943	1
5.	Professor David K. Hsiao, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943	1
6.	Steven A. Demurjian, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943	2
7.	Doyle J. Weishar 400 Ponce Drive Independence, Missouri 64056	3